## JavaScript

JavaScript is a **high-level, interpreted programming language** primarily used to create interactive and dynamic content on web pages. It is one of the core technologies of the World Wide Web, alongside HTML and CSS.

### Features of Javascript:

1. **Lightweight and interpreted**: JavaScript runs directly in the browser without needing compilation, making it fast and efficient.
2. **Object-oriented**: It uses objects (with properties and methods) to organize and manipulate data.
3. **Cross-platform**: JavaScript works on all devices and operating systems that support web browsers.
4. **Dynamically typed**: You don't need to declare variable types; they are determined at runtime.
5. **Event-driven and asynchronous capabilities**: It reacts to user actions (like clicks) and handles tasks without blocking the rest of the code.

JavaScript is used for tasks like form validation, creating animations, handling user events, and more.

```
<!-- Example: Simple HTML with JavaScript -->
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
</head>
<body>
  <h1>Welcome to JavaScript!</h1>
  <script>
    console.log("Hello, World!");
  </script>
</body>
</html>
```

## History and Evolution of JavaScript

- **1995**: JavaScript was created by **Brendan Eich** at Netscape in just 10 days. It was initially called **Mocha**, then renamed **LiveScript**, and finally **JavaScript**.
- **1997**: JavaScript was standardized as **ECMAScript (ES)** by ECMA International.
- **2009**: Node.js was introduced, allowing JavaScript to run on servers.
- **2015**: ES6 (ECMAScript 2015) was released, introducing modern features like let, const, arrow functions, and classes.
- **Present**: JavaScript is continuously evolving with new versions like ES2023, making it more powerful and versatile.

## Role of JavaScript in Web Development

JavaScript is essential for making web pages **interactive** and **user-friendly**. It works alongside:

- **HTML**: Defines the structure of the web page.
- **CSS**: Styles the web page.
- **JavaScript**: Adds behavior to the web page.

### Examples of JavaScript's Role:

1. **Form Validation**: Ensures user inputs are correct before submission.
2. **Dynamic Content**: Updates the page content without reloading (e.g., using AJAX).
3. **Interactive Elements**: Enables dropdown menus, sliders, and modals.
4. **Animations**: Adds motion to elements (e.g., fading effects, image carousels).

```
// Example: Changing the text of a heading dynamically
document.getElementById("heading").innerText = "JavaScript Changed This Text!";
```

## Differences Between JavaScript, Java, and Python

JavaScript is used for **front-end interactivity**, Java for **backend enterprise applications**, and Python for **data science and automation**.

| **Feature** | **JavaScript** | **Java** | **Python** |
|---|---|---|---|
| **Type** | Interpreted, lightweight | Compiled, statically typed | Interpreted, dynamically typed |
| **Primary Use** | Web development | Enterprise applications | General-purpose programming |
| **Syntax** | Flexible and dynamic | Strict and verbose | Simple and easy-to-read |
| **Execution** | Runs in browsers (or Node.js) | Requires JVM | Requires Python interpreter |
| **Learning Curve** | Easy to start, challenging to master | Steep learning curve | Beginner-friendly |
| **Example** | // JavaScript Example: Dynamic Web Page<br>document.getElementById("demo").innerHTML = "Hello from JavaScript!"; | // Java Example: Backend Application<br>public class HelloWorld {<br>  public static void main(String[] args) {<br>    System.out.println("Hello from Java!");<br>  }<br>} | # Python Example: Data Processing<br>print("Hello from Python!") |

## JavaScript Syntax

### Variables and Constants

JavaScript provides three ways to declare variables: var, let, and const. Each has distinct characteristics and use cases.

**Comparison of var, let, and const**

| **Feature** | **var** | **let** | **const** | |
|---|---|---|---|---|
| **Scope** | Function-scoped | Block-scoped | Block-scoped | if (true)<br>{ var a = 10; // Function-scoped<br> let b = 20; // Block-scoped<br>const c = 30; // Block-scoped }<br>console.log(a); // Output: 10<br> console.log(b); // Error: b is not defined<br> console.log(c); // Error: c is not defined |
| **Hoisting** | Hoisted to the top, initialized as undefined. | Hoisted to the top, but not initialized. | Hoisted to the top, but not initialized. | console.log(x); // Output: undefined (var is hoisted)<br> console.log(y); // Error: Cannot access 'y' before initialization<br> console.log(z); // Error: Cannot access 'z' before initialization<br>var x = 10;<br>let y = 20;<br>const z = 30; |
| **Reassignment** | Allowed | Allowed | Not allowed | var x = 10;<br>x = 15; // Allowed<br>console.log(x); // Output: 15<br>let y = 20;<br>y = 25; // Allowed<br>console.log(y); // Output: 25<br>const z = 30;<br> z = 35; // Error: Assignment to constant variable<br>console.log(z); // Output: 30 |

| | | | | |
|---|---|---|---|---|
| **Redeclaration** | Allowed within the same scope | Not allowed within the same scope | Not allowed | var x = 10;<br>var x = 20; // Allowed<br>console.log(x); // Output: 20<br>let y = 30;<br>let y = 40; // Error:<br>Identifier 'y' has already been declared<br>const z = 50;<br>const z = 60; // Error: Identifier 'z' has already been declared |
| **Initialization** | Optional | Optional | Mandatory during declaration | var x ; // Allowed<br>let y ; // Allowed<br>const z ; // not allowed must be initialized |
| **Mutability** | Mutable | Mutable | Immutable (value cannot be reassigned) | |
| **Use Case** | For legacy code or function-scoped needs. | For variables that may change in value. | For constants or fixed values. | |

## Data types ofJavaScript

JavaScript has 7 primitive data types and 1 non-primitive data type. The primitive types represent single values, while the non-primitive type (object) can store collections of data.

### Primitive Data Types

**1. Number**

- Represents numeric values, including integers and floating-point numbers.
- Special numeric values: Infinity, -Infinity, and NaN (Not-a-Number).

**Example:**

```
let num1 = 42;         // Integer
let num2 = 3.14;        // Floating-point number
let infinityVal = 1 / 0; // Infinity
let nanVal = "abc" / 2; // NaN
console.log(num1, num2, infinityVal, nanVal);
```

**2. String**

- Represents textual data enclosed in single ('), double ("), or backticks (`).

**Example:**

```
let singleQuote = 'Hello';
let doubleQuote = "World";
let templateLiteral = `Hello, ${singleQuote}`;
console.log(singleQuote, doubleQuote, templateLiteral);
```

**3. Boolean**

- Represents logical values: true or false.

**Example:**

```
let  isJavaScriptFun = true;
let  isJavaScriptHard = false;
console.log(isJavaScriptFun, isJavaScriptHard);
```

### 4. Undefined

- A variable declared but not assigned a value.

**Example:**

```
let undefinedVar;
console.log(undefinedVar); // undefined
```

### 5. Null

- Represents an intentional absence of value.

**Example:**

```
let nullVar = null;
console.log(nullVar); // null
```

### 6. Symbol (Introduced in ES6)

- Represents unique and immutable values, often used as object keys.

**Example:**

```
let sym1 = Symbol("id");
let sym2 = Symbol("id");
console.log(sym1 === sym2); // false
```

### 7. BigInt (Introduced in ES2020)

- Represents integers larger than the Number type can safely store.

**Example:**

```
let bigIntVal = 1234567890123456789012345678901234567890n;
console.log(bigIntVal);
```

## Non-Primitive Data Types

### Object

- A collection of key-value pairs.
- Includes arrays, functions, and other objects.

**Example:**

```
let obj = { name: "Alice", age: 25 };
let arr = [1, 2, 3];
let func = function () { return "Hello"; };
console.log(obj, arr, func());
```

## Important Points

1. **Type Checking**: Use typeof to check the type of a variable.

**Example:**

```
console.log(typeof 42);          // "number"
console.log(typeof "Hello");     // "string"
console.log(typeof true);        // "boolean"
console.log(typeof undefined);   // "undefined"
console.log(typeof null);        // "object" (legacy bug)
console.log(typeof Symbol("id")); // "symbol"
console.log(typeof 123n);        // "bigint"
console.log(typeof {});          // "object"
console.log(typeof NaN);   // number
```

2. **Null vs Undefined**:
    - **undefined:** A variable is declared but not assigned a value.
    - **null:** Explicitly set to indicate "no value."

3. **Objects and Arrays**:
    - Objects are mutable, and their properties can be modified.
    - Arrays are special types of objects.

## Operators in JavaScript

Operators in JavaScript are used to perform operations on values and variables. They are categorized into several types based on their functionality.

### 1. Arithmetic Operators

It is used to perform mathematical operations.

| Operator | Description | Example | Result |
|---|---|---|---|
| + | Addition | 5 + 2 | 7 |
| - | Subtraction | 5 - 2 | 3 |
| * | Multiplication | 5 * 2 | 10 |
| / | Division | 5 / 2 | 2.5 |
| % | Modulus (Remainder) | 5 % 2 | 1 |
| ** | Exponentiation | 5 ** 2 | 25 |
| ++ | Increment | let x = 5; x++ | 6 |
| -- | Decrement | let x = 5; x-- | 4 |

### 2. Assignment Operators

It is used to assign values to variables.

| Operator | Description | Example | Result |
|---|---|---|---|
| = | Assign | x = 5 | x = 5 |
| += | Add and assign | x += 2 | x = x + 2 |
| -= | Subtract and assign | x -= 2 | x = x - 2 |
| *= | Multiply and assign | x *= 2 | x = x * 2 |
| /= | Divide and assign | x /= 2 | x = x / 2 |
| %= | Modulus and assign | x %= 2 | x = x % 2 |
| **= | Exponentiation and assign | x **= 2 | x = x ** 2 |

### 3. Comparison Operators

It is used to compare two values.

| Operator | Description | Example | Result |
|---|---|---|---|
| == | Equal to | 5 == '5' | true |
| === | Strict equal to | 5 === '5' | false |
| != | Not equal to | 5 != '5' | false |
| !== | Strict not equal to | 5 !== '5' | true |
| > | Greater than | 5 > 2 | true |
| < | Less than | 5 < 2 | false |
| >= | Greater than or equal to | 5 >= 5 | true |
| <= | Less than or equal to | 5 <= 2 | false |

### 4. Logical Operators

It used to combine conditional statements.

| Operator | Description | Example | Result |
|---|---|---|---|
| && | Logical AND | true && false | false |
| \|\| | Logical OR | true \|\|false | true |
| ! | Logical NOT | !true | false |

### 5. Bitwise Operators

It operate on binary representations of numbers.

| Operator | Description | Example | Result |
|---|---|---|---|
| & | AND | 5 & 1 | 1 |
| ` | ` | OR | `5 |
| ^ | XOR | 5 ^ 1 | 4 |
| ~ | NOT | ~5 | -6 |
| << | Left shift | 5 << 1 | 10 |
| >> | Right shift | 5 >> 1 | 2 |
| >>> | Zero-fill right shift | 5 >>> 1 | 2 |

### 6. String Operators

It used to manipulate strings.

| Operator | Description | Example | Result |
|---|---|---|---|
| + | Concatenation | 'Hello' + 'World' | 'HelloWorld' |
| += | Concatenate and assign | let x = 'Hi'; x += '!' | 'Hi!' |

### 7. Type Operators

It used to check or convert data types.

| Operator | Description | Example | Result |
|---|---|---|---|
| typeof | Returns the type of a value | typeof 42 | 'number' |
| instanceof | Checks if an object is an instance of a class | obj instanceof Object | true |

### 8. Ternary Operator

Shorthand for conditional expressions.

| Operator | Description | Example | Result |
|---|---|---|---|
| ? : | Conditional operator | **Ex1:** let x= 5 > 2 ? 'Yes' : 'No'<br>**Ex2:**<br>let age = 18;<br>let result = age >= 18 ? "Adult" : "Minor";<br>console.log(result); // "Adult" | Ex1: 'Yes'<br>Ex2: Adult |

### 9. Other Operators

| Operator | Description | Example | Result |
|---|---|---|---|
| , | Comma operator | let x = (1, 2, 3); | 3 |
| delete | Deletes a property from an object | delete obj.key; | true |
| in | Checks if a property exists in an object | 'key' in obj | true |
| void | Evaluates an expression but returns undefined | void 0 | undefined |

## Conditional Statements and Loops

JavaScript provides Conditional Statements to handle decision-making and Loops  to handle repetitive tasks.

## Conditional Statements

Control structures that execute specific code blocks based on whether a condition evaluates to true or false.

### 1. Conditional Statements

Conditional statements are used to perform different actions based on different conditions.

<table>
<tr><th>Statement</th><th>Description</th><th>Syntax</th><th>Example</th></tr>
<tr><td colspan="4">**if**<br>Executes a block of code if a specified condition is true.<br>**Syntax**<br>if (condition)<br>{ code }<br><br>**Example**<br>if (x > 0)<br>{ console.log("Positive"); }</td></tr>
<tr><td colspan="4">**if-else**<br>Executes one block of code if the condition is true, otherwise executes another block of code.<br>**Syntax**<br>if (condition) { code1 } else { code2 }<br><br>**Example**<br>if (x > 0)<br>{ console.log("Positive"); }<br>else<br>{ console.log("Negative"); }</td></tr>
<tr><td colspan="4">**if-else if-else**<br>Tests multiple conditions, executing the block of code for the first true condition.<br><br>**Syntax**<br>if (condition1)<br>{ code1 }<br>else if (condition2)<br>{ code2 }<br>else<br>{ code3 }<br><br>**Example**<br>if (x > 0) { console.log("Positive"); }<br>else if (x === 0) { console.log("Zero"); }<br>else { console.log("Negative"); }</td></tr>
<tr><td colspan="4">**switch**<br>Evaluates an expression, matching its value to multiple case clauses, and executes the matching block.<br>**Syntax**<br>switch(expression)<br>{ case value1: code;<br>break;<br>case value2: code;</td></tr>
</table>

```
break;
default: code; }
```

**Example**

```
switch (day)
{ case 1: console.log("Monday");
 break;
case 2: console.log("Tuesday");
break;
default: console.log("Invalid day");  }
```

## Loops

Control structures that repeatedly execute a block of code as long as a specified condition remains true.

| Loop | Description | Syntax | Example |
|---|---|---|---|
| for | Loops through a block of code a specified number of times. | for (initialization; condition; increment/decrement) { code } | for (let i = 0; i < 5; i++)<br>{ console.log(i); } |
| while | Loops through a block of code while a specified condition is true. | while (condition) { code } | let i = 0;<br>while (i < 5)<br>{ console.log(i);<br>i++; } |
| do-while | Executes the block of code once, and then repeats the loop while the condition is true. | do { code } while (condition); | let i = 0;<br>do {<br>console.log(i);<br>i++; }<br>while (i < 5); |
| for...in | Loops through the properties of an object. | for (key in object) { code } | let obj = { a: 1, b: 2 };<br>for (let key in obj)<br>{ console.log(key, obj[key]);<br>} |
| for...of | Loops through the values of an iterable object (like an array or string). | for (value of iterable) { code } | let arr = [1, 2, 3];<br>for (let value of arr)<br>{ console.log(value); } |

## Functions in JavaScript

Functions are reusable blocks of code designed to perform a specific task. They help in modularizing and organizing code efficiently.

### 1. Function Declaration and Invocation

- A **function declaration** defines a named function with the function keyword. It can be invoked (called) anywhere in the code after its definition.
- Function declarations are **hoisted**, meaning they can be called before their definition in the code.
- A function can take zero or more parameters and return a value.

**Syntax**

```
function functionName(parameters) {
  // code to execute    }
```

**Example**

```
function greet(name)
 {
  return ‘Hello, ${name}!’;   }
// Function Invocation
console.log(greet("Alice")); // Output: Hello, Alice!
console.log(greet("Bob"));   // Output: Hello, Bob!
```

## 2. Function Expressions

- A **function expression** defines a function as part of an expression. It can be anonymous or named and is often assigned to a variable.
- Function expressions are **not hoisted**, meaning they cannot be called before their definition.
- Commonly used in callbacks and event handlers.

**Syntax**

```
const functionName = function(parameters) {
 // code to execute   };
```

**Example1**

**// Anonymous function expression**

```
  const add = function(a, b) {
    return a + b;
  };
  console.log(add(5, 3)); // Output: 8
```

**Example2**

**// Named function expression**

```
  const multiply = function multiplyNumbers(a, b) {
    return a * b;
  };
  console.log(multiply(4, 2)); // Output: 8
```

## 3. Arrow Functions

- Arrow functions are a concise way to write functions using the => (arrow) syntax. They are especially useful for writing shorter functions.
- Arrow functions do not have their own this context; they inherit this from the surrounding scope.
- For single-line functions, the return keyword can be omitted.

**Syntax**

```
const functionName = (parameters) => {
 // code to execute    };
```

**Example**

```
// Single-line arrow function
const square = (x) => x * x;
console.log(square(4)); // Output: 16

// Multi-line arrow function
const subtract = (a, b) => {   return a - b;   };
console.log(subtract(10, 3)); // Output: 7
```

## 4. Parameters and Return Values

- Functions can take input values (parameters) and optionally return an output value.
- Parameters are placeholders for values passed into the function.
- Functions can have default parameter values.

**Example1**

```
function greet(name = "Guest") {
  return ‘Hello, ${name}!;’  }
console.log(greet());       // Output: Hello, Guest!
console.log(greet("Alice")); // Output: Hello, Alice!
```

**Return Values**

- The return statement specifies the value to be returned by the function.
- If no return is specified, the function returns undefined.

**Example2**

```
function multiply(a, b) {
  return a * b;   }
let result = multiply(3, 4);
console.log(result); // Output: 12
```

## Introduction to DOM (Document Object Model)

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the structure of a webpage as a tree of objects, allowing developers to interact with and manipulate the content, structure, and style of the document programmatically.

### Property of the DOM

1. **Tree Structure**: The DOM represents the HTML document as a hierarchical tree structure where each element is a node.
2. **Programming Interface**: It provides methods and properties to interact with HTML and CSS programmatically.
3. **Dynamic Updates**: Using the DOM, you can dynamically update the content and structure of a webpage without reloading it.
4. **Language-Independent**: The DOM can be used with various programming languages, though JavaScript is the most commonly used language.

### DOM Tree Structure

The DOM tree consists of different types of nodes:

- **Document Node**: Represents the entire document.
- **Element Nodes**: Represent HTML elements.
- **Text Nodes**: Represent the text content within elements.
- **Attribute Nodes**: Represent the attributes of elements.
- **Comment Nodes**: Represent comments in the HTML.

**Example:**

| **For the following HTML:** | **The DOM tree representation is:** |
|---|---|
| **<!DOCTYPE html>** | **Document** |
| **<html>** | └── **html** |
| **<head>** | ├── **head** |
| **<title>My Page</title>** | │ └── **title** |
| **</head>** | │ └── **"My Page"** |
| **<body>** | └── **body** |
| **<h1>Welcome</h1>** | ├── **h1** |
| **<p>This is a paragraph.</p>** | │ └── **"Welcome"** |
| **</body>** | └── **p** |
| **</html>** | └── **"This is a paragraph."** |

## Accessing elements

JavaScript provides the document object to interact with the DOM. Common methods include:

| Method | Description | Example | Output |
|---|---|---|---|
| **getElementById()**<br>• Selects an element based on its unique id attribute.<br>• Returns the first matching element or null if no element is found.<br>document.getElementById("title")<br>First element with id="title".<br>**Example:**<br>&lt;h1 id="title"&gt;Welcome&lt;/h1&gt;<br>&lt;script&gt; let element = document.getElementById("title");<br>console.log(element.textContent); // Output: Welcome &lt;/script&gt; | | | |
| **getElementsByClassName()**<br>• Selects elements by class name.<br>document.getElementsByClassName("text")[0]<br>First element with class="text".<br>**Example:**<br>&lt;p class="text"&gt;Paragraph 1&lt;/p&gt;<br>&lt;p class="text"&gt;Paragraph 2&lt;/p&gt;<br>&lt;script&gt;<br>let elements = document.getElementsByClassName("text");<br>console.log(elements[0].textContent); // Output: Paragraph 1<br>&lt;/script&gt; | | | |
| **getElementsByTagName()**<br>• Selects elements by tag name.<br>document.getElementsByTagName("p")[0]<br>First &lt;p&gt; element.<br>**Example:**<br>&lt;div&gt;Div 1&lt;/div&gt;<br>&lt;div&gt;Div 2&lt;/div&gt;<br>&lt;script&gt; let elements = document.getElementsByTagName("div");<br>console.log(elements[1].textContent); // Output: Div 2 &lt;/script&gt; | | | |
| **querySelector()**<br>• Selects the first matching CSS selector.<br>• document.querySelector(".text")<br>• First element with class="text".<br>**Example:**<br>&lt;p class="text"&gt;Paragraph 1&lt;/p&gt;<br>&lt;p class="text"&gt;Paragraph 2&lt;/p&gt;<br>&lt;script&gt;<br>let element = document.querySelector(".text");<br>console.log(element.textContent); // Output: Paragraph 1<br>&lt;/script&gt; | | | |

**querySelectorAll()**

- Selects all matching CSS selectors.
- document.querySelectorAll(".text")
- NodeList of all .text elements.

**Example:**

```
<p class="text">Paragraph 1</p>
<p class="text">Paragraph 2</p>
<script>
  let elements = document.querySelectorAll(".text");
  elements.forEach((el) => console.log(el.textContent));
  // Output: Paragraph 1
  // Output: Paragraph 2
</script>
```

**document.forms**

- Accesses all <form> elements in the document.
- document.forms["loginForm"]
- Form with id="loginForm".

**Example:**

```
<form id="loginForm"></form>
<script>
  let form = document.forms["loginForm"];
  console.log(form.id); // Output: loginForm
</script>
```

**document.images**

- Accesses all <img> elements.
- document.images[0]
- First <img> element.

**Example:**

```
<img src="image.jpg" alt="Example">
<script> let images = document.images;
console.log(images[0].alt);   // Output: Example   </script>
```

**document.links**

- Accesses all <a> with href attributes.
- document.links[0]
- First <a> element with href.

**Example:**

```
<a href="https://example.com">Example Link</a>
<script>
  let links = document.links;
  console.log(links[0].href); // Output: https://example.com/
</script>
```

## Modifying Elements

Modifying elements in the DOM allows developers to dynamically change the content, attributes, styles, and structure of a webpage.

| Method/Property | Description | Example | Output |
|---|---|---|---|

**innerHTML**

- Sets/retrieves HTML content inside an element.
- Can be used to insert or replace content, including HTML tags.
- element.innerHTML = "<b>Hi</b>";
- **output:** <b>Hi</b>

**Example:**

```
<div id="content">Original Content</div>
<script>
 let element = document.getElementById("content");
 element.innerHTML = "<strong>Updated Content</strong>";
 console.log(element.innerHTML);  // Output: <strong>Updated Content</strong>
</script>
```

**textContent**

- Sets/retrieves plain text content inside an element.
- Ignores any HTML tags and treats them as plain text.
- element.textContent = "Hi"; //Output: Hi

**Example:**

```
<div id="content">Original <strong>Content</strong></div>
<script>
 let element = document.getElementById("content");
 element.textContent = "Updated Content";
 console.log(element.textContent); // Output: Updated Content
</script>
```

**setAttribute()**

- Sets an attribute value on an element.
- element.setAttribute("src", "new.jpg");  //Output: Attribute updated.

**Example:**

```
<img id="image" src="old.jpg" alt="Old Image">
<script>
 let img = document.getElementById("image");
 img.setAttribute("src", "new.jpg");
 console.log(img.src);  // Output: URL of new.jpg
</script>
```

**getAttribute()**

- Retrieves the value of a specified attribute on an element..
- element.getAttribute("alt"); //Output: Returns attribute value.

**Example:**

```
<img id="image" src="example.jpg" alt="Example Image">
<script>
 let img = document.getElementById("image");
 console.log(img.getAttribute("alt")); // Output: Example Image     </script>
```

**removeAttribute()**

- Removes an attribute from an element.
- element.removeAttribute("alt"); //Output: Attribute removed.

**Example:**

```
<img id="image" src="example.jpg" alt="Example Image">
<script>
  let img = document.getElementById("image");
  img.removeAttribute("alt");
  console.log(img.hasAttribute("alt")); // Output: false
</script>
```

**style**

- Modifies inline CSS styles of an element.
- element.style.color = "red"; //Output: Style applied.

**Example:**

```
<p id="text">Hello, World!</p>
<script>
  let text = document.getElementById("text");
  text.style.color = "blue";
  text.style.fontSize = "20px";
  console.log(text.style.color); // Output: blue
</script>
```

**classList**

- Manages classes of an element. Provides methods to add, remove, toggle, or check classes on an element.
- element.classList.add("active");
- Class added/removed/toggled.

**Example:**

```
<div id="box" class="red"></div>
<script>
  let box = document.getElementById("box");
  box.classList.add("blue");
  box.classList.remove("red");
  box.classList.toggle("green");
  console.log(box.classList); // Output: DOMTokenList ["blue", "green"]
</script>
```

**appendChild()**

Adds a new child element.

parent.appendChild(child);

Child added to parent.

**removeChild()**

Removes a child element.

parent.removeChild(child);

Child removed.

**replaceChild()**

Replaces a child element with a new one.

parent.replaceChild(newChild, oldChild);

Child replaced.

## Events and Event Handling in JavaScript

- **Events** in JavaScript are actions or occurrences that happen in the browser, such as user interactions (clicking, typing, scrolling), loading resources, or other activities.
- **Event Handling** refers to the process of responding to these events by executing code.

## Event Types in JavaScript

JavaScript provides a wide range of events, categorized into different types based on their purpose:

| Event Type | Description | Common Events |
|---|---|---|
| **Mouse Events** | Triggered by mouse actions. | **click:** Triggered when an element is clicked.<br>button.addEventListener("click", () => alert("Button clicked!")); |
| | | **dblclick:** Triggered when an element is double-clicked.<br>button.addEventListener("dblclick", () => alert("Button double-clicked!")); |
| | | **mousedown:** Triggered when the mouse button is pressed down.<br>div.addEventListener("mousedown", () => console.log("Mouse button pressed!")); |
| | | **mouseup:**<br>Triggered when the mouse button is released.<br>div.addEventListener("mouseup", () => console.log("Mouse button released!")); |
| | | **mousemove:** Triggered when the mouse is moved over an element.<br>div.addEventListener("mousemove", () => console.log("Mouse moved!")); |
| | | **mouseover:** Triggered when the mouse enters the area of an element.<br>div.addEventListener("mouseover", () => console.log("Mouse over!")); |
| | | **mouseout:** Triggered when the mouse leaves the area of an element.<br>div.addEventListener("mouseout", () => console.log("Mouse out!")); |
| **Keyboard Events** | Triggered by keyboard interactions. | **Keydown:** Triggered when a key is pressed down.<br>document.addEventListener("keydown", (e) => console.log(e.key)); |
| | | **Keyup:** Triggered when a key is released.<br>document.addEventListener("keyup", (e) => console.log("Key released: " + e.key)); |
| | | **Keypress:** Triggered when a key is pressed (deprecated in modern browsers).<br>document.addEventListener("keypress", (e) => console.log("Key pressed: " + e.key)); |
| **Form Events** | Triggered by form actions. | **submit:** Triggered when a form is submitted.<br>form.addEventListener("submit", (e) => { e.preventDefault(); alert("Form submitted!"); }); |
| | | **change:** Triggered when the value of an input, select, or textarea changes.<br>input.addEventListener("change", () => console.log("Value changed!")); |
| | | **focus:** Triggered when an element gains focus.<br>input.addEventListener("focus", () => console.log("Input focused!")); |

| | | |
|---|---|---|
| | | **blur:** Triggered when an element loses focus.<br>input.addEventListener("blur", () => console.log("Input blurred!")); |
| | | **input:** Triggered when the value of an input element is changed.<br>input.addEventListener("input", () => console.log("Input value updated!")); |
| **Window Events** | Triggered by actions related to the browser window. | **load:** Triggered when the page is fully loaded.<br>window.addEventListener("load", () => console.log("Page loaded!")); |
| | | **resize:** Triggered when the browser window is resized.<br>window.addEventListener("resize", () => console.log("Window resized!")); |
| | | **scroll:** window.addEventListener("scroll", () => console.log("Page scrolled!"));<br>Triggered when the user leaves the page (deprecated). |
| | | **unload:** Triggered when the user scrolls the page or an element.<br>window.addEventListener("unload", () => console.log("Page unloading!")); |
| **Touch Events** | Triggered by touch interactions on touch-enabled devices. | **touchstart:** riggered when a touch starts on a touch-enabled device.<br>document.addEventListener("touchstart", () => console.log("Touch started!")); |
| | | **touchmove:** Triggered when a touch moves across the screen.<br>document.addEventListener("touchmove", () => console.log("Touch moved!")); |
| | | **touchend:** Triggered when a touch ends.<br>document.addEventListener("touchend", () => console.log("Touch ended!")); |
| **Clipboard Events** | Triggered by clipboard operations. | **copy:** Triggered when content is copied to the clipboard.<br>document.addEventListener("copy", () => console.log("Content copied!")); |
| | | **cut:** Triggered when content is cut to the clipboard.<br>document.addEventListener("cut", () => console.log("Content cut!")); |
| | | **paste:** Triggered when content is pasted from the clipboard.<br>document.addEventListener("paste", () => console.log("Content pasted!")); |
| **Drag Events** | Triggered by drag-and-drop interactions. | **drag:** Triggered when an element is being dragged.<br>element.addEventListener("drag", () => console.log("Dragging!")); |
| | | **dragstart:** Triggered when dragging starts.<br>element.addEventListener("dragstart", () => console.log("Drag started!")); |
| | | **dragend :** Triggered when dragging ends.<br>element.addEventListener("dragend", () => console.log("Drag ended!")); |
| | | **drop:** Triggered when an element is dropped.<br>element.addEventListener("drop", () => console.log("Dropped!")); |
| **Media Events** | Triggered by media (audio/video) playback. | **play:** Triggered when media playback starts.<br>video.addEventListener("play", () => console.log("Playing!")); |

| | | |
|---|---|---|
| | | **pause:** Triggered when media playback is paused.<br>video.addEventListener("pause", () => console.log("Paused!")); |
| | | **ended:** Triggered when media playback ends.<br>video.addEventListener("ended", () => console.log("Playback ended!")); |
| | | **volumechange:** Triggered when the volume is changed.<br>video.addEventListener("volumechange", () => console.log("Volume changed!")); |
| **Focus Events** | Triggered when an element gains or loses focus. | **focus:** Triggered when an element gains focus.<br>input.addEventListener("focus", () => console.log("Input focused!")); |
| | | **blur:** Triggered when an element loses focus.<br>input.addEventListener("blur", () => console.log("Input blurred!")); |
| **Animation Events** | Triggered by CSS animations. | **animationstart** : Triggered when a CSS animation starts.<br>element.addEventListener("animationstart", () => console.log("Animation started!")); |
| | | **animationend:** Triggered when a CSS animation ends.<br>element.addEventListener("animationend", () => console.log("Animation ended!")); |
| | | **animationiteration:** Triggered when a CSS animation iteration is completed.<br>element.addEventListener("animationiteration", () => console.log("Animation iteration!")); |
| **Transition Events** | Triggered by CSS transitions. | **transitionstart:** Triggered when a CSS transition starts.<br>element.addEventListener("transitionstart", () => console.log("Transition started!")); |
| | | **transitionend:** Triggered when a CSS transition ends.<br>element.addEventListener("transitionend", () => console.log("Transition ended!")); |
| **Pointer Events** | Unified events for mouse, touch, and pen interactions. | **pointerdown:**<br>Triggered when a pointer (mouse, touch, or pen) is pressed down.<br>element.addEventListener("pointerdown", () => console.log("Pointer down!")); |
| | | **pointermove:** Triggered when a pointer moves.<br>element.addEventListener("pointermove", () => console.log("Pointer moved!")); |
| | | **pointerup:** Triggered when a pointer is released.<br>element.addEventListener("pointerup", () => console.log("Pointer up!")); |
| **Custom Events** | User-defined events created using the CustomEvent constructor. | **N/A** |

## Adding Event Listeners and Event Delegation in JavaScript

### Adding Event Listeners

An event listener is a function that waits for an event to occur on a specific element and then executes a specified callback function.

**Syntax**

```
element.addEventListener(event, callbackFunction, useCapture);
```

| Parameter | Description |
|---|---|
| event | The name of the event to listen for (e.g., "click", "mouseover"). |
| callbackFunction | The function to execute when the event occurs. |
| useCapture | Optional boolean indicating the phase to handle the event (true for capture, false for bubble). |

**Example 1: Adding a Click Event Listener**

```
const button = document.getElementById("myButton");
button.addEventListener("click", () => {
  alert("Button clicked!");
});
```

### Event Delegation

Event delegation is a technique where you attach a single event listener to a parent element to handle events on its child elements. This is useful for managing events dynamically or efficiently when there are many child elements.

**Event Delegation is used to:**

- Reduces the number of event listeners in your code.
- Efficient for dynamically created elements.
- Simplifies event handling for similar child elements.

### How Event Delegation Works

Event delegation relies on **event bubbling**, where an event triggered on a child element propagates up to its parent elements.

**Syntax**

```
parentElement.addEventListener(event, (e) => {
  if (e.target.matches(selector)) {
    callbackFunction(e);
  }
});
```

| Parameter | Description |
|---|---|
| parentElement | The parent element where the event listener is attached. |
| event | The name of the event to listen for (e.g., "click", "change"). |
| e.target | The element that triggered the event. |
| selector | A CSS selector to match the child elements for which the event should be handled. |

## Arrays in JavaScript

An **array** is a data structure used to store multiple values in a single variable. Arrays in JavaScript are dynamic and can hold values of different types.

**Property of JavaScript array:**

1. **Dynamic Nature**: Arrays in JavaScript can grow or shrink dynamically. Many methods (map(), reduce(), etc.) rely on custom callback functions for flexibility.
2. **Chaining**: Methods like map() and filter() can be chained for complex operations.
3. **Mutability**: Methods like push(), pop(), shift(), unshift(), sort() and reverse() modify the original array, while map(), filter() and reduce() return new arrays or values.

## Creating Arrays

**1. Using Array Literals**

**Syntax:** let arrayName = [value1, value2, ...];

**Example:**

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits); // Output: ["Apple", "Banana", "Cherry"]
```

**2. Using the Array Constructor**

**Syntax:** let arrayName = new Array(size or elements);

**Example:**

```
let numbers = new Array(5); // Creates an array with 5 empty slots
let colors = new Array("Red", "Green", "Blue");
console.log(colors); // Output: ["Red", "Green", "Blue"]
```

## Manipulating Arrays

| Method | Description | Syntax | Example | Output |
|---|---|---|---|---|
| **push()**<br>• Adds one or more elements to the end of an array.<br>• Returns the new length of the array.<br>**Syntax:**<br>array.push(element1, element2);<br>**Example:**<br>let fruits = ["Apple", "Banana"];<br>fruits.push("Cherry");<br>console.log(fruits); // Output: ["Apple", "Banana", "Cherry"] | | | | |
| **pop()**<br>• Removes the last element from an array.<br>• Returns the removed element.<br>**Syntax :**<br>array.pop();<br>**Example:**<br>let fruits = ["Apple", "Banana", "Cherry"];<br>let lastFruit = fruits.pop();<br>console.log(fruits); // Output: ["Apple", "Banana"]<br>console.log(lastFruit); // Output: "Cherry" | | | | |

**shift()**

- Removes the first element from an array.
- Returns the removed element.

**Syntax:**

```
array.shift();
```

**Example:**

```
let fruits = ["Apple", "Banana", "Cherry"];
let firstFruit = fruits.shift();
console.log(fruits); // Output: ["Banana", "Cherry"]
console.log(firstFruit); // Output: "Apple"
```

**unshift()**

- Adds one or more elements to the beginning of an array.
- Returns the new length of the array.

**Syntax:**

```
array.unshift(element1, element2);
```

**Example:**

```
let fruits = ["Banana", "Cherry"];
fruits.unshift("Apple");
console.log(fruits); // Output: ["Apple", "Banana", "Cherry"]
```

**map()**

- Creates a new array by applying a function to each element of the original array.

**Syntax:**

```
array.map(callback);
```

**Example:**

```
let numbers = [1, 2, 3];
let squared = numbers.map(num => num * num);
console.log(squared); // Output: [1, 4, 9]
```

**filter()**

- Creates a new array with elements that pass a test implemented by a function.

**Syntax:**

```
array.filter(callback);
```

**Example:**

```
let numbers = [1, 2, 3, 4, 5];
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

## Some Advanced Array Methods

| Method | Description | Syntax | Example | Output |
|---|---|---|---|---|

**reduce()**

- Combines all elements of an array into a single value by applying a function iteratively.
- The function takes an **accumulator** (previous result) and the current element as arguments.

**Syntax:**

```
array.reduce(callback(accumulator, currentValue, index, array), initialValue);
```

**Example:**

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Output: 10
```

**find()**

- Returns the first element in the array that satisfies a given condition.
- If no element satisfies the condition, it returns undefined.

**Syntax:**

```
array.find(callback(element, index, array));
```

**Example:**

```
let numbers = [1, 3, 5, 7, 8];
let firstEven = numbers.find(num => num % 2 === 0);
console.log(firstEven); // Output: 8
```

**findIndex()**

- Returns the index of the first element that satisfies a given condition.
- If no element satisfies the condition, it returns -1.

**Syntax:**

```
array.findIndex(callback(element, index, array));
```

**Example:**

```
let numbers = [1, 3, 5, 7, 8];
let firstEvenIndex = numbers.findIndex(num => num % 2 === 0);
console.log(firstEvenIndex); // Output: 4
```

**every()**

- Tests whether all elements in the array satisfy a given condition.
- Returns true if all elements pass the test; otherwise, false.

**Syntax:**

```
array.every(callback(element, index, array));
```

**Example:**

```
let numbers = [2, 4, 6, 8];
let allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // Output: true
```

**some()**

- Tests whether at least one element in the array satisfies a given condition.
- Returns true if any element passes the test; otherwise, false.

**Syntax:**

```
array.some(callback(element, index, array));
```

**Example:**

```
let numbers = [1, 3, 5, 7, 8];
let hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // Output: true
```

**sort()**

- Sorts the elements of an array in place.
- By default, sorts elements as strings in ascending order.
- A custom compare function can be provided for numerical or complex sorting.

**Syntax:**

```
array.sort(compareFunction);
```

**Example:**

```
let numbers = [10, 3, 7, 1];
numbers.sort((a, b) => a - b); // Ascending order
console.log(numbers); // Output: [1, 3, 7, 10]
```

**reverse()**

- Reverses the order of elements in an array in place.

**Syntax:**

```
array.reverse();
```

**Example:**

```
let numbers = [1, 2, 3, 4];
numbers.reverse();
console.log(numbers); // Output: [4, 3, 2, 1]
```

## Objects in JavaScript

Objects in JavaScript are collections of properties and methods. They represent real-world entities and are a cornerstone of JavaScript programming.

### 1. Properties

- Properties are key-value pairs associated with an object.
- The key is always a string (or symbol), and the value can be any data type (string, number, object, function, etc.).

**Syntax:**

```
 let objectName = {
property1: value1,
property2: value2,
};
```

**Example:**

```
let car = {
brand: "Toyota",
model: "Corolla",
year: 2022,
};
console.log(car.brand); // Output: Toyota
```

### 2. Methods

- Methods are functions defined as properties of an object.
- They allow objects to perform actions.

**Syntax:**

```
let objectName = {
  methodName: function() {
    // Code here
   },   };
```

**Example:**

```
let calculator = {
  add: function(a, b) {
    return a + b;
  },
};
console.log(calculator.add(5, 3)); // Output: 8
```

### 3. The this Keyword

- Refers to the object in which it is used.
- In methods, this points to the object calling the method.

**Example:**

```
let person = { name: "Alice",   greet: function() {
                                    return ‘Hello, my name is ${this.name}’;   },   };
console.log(person.greet()); // Output: Hello, my name is Alice
```

### Accessing Properties and Methods

| Access Type | Syntax | Example | Output |
|---|---|---|---|
| Dot Notation | object.property | car.brand | Toyota |
| Bracket Notation | object["property"] | car["model"] | Corolla |
| Accessing Methods | object.method() | calculator.add(5, 3) | 8 |
| Using this | this.property inside a method | person.greet() | Hello, my name is Alice |

### Adding, Modifying, and Deleting Properties

| Operation | Syntax | Example | Output |
|---|---|---|---|
| Add Property | object.newProp = value; | car.color = "Red"; | Adds color: "Red" |
| Modify Property | object.property = newValue; | car.brand = "Honda"; | Updates brand to Honda |
| Delete Property | delete object.property; | delete car.year; | Removes year |

### Object Creation

| Type | Description | Syntax |
|---|---|---|
| Object Literals | The most common way to create objects. | let user = { name: "John", age: 30 }; |
| Using new Object(): | Creates an empty object and adds properties later. | let user = new Object();<br>user.name = "John";<br>user.age = 30; |
| Using Constructor Functions | Used to create multiple objects of the same type. | function User(name, age) {<br>  this.name = name;<br>  this.age = age;<br>}<br>let user1 = new User("Alice", 25);<br>let user2 = new User("Bob", 30); |

| | | |
|---|---|---|
| Using Classes | Modern syntax for creating objects with methods and properties. | class User {<br>  constructor(name, age) {<br>    this.name = name;<br>    this.age = age;<br>  }<br>  greet() {<br>    return `Hi, I'm ${this.name}`;<br>  }<br>}<br>let user = new User("Alice", 25);<br>console.log(user.greet()); // Output: Hi, I'm Alice |

## Introduction to Asynchronous Programming

Asynchronous programming allows a program to handle tasks like file reading, network requests, or timers without blocking the main thread. This ensures that applications remain responsive even when performing time-consuming operations.

## Features of Asynchronous Programming

| **Concept** | **Description** |
|---|---|
| **Synchronous Execution** | Tasks are executed one at a time in a sequence, blocking the next task until the current one completes. |
| **Asynchronous Execution** | Tasks are executed without waiting for the previous task to complete, enabling non-blocking behavior. |

| **Synchronous Example** | **Asynchronous Example** |
|---|---|
| console.log("Start");<br>for (let i = 0; i < 3; i++) {<br>  console.log(i);<br>}<br>console.log("End");<br><br>Output:<br>Start<br>0<br>1<br>2<br>End | console.log("Start");<br>setTimeout(() => console.log("Asynchronous Task"), 1000);<br>console.log("End");<br>Output:<br>Start<br>End<br>Asynchronous Task |

## Callbacks

A callback is a function passed as an argument to another function, executed after the completion of an asynchronous operation.

**Example:** Imagine ordering pizza. You give your phone number to the pizza shop (callback). When the pizza is ready, they call you to notify you (execute the callback).

**Syntax**

```
function asyncOperation(callback) {
   setTimeout(() => {
      console.log("Operation Complete");   callback(); }, 1000);
}
```

**Example**

```
function displayMessage() {
   console.log("Callback executed!");
}
console.log("Start");
asyncOperation(displayMessage);
console.log("End");
```

**Output:**

```
Start
End
Operation Complete
Callback executed!
```

**Advantages**

- Simple to use for basic tasks.

**Disadvantages**

- Can lead to "callback hell."

**Note: Callback Hell:** Callback hell happens when you use too many **nested callbacks**, making your code hard to read, debug, and maintain. It looks like a "pyramid" or "ladder" because one function depends on the result of the previous function.

## Promises

- A **Promise** is an object that represents a task that will complete in the future (success or failure). It has three states:

**States of a Promise**

| State | Description |
|---|---|
| **Pending** | Initial state, neither fulfilled nor rejected. |
| **Fulfilled** | Operation completed successfully. |
| **Rejected** | Operation failed. |

**Syntax**

```
const promise = new Promise((resolve, reject) => {
  if (condition) {
    resolve("Success");
  } else {
    reject("Error");
  }
});
```

**Example**

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully!");
    }, 1000);
  });
};
fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

**Output:**

```
Data fetched successfully!
```

**Advantages**

Avoids callback hell.

Chainable .then() and .catch().

**Disadvantages**

Requires more code for simple operations.

Still requires careful error handling.

## Async/Await

Async/await is a syntactic sugar over promises, making asynchronous code look and behave more like synchronous code.

**Example:** Instead of constantly checking for your package (callbacks or .then), you simply wait until it's delivered (await).

**Syntax**

```
async function asyncFunction() {
  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}
```

**Example**

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully!");
    }, 1000);
  });
};
async function fetchAndDisplay() {
  console.log("Fetching data...");
  const data = await fetchData();
  console.log(data);
}
fetchAndDisplay();
```

**Output:**

```
Fetching data...
Data fetched successfully!
```

**Advantages**

- Simplifies asynchronous code.
- Better error handling with try-catch.

**Disadvantages**

- Requires modern JavaScript environments.
- May block execution within await.

## Comparison of Callbacks, Promises, and Async/Await

| Feature | Callbacks | Promises | Async/Await |
|---|---|---|---|
| **Ease of Use** | Basic but leads to callback hell. | Cleaner syntax with .then() and .catch(). | Simplifies code, resembles synchronous flow. |
| **Error Handling** | Difficult to manage. | Handled using .catch(). | Managed using try-catch. |
| **Code Readability** | Poor in nested callbacks. | Better readability. | Excellent readability. |
| **Example** | callbackFunction() | promise.then().catch() | await promise |

## Fetch API for Making HTTP Requests

The **Fetch API** is a modern JavaScript interface for making HTTP requests to servers. It is built on promises and allows you to perform tasks like retrieving data from APIs, submitting forms, or interacting with RESTful services in a clean and readable way.

### Features of Fetch API

| Feature | Description |
|---|---|
| **Promise-Based** | The Fetch API uses promises, making it easier to handle asynchronous requests. |
| **Default Method** | The default HTTP method is GET, but you can specify others like POST, PUT, and DELETE. |
| **Response Object** | The response object provides methods like .json(), .text(), and .blob() to parse responses. |
| **Supports Headers** | You can set custom headers for requests. |

### Syntax

```
fetch(url, options)
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });
```

| Parameter | Description |
|---|---|
| url | The URL to which the request is sent. |
| options | Optional object specifying request details like method, headers, body, etc. |

## Making HTTP Requests Using Fetch

1. **GET Request:** Used to retrieve data from a server.
2. **POST Request:** Used to send data to a server.
3. **PUT Request:** Used to update existing data on the server.
4. **DELETE Request:** Used to delete data from the server.

### Handling Responses

The response object provides methods to process the data:

| Method | Description | Example |
|---|---|---|
| .json() | Parses the response as JSON. | response.json() |
| .text() | Parses the response as plain text. | response.text() |
| .blob() | Parses the response as a binary large object (e.g., images). | response.blob() |
| .status | Returns the HTTP status code of the response. | response.status |
| .ok | Returns true if the HTTP status code is in the range 200–299. | response.ok |

## Comparison of Fetch API with XMLHttpRequest

Would you like to explore **error handling** or **real-world examples** in more depth?

| **Feature** | **Fetch API** | **XMLHttpRequest** |
|---|---|---|
| **Ease of Use** | Cleaner syntax with promises. | Callback-based, less readable. |
| **Built-In JSON Support** | Yes, using .json(). | No, requires manual parsing. |
| **Modernity** | New and widely adopted. | Older and less commonly used now. |

## Error Handling and Debugging in JavaScript

Effective error handling and debugging are essential to ensure smooth execution of JavaScript code. JavaScript provides tools like try...catch blocks for error handling and browser debugging tools for identifying and resolving issues.

### Error Handling: try...catch Block

The try...catch block allows you to handle runtime errors gracefully without breaking the entire application. Errors in the try block are caught in the catch block, where you can handle them.

**Syntax**

```
try {
  // Code that might throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Optional: Code that always executes
}
```

| **Keyword** | **Description** |
|---|---|
| try | Block of code to test for errors. |
| catch | Block of code to handle errors. The error object contains details about the error. |
| finally | Optional block that always executes after try or catch, regardless of an error. |

**Error Handling Methods**

| **Method** | **Description** | **Example** |
|---|---|---|
| try...catch | Handles runtime errors. | Catching JSON parsing errors. |
| throw | Manually throws an error. | throw new Error("Custom error message"); |
| finally | Executes code after try/catch, regardless of success or failure. | Cleaning up resources like closing a file or database connection. |

### Debugging Tools in the Browser

Modern browsers provide powerful tools to debug JavaScript applications. These tools allow you to inspect code, set breakpoints, and monitor application behavior.

**1. Console:** The console is used to log messages, errors, and warnings.

| **Command** | **Description** | **Example** |
|---|---|---|
| console.log() | Logs messages to the console. | console.log("Hello, World!"); |
| console.error() | Logs error messages. | console.error("An error occurred!"); |
| console.warn() | Logs warnings. | console.warn("This is a warning!"); |

**2. Breakpoints:** Breakpoints pause code execution to allow step-by-step debugging.

**Steps to Use Breakpoints:**

1. Open the browser’s Developer Tools (F12 or Ctrl+Shift+I).
2. Navigate to the **Sources** tab.
3. Locate the script and click on the line number to set a breakpoint.
4. Refresh the page to trigger the breakpoint.

**3. Call Stack:** The call stack shows the sequence of function calls leading to the current execution point.

**4. Watch Expressions:** Monitor specific variables or expressions during debugging.

**Steps:**

1. Open the Developer Tools.
2. Go to the **Watch** section in the **Sources** tab.
3. Add variables or expressions to watch their values in real-time.

**5. Network Tab:** The **Network** tab helps debug HTTP requests and responses.

| **Feature** | **Description** |
|---|---|
| Status Code | Shows HTTP status codes like 200, 404, etc. |
| Headers | Displays request and response headers. |
| Payload | Shows the data sent in requests. |

**6. Debugger Keyword :** The debugger keyword pauses code execution at a specific point.