

## What is JavaScript?

JavaScript is a high-level, interpreted programming language primarily used for client-side web development. It is one of the core technologies of the World Wide Web and enables dynamic and interactive elements on websites. JavaScript code is typically embedded directly within HTML files and executed by web browsers.

\*\*\* **A high-level language** is a programming language that is designed to be human-readable and easier to understand compared to low-level languages like assembly or machine code. High-level languages use natural language constructs and abstractions that are closer to human language, making it more intuitive for programmers to write code. They are generally portable and independent of the hardware architecture.

\*\*\* **scripting language** is a type of programming language that is often used for automating tasks or creating scripts to control software applications.

### \*\*\*Interpreted Programming Language:

In an interpreted programming language, the source code is executed line by line at runtime by an interpreter without the need for a separate compilation step. The interpreter reads each line, translates it into machine code or bytecode, and executes it immediately.

Examples of interpreted programming languages include **Python, JavaScript, Ruby**

1. **JavaScript:** As mentioned earlier, JavaScript is primarily an interpreted language. It is executed by JavaScript engines found in web browsers or server-side environments like Node.js. The engines interpret the JavaScript code and execute it dynamically.

### \*\*\*Compiled Programming Language:

In a compiled programming language, the source code is first translated into machine code or bytecode through a separate compilation process. This compiled code can be directly executed by the target platform without further translation.

Examples of compiled programming languages include **C, C++, JAVA**.

It's worth mentioning that the line between interpreted and compiled languages can sometimes be blurry, as there are hybrid approaches and optimizations employed by modern programming language implementations. Some languages, like JavaScript, utilize a combination of interpretation and Just-in-Time (JIT) compilation techniques to improve performance.

Here are some key points about JavaScript:

1. **Client-side scripting:** JavaScript is primarily used for client-side scripting, meaning it runs on the user's web browser rather than on the web server. It allows web developers to create interactive web pages by manipulating the Document Object Model (DOM) and responding to user events.

\*\*\* **DOM (Document Object Model)** is a programming interface that represents the structure of a web page as a tree, allowing developers to manipulate its elements using code.

- **Document:** Refers to the web page or XML document being displayed in the browser.
- **Object:** Represents each element or entity in the document, such as HTML tags, attributes, text, etc.

- **Model:** Describes how these objects are organized and their relationships with each other, forming a tree-like structure.
2. **Versatility:** JavaScript is a versatile language that can be used for a wide range of tasks, including form validation, dynamic content updates, creating animations, handling asynchronous requests, building web applications, and much more.
  3. **Syntax and structure:** JavaScript syntax is like that of other programming languages like C, C++, and Java, making it relatively easy to learn if you're familiar with those languages. It follows an object-oriented programming (OOP) paradigm and supports functional programming concepts as well.
  4. **Cross-platform compatibility:** JavaScript is supported by all major web browsers, including Chrome, Firefox, Safari, and Edge. It can also be used on various platforms beyond web browsers, such as server-side environments (e.g., Node.js) and mobile app development frameworks (e.g., React Native).

5. **Third-party libraries and frameworks:** JavaScript have a vast ecosystem of third-party libraries and frameworks that extend its capabilities and simplify common tasks. Popular libraries and frameworks include React.js, Angular.js, Vue.js, jQuery, and many others.

**\*\*\*Library: A library is a collection of pre-written code that provides specific functionalities. Developers can selectively use components from the library in their projects based on their needs.**

Example: jQuery is a library for JavaScript that simplifies DOM manipulation and AJAX interactions.

**\*\*\*Framework: A framework is a comprehensive set of tools, libraries, and conventions that provide a foundation and structure for building applications. Frameworks dictate the overall architecture and flow of an application.**

Example: Angular is a framework for building web applications that provides a structured approach to development, including components, routing, and data management.

6. **Security considerations:** While JavaScript enables powerful functionality, it also introduces potential security risks, such as cross-site scripting (XSS) attacks. Web developers need to be mindful of security best practices and validate user inputs to prevent vulnerabilities.

**\*\*\*XSS (Cross-Site Scripting)** is a type of security vulnerability where attackers inject malicious scripts into web pages viewed by users. These scripts can execute unauthorized actions, steal sensitive information, or manipulate the website's content.

### **Variables in JavaScript:**

In JavaScript, variables are used to store and manipulate data. Here are some key points about variables in JavaScript:

**Variable declaration:** Variables in JavaScript are declared using the **var**, **let**, or **const** keywords.

### 1. var:

- **var** was traditionally used to declare variables in JavaScript.
- Variables declared with **var** are function-scoped and can be redeclared and reassigned.
- **var** variables are hoisted, meaning they are moved to the top of their scope during the execution phase.

Example:

```
function example() {  
  var x = 10;  
  if (true) {  
    var x = 20;  
    console.log(x); // Output: 20  
  }  
  console.log(x); // Output: 20  
}
```

In this example, the variable **x** is declared with **var** within the function scope. Inside the if block, a new variable **x** is created, effectively overwriting the previous value of **x**.

### 2. let:

- Variables declared with **let** are block-scoped and can be reassigned but not redeclared.
- **let** variables are not hoisted, meaning they are only accessible after they have been declared.

Example:

```
function example() {  
  let x = 10;  
  if (true) {  
    let x = 20;  
    console.log(x); // Output: 20  
  }  
  console.log(x); // Output: 10  
}
```

In this example, the variable **x** is declared with **let** within the function scope. Inside the if block, a new block-scoped variable **x** is created, which does not affect the value of the outer **x** variable.

### 3. const:

- Variables declared with **const** are block-scoped and cannot be reassigned or redeclared.
- **const** variables must be assigned a value upon declaration and cannot be changed thereafter.

Example:

```
function example() {  
  const x = 10;  
  if (true) {  
    const x = 20;  
    console.log(x); // Output: 20  
  }  
  console.log(x); // Output: 10  
}
```

In this example, the variable **x** is declared with **const**. Since **const** variables cannot be reassigned, a new block-scoped variable **x** is created within the if block, but it does not affect the value of the outer **x** variable.

To summarize:

- Use **var** for function-scoped variables that may need to be reassigned or redeclared.
- Use **let** for block-scoped variables that may need to be reassigned but not redeclared.
- Use **const** for block-scoped variables that should not be reassigned or redeclared.

## Naming Convention:

In JavaScript, naming conventions refer to the guidelines and rules used to name variables, functions, classes, and other identifiers in a consistent and readable manner. Following a standard naming convention makes code more maintainable and easier to understand for both the developer and others who might read or collaborate on the code. There are several naming conventions in JavaScript, but one of the most popular ones is called "camelCase."

In camelCase, identifiers are written without spaces, and each word is capitalized except for the first one. Here's an example:

```
// Variables  
let firstName = "John";  
let lastName = "Doe";  
let age = 30;  
let isStudent = true;  
  
// Functions  
function calculateSum(a, b) {  
  return a + b;  
}
```

**Note** that there are other naming conventions in JavaScript as well, such as PascalCase (similar to camelCase, but with the first letter of each word capitalized) for class names or snake\_case (all lowercase with underscores between words) which is also used in some coding styles. The most important thing is to choose a convention and stick to it consistently throughout your codebase.

## Data types in JavaScript

JavaScript supports several data types to represent different kinds of values. Here are the commonly used data types in JavaScript along with examples:

### 1. 1. Primitive Data Types:

- **String:**

**javascriptCopy code**

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

- **Number:**

**javascriptCopy code**

```
const age = 25;  
const pi = 3.14;
```

- **Boolean:**

**javascriptCopy code**

```
const isStudent = true;  
const hasLicense = false;
```

- **Undefined:**

**javascriptCopy code**

```
console.log(x)
```

- **Null:**

**javascriptCopy code**

```
const car = null;
```

### 2. Composite Data Types:

- **Object:**

**javascriptCopy code**

```
const person = {  
  name: 'Alice',  
  age: 30  
};
```

- **Array:**

**javascriptCopy code**

```
const numbers = [1, 2, 3, 4, 5];
```

- **Function:**

**javascriptCopy code**

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

### 3. Special Data Types:

- **NaN (Not-a-Number):**

#### javascriptCopy code

```
const result = 'abc' / 2; // NaN
```

- **Infinity and -Infinity:**

#### javascriptCopy code

```
const positiveInfinity = Infinity;  
const negativeInfinity = -Infinity;
```

### How to know the data types in java script?

#### // Printing data types in JavaScript

##### // Strings

```
console.log(typeof "Hello"); // Output: string
```

##### // Numbers

```
console.log(typeof 42); // Output: number
```

##### // Booleans

```
console.log(typeof true); // Output: boolean
```

##### // Arrays

```
console.log(typeof [1, 2, 3]); // Output: object
```

##### // Objects

```
console.log(typeof { name: "John", age: 30 }); // Output: object
```

##### // Functions

```
console.log(typeof function() {}); // Output: function
```

##### // Undefined

```
console.log(typeof undefined); // Output: undefined
```

**// Null**

`console.log(typeof null);` // Output: object (Note: This is a known quirk in JavaScript)

## **Type Casting in JavaScript**

Type casting, also known as type conversion, is the process of transforming a value from one data type to another in JavaScript.

### **Types of Type Casting:**

#### **Implicit (Automatic):**

- Occurs automatically during operations or assignments.
- Can lead to unexpected results if not understood properly.

#### **Examples:**

String to number conversion in math operations (e.g., "10" + 5 becomes 15).

Number to string conversion in concatenation (e.g., 5 + "hello" becomes "5hello").

#### **Explicit (Manual):**

You control the conversion using built-in functions:

- `Number()`: Converts to a number.
- `String()`: Converts to a string.
- `Boolean()`: Converts to a boolean (true or false).

#### **Example:**

JavaScript

```
let age = "25";
```

```
let ageAsNumber = Number(age); // ageAsNumber is 25 (number)
```

```
let username = 123;
```

```
let usernameAsString = String(username); // usernameAsString is "123" (string)
```



## **Operators in JavaScript:**

JavaScript provides a variety of operators that allow you to perform operations on values and variables. Here are some commonly used operators in JavaScript:

### **1. Arithmetic Operators:**

- Addition: +
- Subtraction: -
- Multiplication: \*
- Division: /
- Remainder (Modulo): %
- Increment: ++
- Decrement: --

### **2. Assignment Operators:**

- Assign: =
- Addition and assign: +=
- Subtraction and assign: -=
- Multiplication and assign: \*=
- Division and assign: /=
- Remainder and assign: %=

### **3. Comparison Operators:**

- Equal to: ==
- Not equal to: !=
- Strict equal to: ===
- Strict not equal to: !==
- Greater than: >
- Less than: <
- Greater than or equal to: >=
- Less than or equal to: <=

### **4. Logical Operators:**

- Logical AND: &&
- Logical OR: ||
- Logical NOT: !

## 5. Conditional (Ternary) Operator:

- Conditional expression: **condition ? expression1 : expression2**

## 6. Typeof Operator:

- Returns the data type of a value: **typeof**

## 7. String Concatenation Operator:

- Concatenates two strings: **+**

## 8. Bitwise Operators:

- Bitwise AND: **&**
- Bitwise OR: **|**
- Bitwise XOR: **^**
- Bitwise NOT: **~**
- Left shift: **<<**
- Right shift: **>>**
- Zero-fill right shift: **>>>**

In JavaScript, bitwise operators are used to manipulate individual bits of integer numbers. These operators treat their operands as sets of 32-bit binary digits (zeros and ones) and perform bitwise operations on them. The result is typically shown as a decimal value, although the operations are performed on binary representations.

Here are the bitwise operators in JavaScript along with their descriptions:

Bitwise AND (&): Sets each bit to 1 if both bits are 1.

Bitwise OR (|): Sets each bit to 1 if either of the bits is 1.

Bitwise XOR (^): Sets each bit to 1 if the corresponding bits are different.

Bitwise NOT (~): Inverts all the bits (1 becomes 0 and 0 becomes 1).

Example: for bitwise(AND)

```
let num1 = 5;    // Binary: 0101
let num2 = 1;    // Binary: 0001
let result = num1 & num2;
console.log(result); // Output: 1 (Binary: 0001)
```

Example: for bitwise(OR):

```
let num1 = 5; // Binary: 0101
let num2 = 1; // Binary: 0001
let result = num1 | num2;
console.log(result); // Output: 5 (Binary: 0101)
```

These are just a few examples of operators available in JavaScript. Operators enable you to perform mathematical computations, assign values, compare values, and make logical decisions within your JavaScript code.

## Control Flow in JavaScript

Control flow in JavaScript determines the order in which statements are executed based on conditions and loops. Here are some control flow statements in JavaScript with examples:

### 1. Conditional Statements:

- **if...else:** Executes a block of code if a condition is true; otherwise, executes a different block of code.

```
let age = 18;
if (age >= 18) {
    console.log("You are an adult.");
} else {
    console.log("You are a minor.");
}
```

- **switch...case:** Evaluates an expression and executes different blocks of code based on different cases.

```

let day = "Monday";
switch (day) {
  case "Monday":
    console.log("It's Monday.");
    break;
  case "Tuesday":
    console.log("It's Tuesday.");
    break;
  default:
    console.log("It's some other day.");
    break;
}

```

## 2. Loops:

- **for loop:** Repeats a block of code a specified number of times.

```

for (let i = 0; i < 5; i++) {
  console.log(i);
}

```

- **while loop:** Repeats a block of code while a specified condition is true.

```

let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}

```

- **do...while loop:** Repeats a block of code at least once, then continues while a specified condition is true.

```

let count = 0;
do {
  console.log(count);
  count++;
} while (count < 5);

```

- **forEach loop:** Iterates over elements of an array and executes a function for each element.

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number);
});
```

### 3. Control Flow Statements:

- **break:** Terminates the current loop or switch statement.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    break;
  }
  console.log(i);
}
```

- **continue:** Skips the current iteration of a loop and continues with the next iteration.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    continue;
  }
  console.log(i);
}
```

- **return:** Exits the current function and optionally returns a value.

```
function addNumbers(a, b) {
  if (typeof a !== "number" || typeof b !== "number") {
    return "Invalid input.";
  }
  return a + b;
}
```

These control flow statements allow you to conditionally execute blocks of code, repeat code execution based on conditions, and control the flow of your JavaScript program.

# Function in JavaScript

## **\*\*What is a Function?\*\***

A function in JavaScript is a block of code that performs a specific task or set of tasks. It can be thought of as a mini-program within your main program. Functions are defined with a name and can take inputs (parameters) and return a value.

## **\*\*Function Declaration\*\***

To declare a function, we use the `function` keyword followed by the function's name and parentheses, like this:

```
```javascript
function functionName(parameters) {
    // Function body (code to be executed)
    // ...
    return result; // Optional return statement
}
```
```

## **\*\*Function Invocation\*\***

Once a function is declared, we can use it by "calling" or "invoking" the function by its name followed by parentheses, like this:

```
```javascript
functionName(arguments);
```
```

## **\*\*Parameters and Arguments\*\***

Functions can accept parameters, which act as placeholders for values that the function will work with. When calling a function, we provide actual values called arguments that match the function's parameters, like this:

```
``javascript
function greet(name) {
  console.log("Hello, " + name + "!");
}

greet("Alice"); // Output: Hello, Alice!
greet("Bob");  // Output: Hello, Bob!
``
```

## **\*\*Return Statement\*\***

Functions can also return a value using the `return` keyword. This allows a function to produce an output that can be used elsewhere in the code.

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

const result = add(2, 3);
console.log(result); // Output: 5
``
```

## **\*\*Scope\*\***

Remember that variables declared inside a function have local scope, meaning they are only accessible within that function. Variables declared outside a function have global scope and can be accessed throughout the code.

### **\*\*Function Best Practices\*\***

1. Use meaningful names for functions to make the code more readable.
2. Keep functions small and focused on one task (single responsibility principle).
3. Comment your code to explain the purpose and behavior of the functions.
4. Test your functions with various inputs to ensure they work correctly.

**Declaring and Invoking Functions:** In JavaScript, you can declare a function using either a function declaration or a function expression. To invoke or call a function, you use its name followed by parentheses ().

#### **1. Function Declaration:**

- A function declaration starts with the **function** keyword, followed by the function name, a list of parameters (optional), and the function body enclosed in curly braces {}.
- **Function declarations are hoisted**, meaning they can be called before they are declared.

Example:

javascriptCopy code

```
function greet(name) { console.log("Hello, " + name + "!"); }  
greet("John"); // Output: Hello, John!
```

#### **2. Function Expression:**

- A function expression defines a function as part of a variable assignment.
- The function is assigned to a variable, which can be invoked like any other variable.

Example:

javascriptCopy code

```
let greet = function(name) {
```



```
console.log(name)
```

```
};
```

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

**Passing Parameters:** Functions in JavaScript can accept parameters, which are variables that hold the values passed to the function when it is called.

Example:

javascriptCopy code

```
function add(a, b) {
```

```
let sum = a + b; console.log(sum); }
```

```
add(2, 3); // Output: 5
```

**Returning Values:** Functions can also return values using the **return** statement. The returned value can be assigned to a variable or used directly.

Example:

javascriptCopy code

```
function multiply(a, b) {
```

```
return a * b;
```

```
}
```

```
let result = multiply(4, 5);
```

```
console.log(result); // Output: 20
```

## Types of Function in JavaScript

### 1. Named Function:

- A function with a specified name that can be called using that name.

javascriptCopy code

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

### 2. Anonymous Function:

- A function without a specified name. It is usually assigned to a variable or used as a callback function.

javascriptCopy code

```
let greet = function(name) { console.log("Hello, " + name + "!"); };
```

### 3. Arrow Function (ES6):

- A concise syntax for creating functions using the => arrow operator. It is often used for shorter function expressions.

javascriptCopy code

```
let multiply = (a, b) => a * b;
```

### 4. Immediately Invoked Function Expression (IIFE):

A function that is defined and invoked immediately. It helps create a private scope and avoid variable conflicts.

javascriptCopy code

```
(function() {
```

```
// Code executed immediately
```

```
Console.log("I am immediately invoked function Expression");
```

```
})();
```

```
result = (function(a, b){  
    return a - b;  
})(100, 42);
```

```
console.log(result); // 58
```

### 5. Generator Function:

A special type of function that can be paused and resumed, allowing the generation of a sequence of values over time.

```
function* numberGenerator() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
const generator = numberGenerator();  
console.log(generator.next().value); // Output: 1  
console.log(generator.next().value); // Output: 2  
console.log(generator.next().value); // Output: 3
```

## 6. Constructor Function:

A function used to create and initialize objects. It is typically invoked using the new keyword.

javascriptCopy code

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}  
  
let john = new Person("John", 25);
```

## 7. Callback Function:

- A function passed as an argument to another function and invoked within that function. It is commonly used for asynchronous operations or event handling.

javascriptCopy code

```
function fetchData(callback) { // Asynchronous operation // Once completed, invoke the  
  callback function callback(data); }
```

### 1. Synchronous Functions:

- Synchronous functions execute one operation at a time, blocking further execution until the current operation is completed.
- They follow the traditional sequential flow of code execution.
- Examples of synchronous functions include regular named functions, anonymous functions, and arrow functions without asynchronous operations.

Example:

javascriptCopy code

```
function add(a, b) {  
  return a + b; }  
  
let result = add(2, 3);  
  
console.log(result); // Output: 5
```

### 2. Asynchronous Functions:

- Asynchronous functions allow multiple operations to be executed concurrently without blocking the code execution flow.
- They are often used for time-consuming operations such as network requests, file I/O, or database queries.
- Asynchronous functions use techniques like callbacks, promises, or async/await to handle asynchronous operations and their results.

Example with a callback function:

```
function fetchData(callback) {  
  setTimeout(function() {  
    let data = "Some data";  
    callback(data);  
  }, 2000);  
}  
  
fetchData(function(data) {  
  console.log(data); // Output after 2 seconds: Some data  
});
```

In JavaScript, a Promise is an object used for handling asynchronous operations. It represents a value that may not be available yet but will be resolved in the future, either with a result or with an error. Promises provide a way to handle asynchronous code more elegantly and avoid callback hell.

- A Promise is a mechanism to handle asynchronous operations, and it is not a function.
- Asynchronous functions are functions that perform asynchronous tasks and can return Promises or use callbacks.

A Promise has three possible states:

1. Pending: The initial state. The Promise is still waiting for the asynchronous operation to complete.

2. Fulfilled: The asynchronous operation completed successfully, and the Promise has a resolved value.

3. Rejected: The asynchronous operation failed, and the Promise has a reason (error) for the failure.

Promises enable us to write more readable and maintainable asynchronous code by avoiding deeply nested callbacks and making error handling more straightforward.

Example with a Promise:

```
// Imagine we have an asynchronous function that simulates fetching data from a server with a delay.
function fetchDataFromServer() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = { name: "John", age: 30 };
      // For simplicity, let's assume the data was successfully fetched after a delay of 1 second.
      resolve(data); // Resolve the Promise with the fetched data.
      // If there was an error, you can reject the Promise with a reason like this:
      // reject(new Error("Failed to fetch data from the server."));
    }, 1000); // Simulate a 1-second delay for fetching data.
  });
}

// Now, let's use the fetchDataFromServer function to handle the Promise.
fetchDataFromServer()
  .then((data) => {
    // This block will execute when the Promise is fulfilled (resolved successfully).
    console.log("Fetched data:", data);
  })
  .catch((error) => {
    // This block will execute when the Promise is rejected (an error occurred).
    console.error("Error fetching data:", error);
  });
```

Example with async/await (ES8):

```

async function fetchData() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      let data = "Some data";
      resolve(data);
    }, 2000);
  });
}

async function process() {
  let data = await fetchData();
  console.log(data); // Output after 2 seconds: Some data
}

process();

```

Asynchronous functions are essential for handling time-consuming operations without blocking the entire program execution. They allow better responsiveness and enable handling complex operations, such as fetching data from a server, in a more efficient manner.

### Example of Callback function

Imagine you are hosting a dinner party at your home. You have invited your friends, Ram and Shyam, to join you. As a responsible host, you want to make sure that everyone has a good time and enjoys the delicious food.

In JavaScript terms:

#### 1. Setting the Dinner Table (Function Definition):

- Before the party starts, you prepare the dining table by setting the plates, cutlery, and glasses. You define a function called **prepareTable** to handle this task.
- The **prepareTable** function takes a callback function called **onTableReady** as a parameter. This callback function will be invoked once the table is prepared.

#### 2. Preparing the Dinner Table (Function Execution):

- You start executing the **prepareTable** function by passing a callback function named **onTableReady** to it.

- As you start setting the table, you also assign unique place cards to each friend, Ram and Shyam, indicating their seating arrangement.
- Once the table is fully prepared, you call the **onTableReady** callback function to inform your friends that the table is ready.

### 3. Enjoying the Dinner (Callback Function Execution):

- Ram and Shyam receive the callback message that the table is ready.
- They arrive at your home and take their seats at the table according to their assigned place cards.
- Now, the dinner begins, and everyone enjoys the delicious food and engages in conversations.

```
function prepareTable(onTableReady) {  
  console.log("Setting the dinner table...");  
  console.log("Assigning place cards to guests...");  
  
  // Simulating table preparation time  
  setTimeout(function() {  
    console.log("Table is ready!");  
  
    // Invoke the callback function to notify the guests  
    onTableReady();  
  }, 3000); // Simulate a 3-second delay  
}  
  
function onTableReadyCallback() {  
  console.log("Table is ready! Guests can take their seats.");  
  console.log("Let's start the dinner!");  
}  
  
// Start preparing the table and pass the onTableReadyCallback as the call  
prepareTable(onTableReadyCallback);
```

In this story:

- The **prepareTable** function represents the task of setting the dinner table. It takes the **onTableReady** callback function as a parameter.

- Inside the **prepareTable** function, you simulate the time it takes to prepare the table using **setTimeout**. After the 3-second delay, you log a message indicating that the table is ready and then invoke the **onTableReady** callback function.
- The **onTableReadyCallback** function represents the response or action taken when the table is ready. In this case, it logs a message indicating that the table is ready, and the guests can take their seats.
- Finally, you start preparing the table by calling the **prepareTable** function and passing the **onTableReadyCallback** function as the callback. Once the table is ready, the **onTableReadyCallback** function is executed.

So, in this story, the callback function **onTableReadyCallback** acts as an event handler that gets triggered once the table is ready. It allows you to perform specific actions or respond to an event (in this case, the table being ready) without blocking the execution of other tasks.

```
function getUserData(userId, successCallback, errorCallback) {
  // Simulating an asynchronous operation (e.g., fetching user data from a server)
  setTimeout(function() {
    let user = null; // Assume user data is not found
    if (userId === 123) {
      user = {
        id: 123,
        name: "John Doe",
        email: "john@example.com"
      };
    }
    if (user) {
      successCallback(user); // Invoke the success callback with the user data
    } else {
      errorCallback("User not found"); // Invoke the error callback with an error message
    }
  }, 2000);
}

function displayUserData(userData) {
  console.log("User ID: " + userData.id);
  console.log("Name: " + userData.name);
  console.log("Email: " + userData.email);
}

function displayError(error) {
  console.log("Error: " + error);
}

// Usage: Pass the displayUserData and displayError functions as callbacks to getUserData
getUserData(123, displayUserData, displayError);
```

In this example, we have a `getUserData` function that simulates an asynchronous operation of fetching user data from a server based on a given `userId`. It uses `setTimeout` to introduce a delay of 2000 milliseconds (2 seconds) to simulate the asynchronous behavior.

The `getUserData` function takes three parameters: `userId`, `successCallback`, and `errorCallback`. After the simulated asynchronous operation is completed, it checks if the user data is found based on the `userId`. If the user data is found, it invokes the `successCallback` function and passes the user data as an argument. If the user data is not found, it invokes the `errorCallback` function and passes an error message.



We also have a `displayUserData` function that serves as the success callback. It takes the user data as an argument and displays the user's ID, name, and email.

Additionally, we have a `displayError` function that serves as the error callback. It takes an error message as an argument and displays the error.

Finally, we invoke the `getUserData` function and pass `displayUserData` as the success callback and `displayError` as the error callback. Depending on whether the user data is found or not, the corresponding callback function is invoked. In this case, if the user data with `userId 123` is found, it will be displayed using the `displayUserData` function. Otherwise, an error message will be displayed using the `displayError` function.

This example showcases the usage of callback functions to handle success and error scenarios in an asynchronous operation, allowing you to perform different actions based on the outcome of the operation.

### **Example of Promise**

Imagine you are a student waiting for your friend, Ram, to send you an important document via email. However, RAM is notorious for being late with her tasks, and you are getting impatient. You want to know when the document will arrive so that you can plan your study schedule accordingly.

In JavaScript terms:

#### **1. Without Promises (Traditional Approach):**

- You decide to check your email every 5 minutes to see if Ram has sent the document.
- You keep manually refreshing your inbox repeatedly, wasting your time and effort.
- You have no idea when Ram will actually send the document, and you cannot do anything else productive until it arrives.

This is similar to the synchronous programming approach, where you keep checking for updates (polling) and block further execution until the desired result is obtained.

#### **2. With Promises (Modern Approach):**

- Instead of continuously checking your email, you ask Ram to promise that He will send you the document within a specific timeframe.
- Ram agrees and gives you a promise that he will send the document within the next hour.
- You can continue with your other tasks, knowing that you will receive the document as soon as Ram fulfils her promise.

- Once Ram sends the document, you receive a notification, and you can access and use the document for your studies.

This is similar to the asynchronous programming approach using Promises, where you make a request and receive a Promise in return. The Promise represents the eventual completion (fulfilment or rejection) of an asynchronous operation.

```
const promise = new Promise((resolve, reject) => {
  // Simulating an asynchronous operation (e.g., fetching data from a server)
  setTimeout(function() {
    const document = "Important document";
    const error = false; // Simulate no error

    if (error) {
      reject("Error: Failed to send the document");
    } else {
      resolve(document);
    }
  }, 5000); // Simulate a 5-second delay
});

promise.then((document) => {
  console.log("Received document:", document);
}).catch((error) => {
  console.log(error);
});
```

In this example:

- You create a new Promise using the Promise constructor, which takes a callback function as an argument. The callback function receives two parameters: resolve and reject.
- Inside the callback function, you simulate an asynchronous operation using setTimeout. After the 5-second delay, it either fulfills the Promise by invoking resolve with the document or rejects the Promise by invoking reject with an error message.
- You attach a then method to the Promise, which handles the fulfillment case. It receives the document as an argument and logs it to the console.
- You attach a catch method to the Promise, which handles the rejection case. It receives the error message as an argument and logs the error.

So, in this story, the Promise represents Ram's commitment to sending the document. You can continue with your tasks until the Promise is fulfilled. Once the document arrives (Promise is fulfilled), the then method is executed, and you can access and use the document. If an error occurs (Promise is rejected), the catch method is executed, and you can handle the error gracefully.

Promises allow you to work with asynchronous operations in a more organized and efficient manner by providing a clear way to handle success and failure scenarios.

## **Async and await**

Imagine you are a detective solving a mysterious crime. You have a list of suspects to investigate, and you need to gather evidence from various sources, such as interviewing witnesses, checking surveillance footage, and analyzing forensic reports. However, each source of evidence takes time to gather and process.

In JavaScript terms:

### **1. Traditional Investigation Approach (Callback Hell):**

- In the traditional approach, you start investigating the first source of evidence and provide a callback function to handle the result.
- Once the result is obtained, you move on to the next source of evidence, but this time you nest another callback within the previous callback.
- This nesting of callbacks continues for each piece of evidence, creating a tangled web of callbacks, often referred to as "Callback Hell."
- As a detective, it becomes challenging to manage and follow the flow of the investigation due to the nested structure.

### **2. Modern Investigation Approach (async/await):**

- In the modern approach, you declare your investigation function as `async`, allowing you to use the `await` keyword within it.
- Each source of evidence is written as a separate function that returns a promise, representing the asynchronous operation.
- Within your investigation function, you can use `await` to pause the execution until the promise is resolved or rejected, making your code look more linear and readable.
- The `await` keyword allows you to await the completion of one task before moving on to the next, simplifying the flow of the investigation.

```
function interviewWitness() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Interviewing witness...");
      const statement = "The suspect was tall and had a beard.";
      resolve(statement);
    }, 2000); // Simulate 2 seconds of interviewing
  });
}

function analyzeSurveillanceFootage() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Analyzing surveillance footage...");
      const footage = "The suspect entered the building at 9:00 PM.";
      resolve(footage);
    }, 1500); // Simulate 1.5 seconds of analysis
  });
}

function processForensicReports() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Processing forensic reports...");
      const report = "Fingerprints found at the crime scene match the suspect.";
      resolve(report);
    }, 3000); // Simulate 3 seconds of processing
  });
}

async function solveCrime() {
  try {
    console.log("Starting the investigation...");

    const witnessStatement = await interviewWitness();
```

```

async function solveCrime() {
  try {
    console.log("Starting the investigation...");

    const witnessStatement = await interviewWitness();
    console.log("Witness statement:", witnessStatement);

    const surveillanceFootage = await analyzeSurveillanceFootage();
    console.log("Surveillance footage:", surveillanceFootage);

    const forensicReport = await processForensicReports();
    console.log("Forensic report:", forensicReport);

    console.log("Crime solved successfully!");
  } catch (error) {
    console.log("Error occurred during investigation:", error);
  }
}

// Start the investigation
solveCrime();

```

You, 1 second ago • Uncommitted changes

In this story:

- You have separate functions, `interviewWitness`, `analyzeSurveillanceFootage`, and `processForensicReports`, representing different sources of evidence. Each function returns a Promise, simulating an asynchronous operation.
- The `solveCrime` function is declared as `async` to enable the usage of the `await` keyword. It represents the main investigation function.
- Inside the `solveCrime` function, you await the completion of each source of evidence by calling the respective functions with `await`.
- Each piece of evidence is logged to the console once it becomes available.
- The investigation progresses linearly, and the code looks cleaner and more organized compared to the callback approach.
- If an error occurs during any step of the investigation, it is caught in the `catch` block and handled appropriately.

So, in this story, the `async` and `await` functions allow you to write asynchronous code in a more synchronous manner, making it easier to read, understand, and manage the flow of the investigation.

**Certainly! Let's dive into arrays and objects in more detail:**

## **1. Arrays:**

An array is a data structure that allows you to store a collection of elements of the same data type in a sequential order. Each element in the array is identified by an index, which represents its position in the array. The index starts from 0 for the first element, 1 for the second element, and so on.

Arrays are commonly used to group related data together and are very useful when you need to work with multiple values of the same type.

Features of arrays:

- Homogeneous data: All elements in an array must be of the same data type. For example, an array of integers cannot contain strings.
- Fixed-size or dynamic: Some programming languages require you to specify the size of the array when creating it, and this size remains fixed throughout the program's execution. Other languages allow dynamic resizing, meaning you can add or remove elements as needed.
- Random access: Since elements in an array are indexed, you can directly access any element by using its index, making it efficient to retrieve values.

Example in JavaScript:

```
````javascript
# Creating an array of integers
const numbers = [1, 2, 3, 4, 5]

# Accessing elements in the array
Console.log(numbers[0]) # Output: 1
Console.log(numbers[2]) # Output: 3
...````
```

## 2. Objects:

An object is a data structure that allows you to store data in the form of key-value pairs. Each value in the object is associated with a unique key, which acts as an identifier for that value. Unlike arrays, the order of elements in an object is not guaranteed, as they are accessed using their keys rather than indices.

Objects are particularly useful when you want to represent real-world entities with different properties or attributes. It allows you to organize related information in a structured manner.

Features of objects:

- Heterogeneous data: Objects can store values of different data types as their properties. For example, an object representing a person can have a name (string), age (integer), and hobbies (an array of strings).
- Dynamic properties: You can add or remove properties from an object at runtime, making it a flexible data structure.
- Key-based access: To access the value of a property in an object, you use the associated key, which allows for easy retrieval of information.

Example in JavaScript:

```
``javascript
// Creating an object to represent a person
const person = {
  name: "John",
  age: 30,
  hobbies: ["Reading", "Cooking", "Running"],
};

// Accessing values in the object
```

```
console.log(person.name); // Output: "John"
console.log(person.age); // Output: 30
console.log(person.hobbies); // Output: ["Reading", "Cooking", "Running"]
...

```

Arrays and objects are both powerful data structures with different use cases. Arrays are suitable for collections of similar data, while objects are ideal for representing entities with various properties. Understanding how to use arrays and objects effectively will significantly enhance your ability to work with complex data in programming.

### **Array Methods:**

Arrays in JavaScript come with various built-in methods that allow you to manipulate and perform operations on array elements.

Example: Using array methods

javascript

Copy code

```
const numbers = [3, 1, 5, 2, 4];
```

```
numbers.push(6); // Add an element at the end
console.log(numbers); // Output: [3, 1, 5, 2, 4, 6]
```

```
numbers.pop(); // Remove the last element
console.log(numbers); // Output: [3, 1, 5, 2, 4]
```

```
numbers.sort(); // Sort the array
console.log(numbers); // Output: [1, 2, 3, 4, 5]
```

In this example, we used the `push()` method to add an element at the end of the array, the `pop()` method to remove the last element, and the `sort()` method to sort the array in ascending order.



## Object Methods:

Objects in JavaScript can also have methods, which are functions associated with the object.

Example: Using object methods

```
const rectangle = {  
  width: 10,  
  height: 5,  
  getArea: function() {  
    return this.width * this.height;  
  }  
};
```

```
console.log(rectangle.getArea()); // Output: 50
```

In this example, we defined an object rectangle with properties width and height, as well as a method getArea() that calculates the area of the rectangle.

Arrays and objects are fundamental data structures in JavaScript that are extensively used for organizing and manipulating data. Understanding their concepts and utilizing their methods is essential for effective JavaScript programming.

**Both arrays and objects in JavaScript can use the `find`, `map`, and `filter` methods, but they are used slightly differently for each data structure.**

### **\*\*Arrays:\*\***

1. `find`: Finds the first element in the array that satisfies the provided testing function. It returns the value of the first element found, or `undefined` if no element satisfies the condition.
2. `map`: Creates a new array with the results of calling a provided function on every element in the array. It returns a new array with the transformed elements.

3. `filter`: Creates a new array with all elements that pass the provided testing function. It returns a new array containing only the elements that satisfy the condition.

**\*\*Example:\*\***

```
```javascript
```

```
const numbers = [10, 20, 30, 40, 50];
```

```
const foundNumber = numbers.find(num => num > 30);
```

```
console.log(foundNumber); // Output: 40
```

```
const filteredNumbers = numbers.filter(num => num > 30);
```

```
console.log(filteredNumbers); // Output: [40, 50]
```

```
const squaredNumbers = numbers.map(num => num * num);
```

```
console.log(squaredNumbers); // Output: [100, 400, 900, 1600, 2500]
```

**\*\*Objects:\*\***

1. `find`: The `find` method is not directly available for objects in JavaScript. Since objects do not have indexes like arrays, you can use `Object.keys` or `Object.values` to get an array and then use the `find` method on it. Alternatively, you can use `Object.entries` to get key-value pairs as an array and then apply the `find` method.

**\*\*Example:\*\***

```
```javascript
```

```
const person = {
```

```
  name: 'John Doe',
```

```
  age: 30,
```

```
  occupation: 'Developer'
```

```
};
```

```
// Example using Object.keys()
const key = Object.keys(person).find(key => key === 'age');
console.log(person[key]); // Output: 30

// Example using Object.entries()
const [property, value] = Object.entries(person).find(([key, value]) => key === 'occupation');
console.log(value); // Output: 'Developer'
```

```

2. `map`: The `map` method is not directly available for objects. To achieve similar behavior, you can use `Object.entries` to get an array of key-value pairs, apply the `map` method, and then convert it back to an object using `Object.fromEntries`.

**Example:**

```
```javascript
const person = {
  name: 'John Doe',
  age: 30,
  occupation: 'Developer'
};

const updatedPerson = Object.fromEntries(
  Object.entries(person).map(([key, value]) =>
    key === 'name' ? [key, value.toUpperCase()] : [key, value]
  )
);

console.log(updatedPerson);
// Output: { name: 'JOHN DOE', age: 30, occupation: 'Developer' }
```

```

3. `filter`: The `filter` method is also not directly available for objects. You can use `Object.entries` to get an array of key-value pairs, apply the `filter` method, and then convert it back to an object using `Object.fromEntries`.

**Example:**

```
````javascript
const person = {
  name: 'John Doe',
  age: 30,
  occupation: 'Developer'
};

const filteredPerson = Object.fromEntries(
  Object.entries(person).filter(([key, value]) => key !== 'age')
);

console.log(filteredPerson);
// Output: { name: 'John Doe', occupation: 'Developer' }
````
```

In summary, arrays have direct access to `find`, `map`, and `filter` methods, while objects require additional steps to apply these methods. By using `Object.entries`, you can convert objects into arrays and utilize the array methods, and then convert the resulting arrays back to objects using `Object.fromEntries`.

**// Create an array of objects**

```
const arrayOfObjects = [  
  { name: 'Alice', age: 25 },  
  { name: 'Bob', age: 30 },  
  { name: 'Charlie', age: 28 }  
];
```

**// Use the map function to fetch values from objects**

```
const names = arrayOfObjects.map(obj => obj.name);  
const ages = arrayOfObjects.map(obj => obj.age);
```

```
console.log(names); // Output: ['Alice', 'Bob', 'Charlie']
```

```
console.log(ages); // Output: [25, 30, 28]
```

## Classes in JavaScript

In JavaScript, classes provide a way to create objects with shared properties and methods. They serve as blueprints for creating instances of objects, making it easier to organize and manage code.

### **\*\*Step 1: Declaring a Class\*\***

To declare a class, you use the `'class'` keyword followed by the class name. Inside the class, you define its properties and methods.

The constructor in a class is required because it serves as a special method responsible for initializing the object when it is created. It allows you to set the initial state of the object and define its properties. The constructor method is automatically called when you use the **new** keyword to create an instance of the class.

Here are the key reasons why constructors are required:

1. **Object Initialization:** When you create an object from a class, it often needs to be set up with specific initial values for its properties. The constructor method allows you to specify these initial values and prepare the object for use.
2. **Property Assignment:** In the constructor, you can assign values to the properties of the object using the **this** keyword, which refers to the instance being created. This helps ensure that each instance has its own unique set of property values.

3. State Management: Objects often need to maintain certain states or internal data. The constructor allows you to establish the initial state of the object and ensure it starts in a consistent and well-defined state.
4. Automatic Invocation: The constructor is automatically called when a new instance of the class is created with the **new** keyword. This ensures that the necessary setup code is executed without requiring explicit calls.

```
``javascript
// Declaration of a simple class called "Person"
class Person {
  // Constructor method to initialize the object
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  // Method to greet the person
  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
``
```

### **\*\*Step 2: Creating an Instance\*\***

Once the class is defined, you can create instances of the class using the `new` keyword. The `new` keyword calls the class constructor to create a new object.

```
``javascript
// Creating instances of the Person class
const ram = new Person("Ram", 25);
const rahul = new Person("Rahul", 30);
```

...

### **\*\*Step 3: Accessing Properties and Methods\*\***

You can access the properties and methods of an instance using the dot notation (`instance.property`` or `instance.method()``).

```
``javascript
```

```
// Accessing properties and calling methods
```

```
console.log(person1.name); // Output: "Alice"
```

```
console.log(person2.age); // Output: 30
```

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

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

...

### **\*\*Step 4: Inheritance\*\***

JavaScript supports class inheritance, allowing you to create a new class that inherits properties and methods from an existing class.

```
``javascript
```

```
// Creating a subclass "Student" that inherits from "Person"
```

```
class Student extends Person {
```

```
  constructor(name, age, major) {
```

```
    // Calling the parent class constructor using "super"
```

```
    super(name, age);
```

```
    this.major = major;
```

```
  }
```

```
// Overriding the greet method of the parent class
```

```
  greet() {
```

```
    console.log(`Hello, my name is ${this.name}, I am ${this.age} years old, and I'm majoring
in ${this.major}.`);
  }
}
...`
```

### **\*\*Step 5: Using the Subclass\*\***

Now, you can create instances of the subclass and use its methods, including the overridden methods.

```
```javascript
// Creating instances of the Student subclass
const student1 = new Student("Eve", 22, "Computer Science");
const student2 = new Student("Jack", 24, "Biology");

// Accessing properties and calling methods of the Student class
console.log(student1.name); // Output: "Eve"
console.log(student2.age); // Output: 24

student1.greet(); // Output: "Hello, my name is Eve, I am 22 years old, and I'm majoring in
Computer Science."
student2.greet(); // Output: "Hello, my name is Jack, I am 24 years old, and I'm majoring in
Biology."
...`
```

With classes and inheritance, you can create more complex and organized code structures in JavaScript, making it easier to manage and extend your applications.

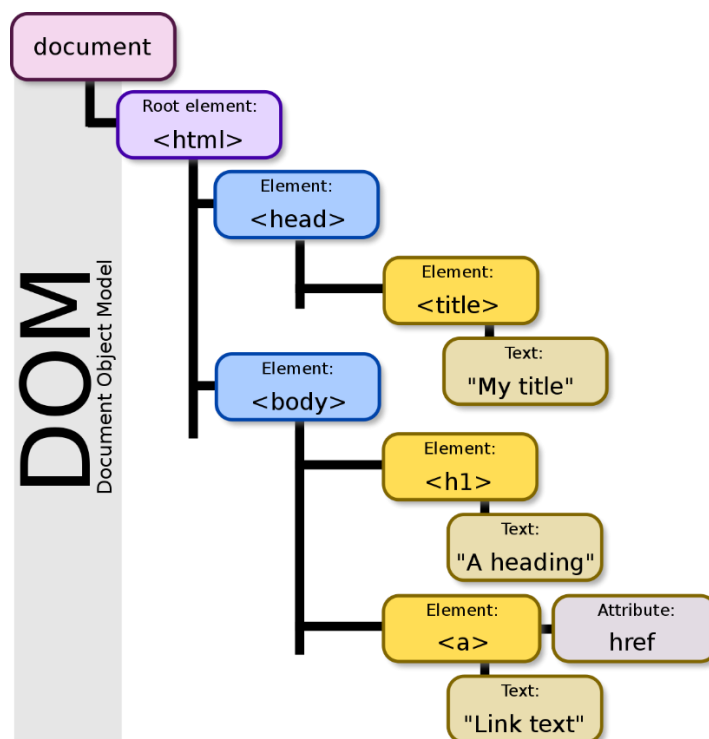


## DOM Manipulation in JavaScript

DOM (Document Object Model) manipulation refers to the process of accessing, modifying, or updating elements and content within an HTML document using JavaScript. It allows you to dynamically modify the structure, style, and content of a web page.

The Document Object Model (DOM) represents the HTML document as a tree-like structure, where each element, attribute, and text node are represented as an object. Through DOM manipulation, you can access these objects, traverse the DOM tree, and perform various operations on the elements.

- **Document:** Refers to the web page or XML document being displayed in the browser.
- **Object:** Represents each element or entity in the document, such as HTML tags, attributes, text, etc.
- **Model:** Describes how these objects are organized and their relationships with each other, forming a tree-like structure.



DOM provides several methods to select elements:

- **getElementById:** Selects an element with a specific ID.
- **getElementsByClassName:** Selects elements with a specific class name.
- **getElementsByTagName:** Selects elements with a specific tag name.

- **querySelector**: Selects the first element that matches a specific CSS selector.
- **querySelectorAll**: Selects all elements that match a specific CSS selector.

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Manipulation Example</title>
</head>
<body>
  <h1 id="title">Hello, World!</h1>
  <p class="content">This is a paragraph.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
  <button id="btn">Click Me</button>

  <script src="script.js"></script>
</body>
</html>
```

Accessing Elements:

By ID: Use `getElementById` to retrieve an element with a specific ID.

```
const titleElement = document.getElementById('title');
console.log(titleElement.textContent); // Output: Hello, World!
```

By Class Name: Use `getElementsByClassName` to retrieve elements with a specific class name (returns a collection).

```
const contentElements = document.getElementsByClassName('content');
console.log(contentElements[0].textContent); // Output: This is a paragraph.
```

By Tag Name: Use `getElementsByTagName` to retrieve elements with a specific HTML tag name (returns a collection).

```
const listItemElements = document.getElementsByTagName('li');
console.log(listItemElements.length); // Output: 3
```

Modifying Elements:

Changing Text Content: Use the `textContent` property to update the text content of an element.

```
titleElement.textContent = 'Hello, OpenAI!';
```

Changing HTML Content: Use the `innerHTML` property to update the HTML content of an element.

```
const paragraphElement = document.querySelector('.content');
paragraphElement.innerHTML = 'This is a <strong>modified</strong> paragraph';
```

## Modifying Attributes:

In JavaScript, attributes are properties of HTML elements that provide additional information or modify the behavior of the element. Attributes can be specified directly in the HTML markup or manipulated dynamically using JavaScript.

When an HTML document is loaded into the browser, it is represented as a Document Object Model (DOM) tree. The DOM is a programming interface for web documents and provides a structured representation of the HTML elements and their attributes. In the DOM, attributes are exposed as properties of the corresponding HTML elements.

For example, consider an HTML element like this:

```
<input type="text" id="myInput" value="Hello, world!" />
```

In this case, the `type`, `id`, and `value` are attributes of the `<input>` element. You can access and manipulate these attributes using JavaScript. Here's an example:

```
// Accessing attributes using DOM properties
var inputElement = document.getElementById("myInput");
console.log(inputElement.type); // Output: "text"
```

```
console.log(inputElement.value); // Output: "Hello, world!"
```

```
// Modifying attributes using DOM properties
```

```
inputElement.value = "New value";
```

```
console.log(inputElement.value); // Output: "New value"
```

As shown in the example, you can access an attribute by accessing the corresponding property of the DOM element. Similarly, you can modify the attribute by assigning a new value to the property.

In addition to accessing attributes directly through DOM properties, you can also use methods like `getAttribute()` and `setAttribute()` to get and set attribute values dynamically. Here's an example:

```
// Using getAttribute() and setAttribute() methods
```

```
var inputElement = document.getElementById("myInput");
```

```
console.log(inputElement.getAttribute("value")); // Output: "Hello, world!"
```

```
inputElement.setAttribute("value", "New value");
```

```
console.log(inputElement.getAttribute("value")); // Output: "New value"
```

These methods provide an alternative way to interact with attributes if you prefer working with strings instead of DOM properties.

Certainly! In JavaScript, you can remove attributes from HTML elements using the `removeAttribute()` method, and you can check if an element has a specific attribute using the `hasAttribute()` method.

Removing an Attribute:

To remove an attribute from an HTML element, you can use the `removeAttribute()` method. Here's an example:

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

```
element.removeAttribute("attributeName");
```

In the above code, `myElement` is the ID of the HTML element, and `"attributeName"` is the name of the attribute you want to remove.

### Checking if an Attribute Exists:

To check if an element has a particular attribute, you can use the `hasAttribute()` method. It returns a Boolean value (true or false) indicating whether the attribute exists on the element. Here's an example:

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

if (element.hasAttribute("attributeName")) {

    // Attribute exists

} else {

    // Attribute does not exist

}
```

In the above code, `myElement` is the ID of the HTML element, and `"attributeName"` is the name of the attribute you want to check.

It's important to note that when you remove an attribute using `removeAttribute()`, it permanently deletes the attribute from the element. However, if you want to temporarily disable an attribute without removing it, you can set its value to null or an empty string (""), like this:

### Javascript Copy code

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

element.setAttribute("attributeName", null); // or element.setAttribute("attributeName", "");
```

This will effectively disable the attribute without removing it completely.

### Creating and Appending Elements:

Creating Elements: Use the `createElement` method to create a new element.

```
const newElement = document.createElement('div');
newElement.textContent = 'Newly created element';
```

Appending Elements: Use the `appendChild` method to add the newly created element to an existing element.

```
const bodyElement = document.body;
bodyElement.appendChild(newElement);
```

## JavaScript Sibling

In this tutorial, you will learn how to select the next siblings, previous siblings, and all siblings of an element.

Let's say you have the following list of items:

```
<ul id="menu">
  <li>Home</li>
  <li>Products</li>
  <li class="current">Customer Support</li>
  <li>Careers</li>
  <li>Investors</li>
  <li>News</li>
  <li>About Us</li>
</ul>
```

Code language: HTML, XML (xml)

## Get the next siblings

To get the next sibling of an element, you use the `nextElementSibling` attribute:

```
let nextSibling = currentNode.nextElementSibling;
```

Code language: JavaScript (javascript)

The `nextElementSibling` returns null if the specified element is the last one in the list.

The following example uses the `nextElementSibling` property to get the next sibling of the list item that has the `current` class:

```
let current = document.querySelector('.current');
let nextSibling = current.nextElementSibling;

console.log(nextSibling);
```

Code language: JavaScript (javascript)

Output:

```
<li>Careers</li>
```

Code language: HTML, XML (xml)

In this example:

- First, select the list item whose class is `current` using the [querySelector\(\)](#).

- Second, get the next sibling of that list item using the `nextElementSibling` property.

To get all the next siblings of an element, you can use the following code:

```
let current = document.querySelector('.current');
let nextSibling = current.nextElementSibling;

while(nextSibling) {
  console.log(nextSibling);
  nextSibling = nextSibling.nextElementSibling;
}
```

Code language: JavaScript (javascript)

## Get the previous siblings

To get the previous siblings of an element, you use the `previousElementSibling` attribute:

```
let current = document.querySelector('.current');
let prevSibling = current.previousElementSibling;
```

Code language: JavaScript (javascript)

The `previousElementSibling` property returns `null` if the current element is the first one in the list.

The following example uses the `previousElementSibling` property to get the previous siblings of the list item that has the `current` class:

```
let current = document.querySelector('.current');
let prevSiblings = current.previousElementSibling;

console.log(prevSiblings);
```

Code language: JavaScript (javascript)

And the following example selects all the previous siblings of the list item that has the `current` class:

```
let current = document.querySelector('.current');
let prevSibling = current.previousElementSibling;
while(prevSibling) {
  console.log(prevSibling);
  prevSibling = current.previousElementSibling;
}
```

Code language: JavaScript (javascript)

## Get all siblings of an element

To get all siblings of an element, we'll use the logic:

- First, select the parent of the element whose siblings you want to find.
- Second, select the first child element of that parent element.
- Third, add the first element to an array of siblings.

- Fourth, select the next sibling of the first element.
- Finally, repeat the 3rd and 4th steps until there are no siblings left. In case the sibling is the original element, skip the 3rd step.

The following function illustrates the steps:

```
let getSiblings = function (e) {
  // for collecting siblings
  let siblings = [];
  // if no parent, return no sibling
  if(!e.parentNode) {
    return siblings;
  }
  // first child of the parent node
  let sibling = e.parentNode.firstChild;
  // collecting siblings
  while (sibling) {
    if (sibling.nodeType === 1 && sibling !== e) {
      siblings.push(sibling);
    }
    sibling = sibling.nextSibling;
  }
  return siblings;
};
```

Code language: JavaScript (javascript)

Put it all together:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>JavaScript Siblings</title>
</head>
<body>
  <ul id="menu">
    <li>Home</li>
    <li>Products</li>
    <li class="current">Customer Support</li>
    <li>Careers</li>
    <li>Investors</li>
    <li>News</li>
    <li>About Us</li>
  </ul>
  <script>
    let getSiblings = function (e) {
      // for collecting siblings
      let siblings = [];
      // if no parent, return no sibling
```



```

        if(!e.parentNode) {
            return siblings;
        }
        // first child of the parent node
        let sibling = e.parentNode.firstChild;
        // collecting siblings
        while (sibling) {
            if (sibling.nodeType === 1 && sibling !== e) {
                siblings.push(sibling);
            }
            sibling = sibling.nextSibling;
        }
        return siblings;
    };

```

```

        let siblings =
getSiblings(document.querySelector('.current'));
        siblingText = siblings.map(e => e.innerHTML);
        console.log(siblingText);
    </script>
</body>
</html>

```

Code language: HTML, XML (xml)

Output:

```

["Home", "Products", "Careers", "Investors", "News", "About Us"]

```

Code language: JSON / JSON with Comments (json)

## Summary

- The `nextElementSibling` returns the next sibling of an element or `null` if the element is the last one in the list.
- The `previousElementSibling` returns the previous sibling of an element or `null` if the element is the first one in the list.
- To get all siblings of an element, you can use a helper function that utilizes the `nextElementSibling` property.

## Element Style

### Manipulating Element's Styles

- [style property](#) – get or set inline styles of an element.
- [getComputedStyle\(\)](#) – return the computed style of an element.
- [className property](#) – return a list of space-separated CSS classes.
- [classList property](#) – manipulate CSS classes of an element.
- [Element's width & height](#) – get the width and height of an element.

### Event Handling:

Attaching Event Listeners: Use the `addEventListener` method to attach an event listener to an element.

onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<h2>JavaScript HTML Events</h2>
```

```
<p>Click the button to display the date.</p>
```

```
<button onclick="displayDate()">The time is?</button>

<script>

function displayDate() {

    document.getElementById("demo").innerHTML = Date();

}

</script>

<p id="demo"></p>

</body>

</html>
```

```
buttonElement.addEventListener('click', function() {
    console.log('Button clicked!');
});
```

Modifying Styles:

Changing CSS Properties: Use the style property to modify CSS properties of an element.

```
paragraphElement.style.color = 'red';
paragraphElement.style.fontSize = '20px';
```


DOM manipulation allows you to dynamically interact with the structure and content of a web page, enabling you to create interactive and responsive user experiences. These examples demonstrate some of the fundamental operations for DOM manipulation in JavaScript, but there are many more methods and techniques available to manipulate the DOM based on your specific needs.

Here is some more concept related to DOM manipulation:

### 1. Removing Elements:

- Use the `removeChild` method to remove a specific child element from its parent.


javascript

 Copy code

```
const parentElement = document.getElementById('parent');
const childElement = document.getElementById('child');
parentElement.removeChild(childElement);
```

- Use the `remove` method to remove an element directly.

javascript


 Copy code

```
const elementToRemove = document.getElementById('element');
elementToRemove.remove();
```

### 2. Modifying Classes:

- Adding and Removing Classes: Use the `classList` property to manipulate CSS classes of an element.


javascript

 Copy code

```
const element = document.getElementById('element');
element.classList.add('new-class');
element.classList.remove('old-class');
```

- **Toggling Classes:** Use the `classList.toggle` method to add or remove a class based on its presence.

javascript


 Copy code

```
const element = document.getElementById('element');
element.classList.toggle('active');
```

### 3. Traversing the DOM:

- **Parent Element:** Use the `parentNode` property to access the parent element of a given element.


javascript

 Copy code

```
const childElement = document.getElementById('child');
const parentElement = childElement.parentNode;
```

- **Child Elements:** Use the `childNodes` property to access the child nodes of an element (including text nodes).


javascript

 Copy code

```
const parentElement = document.getElementById('parent');
const childNodes = parentElement.childNodes;
```

- **Query Selector:** Use the `querySelector` method to find the first matching element within an element or the document.


javascript

 Copy code

```
const element = document.querySelector('.class');
```

- **Query Selector All:** Use the `querySelectorAll` method to find all matching elements within an element or the document.

javascript

 Copy code

```
const elements = document.querySelectorAll('.class');
```

