JAVASCRIPT

JavaScript হচ্ছে একটি বহুল ব্যবহৃত প্রোগ্রামিং ভাষা যা সাধারণত web পেজে dynamic কন্টেন্ট যোগ করার জন্য ব্যবহৃত হয়। এটি client-side এবং server-side দুই ক্ষেত্রেই ব্যবহৃত হয়। JavaScript কে সংক্ষেপে **JS** বলা হয়।
এটি **HTML** এবং **CSS** এর সাথে মিলে web পেজকে আরও ইন্টারেক্টিভ এবং আকর্ষণীয় করে তোলে।

🧐 What is JavaScript?

JavaScript হচ্ছে একটি **high-level**, **interpreted** প্রোগ্রামিং ভাষা যা **ECMAScript** স্ট্যান্ডার্ড অনুসারে কাজ করে। JavaScript কে সাধারণত web ডেভেলপমেন্টে ব্যবহার করা হয় dynamic কন্টেন্ট, যেমন ইন্টারেক্টিভ ফর্ম, অ্যানিমেশন, এবং আরও অনেক কিছু যোগ করার জন্য। এটি **HTML** এবং **CSS** এর সাথে মিলে web পেজের ইন্টারঅ্যাকশন বাড়ায়।

📜 History of JavaScript

JavaScript এর জন্ম হয়েছিল ১৯৯৫ সালে **Netscape** কোম্পানিতে। শুরুতে এর নাম ছিল **Mocha**, পরে নামকরণ করা হয় **LiveScript**, এবং শেষে রাখা হয় **JavaScript**। এর মূল উদ্দেশ্য ছিল web browser এ ইন্টারেক্টিভ ফিচার যোগ করা। বর্তমানে JavaScript অনেক বেশি উন্নত হয়েছে এবং এটি web ডেভেলপমেন্টের পাশাপাশি server-side প্রোগ্রামিং, mobile app ডেভেলপমেন্ট, এবং আরও অনেক ক্ষেত্রে ব্যবহৃত হয়।

✨ Features of JavaScript

JavaScript এর কিছু গুরুত্বপূর্ণ ফিচার নিচে দেওয়া হলোঃ

1. **Lightweight and Interpreted**: JavaScript খুবই lightweight এবং browser এর মাধ্যমে সরাসরি interpret হয়। অর্থাৎ, এটি রান করার জন্য আলাদা করে compile করার প্রয়োজন হয় না।
2. **Cross-platform Compatibility**: JavaScript সকল আধুনিক browser এ সমর্থিত এবং এটি বিভিন্ন প্ল্যাটফর্মে কাজ করতে পারে।
3. **Dynamic Typing**: JavaScript এ ভেরিয়েবলের ডেটা টাইপ পরিবর্তনযোগ্য এবং এটি dynamic ভাবে কাজ করে।
4. **Prototype-based Object Orientation**: JavaScript প্রোটোটাইপ-ভিত্তিক অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং সমর্থন করে।
5. **First-class Functions**: JavaScript এ ফাংশনকে ভেরিয়েবল হিসেবে রাখা, আর্গুমেন্ট হিসেবে পাস করা এবং রিটার্ন করা যায়।

🛠️ Uses of JavaScript

JavaScript এর ব্যবহার অনেক বিস্তৃত এবং এটি web ডেভেলপমেন্টের বিভিন্ন ক্ষেত্রে ব্যবহৃত হয়। নিচে JavaScript এর কিছু সাধারণ ব্যবহার উল্লেখ করা হলোঃ

- **Web Development**: web পেজে ইন্টারেক্টিভ ফিচার যোগ করতে যেমন form validation, dynamic কন্টেন্ট আপডেট ইত্যাদি।
- **Server-side Programming**: **Node.js** ব্যবহার করে JavaScript server-side প্রোগ্রামিংয়ের জন্যও ব্যবহৃত হয়।
- **Mobile App Development**: **React Native** এর মত framework ব্যবহার করে mobile app তৈরি করা যায়।
- **Game Development**: JavaScript এর মাধ্যমে browser ভিত্তিক game তৈরি করা সম্ভব।

📝 JavaScript Example

নিচে একটি সহজ উদাহরণ দেওয়া হলো যেখানে JavaScript এর মাধ্যমে একটি HTML element এর কন্টেন্ট পরিবর্তন করা হয়েছে:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>JavaScript Example</title>
  </head>
  <body>
    <h1 id="myHeading">👋 Hello, World!</h1>
    <button onclick="changeContent()">Change Content</button>

    <script>
      function changeContent() {
        document.getElementById("myHeading").innerHTML = "Content Changed!";
      }
  </script>
 </body>
</html>
```

উপরের কোডটিতে, একটি **button** এ click করার মাধ্যমে **h1** element এর টেক্সট পরিবর্তন করা হয়েছে।

Output

উপরের কোডটি একটি browser এ রান করার পরে, আপনি দেখতে পাবেন একটি heading যেখানে লেখা থাকবে "👋 Hello, World!" এবং নিচে একটি button থাকবে "Change Content" নামে। যখন আপনি button এ click করবেন, তখন heading এর টেক্সট পরিবর্তিত হয়ে "Content Changed!" হয়ে যাবে।

ব্যাখ্যা

1. **HTML Structure**: উপরের কোডে HTML এর মাধ্যমে পেজের স্ট্রাকচার তৈরি করা হয়েছে।
2. **JavaScript Function**: changeContent() ফাংশনটি ব্যবহার করে আমরা myHeading element এর content পরিবর্তন করেছি।
3. **Event Handling**: button এ click করার মাধ্যমে JavaScript এর function কল করা হয়েছে, যা নির্দিষ্ট element এর content পরিবর্তন করেছে।

🚀 Single Threaded vs Multi-Threaded Language

JavaScript একটি **Single Threaded** প্রোগ্রামিং ভাষা, যার মানে এটি একই সময়ে একটি কাজ করতে পারে। **Single Thread** মানে একটি থ্রেড বা ধারা আছে যা একবারে একটি কাজ করে। এক্ষেত্রে JavaScript এর **call stack** এবং **event loop** এর ব্যবহার করা হয়।

Single Threaded

Single Threaded মানে হচ্ছে একই সময়ে শুধুমাত্র একটি কাজ সম্পন্ন করা। JavaScript এ সব কাজ একটিমাত্র থ্রেডে সম্পন্ন হয়। অর্থাৎ, JavaScript এর **call stack** এ এক সময়ে একটিমাত্র ফাংশন বা টাস্কই এক্সিকিউট হয়। এর ফলে JavaScript এর কোডগুলো সিকোয়েন্স আকারে রান করে এবং পরবর্তী কোড রান করার আগে বর্তমান কোডের এক্সিকিউশন শেষ করতে হয়।

উদাহরণ (Single Threaded):

```
function taskOne() {
  console.log("Task One is starting...");
  for (let i = 0; i < 1000000000; i++) {
    // Simulating a time-consuming task
  }
  console.log("Task One is completed.");
}

function taskTwo() {
  console.log("Task Two is starting...");
}

taskOne();
taskTwo();
```

Output:

```
Task One is starting...
(Task One is completed after a long delay)
Task Two is starting...
```

ব্যাখ্যা:

উপরের উদাহরণে, প্রথমে taskOne() ফাংশনটি কল করা হয়। এটি একটি লম্বা সময় নেওয়া লুপ সম্পন্ন করে, এরপর taskTwo() ফাংশনটি চালানো হয়। যেহেতু JavaScript একটি single-threaded ভাষা, তাই taskOne() সম্পন্ন না হওয়া পর্যন্ত taskTwo() শুরু হতে পারে না।

Multi-Threaded

Multi-Threaded প্রোগ্রামিং ভাষাগুলোর ক্ষেত্রে একই সাথে একাধিক কাজ করা যায়। এক্ষেত্রে প্রতিটি কাজ আলাদা থ্রেডে চালানো হয়। যেমন, একটি থ্রেড ইন্টারফেস নিয়ে কাজ করছে, অন্য একটি থ্রেড ডেটা প্রসেস করছে। এর ফলে একই সময়ে একাধিক কাজ চালানো সম্ভব হয়, যা অ্যাপ্লিকেশনকে আরও দ্রুত ও কার্যকর করে তোলে।

উদাহরণ (Multi-Threaded):

ধরা যাক আমরা একটি ভাষা ব্যবহার করছি যেখানে multi-threading সমর্থন করে। নিচের উদাহরণটি একটি সাধারণ multi-threaded প্রোগ্রামের ধারণা দেয়:

```
class TaskOne extends Thread {
    public void run() {
        System.out.println("Task One is starting...");
        for (int i = 0; i < 1000000000; i++) {
            // Simulating a time-consuming task
        }
        System.out.println("Task One is completed.");
    }
}

class TaskTwo extends Thread {
    public void run() {
        System.out.println("Task Two is starting...");
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        TaskOne t1 = new TaskOne();
        TaskTwo t2 = new TaskTwo();
        t1.start();
        t2.start();
    }
}
```

Output:

```
Task One is starting...
Task Two is starting...
(Task One is completed after a long delay)
```

ব্যাখ্যা:

উপরের উদাহরণে, TaskOne এবং TaskTwo দুটি আলাদা থ্রেডে এক্সিকিউট করা হয়। এর ফলে TaskTwo ফাংশনটি TaskOne এর সম্পূর্ণ হওয়ার জন্য অপেক্ষা না করে সাথে সাথেই শুরু হতে পারে। এটি multi-threading এর সুবিধা, যা একই সময়ে একাধিক কাজ করতে দেয়।

JavaScript কেন Single Threaded?

JavaScript মূলত **Single Threaded** কারণ এটি প্রথমে তৈরি হয়েছিল **browser** এর জন্য, যেখানে ব্যবহারকারীর সাথে সরাসরি ইন্টারঅ্যাকশন থাকে। Single Threaded ডিজাইনের ফলে JavaScript সহজভাবে ব্যবহারকারীর সাথে ইন্টারঅ্যাক্ট করতে পারে এবং কোনও ফাংশন এক্সিকিউশন চলাকালীন অন্য কোনও কাজ বাধা সৃষ্টি করে না।

যখন একটি কাজ JavaScript এ শুরু হয়, তখন তা **call stack** এ যোগ করা হয় এবং এক্সিকিউট হয়। যদি কোনও ফাংশন এক্সিকিউশন লম্বা সময় নেয়, তবে পরবর্তী কাজগুলি অপেক্ষা করে, যা **blocking** বলে। তবে **asynchronous** কার্যপ্রণালী যেমন **callback**, **promises**, এবং **async/await** এর মাধ্যমে JavaScript কিছুটা **non-blocking** কাজ করতে পারে।

JavaScript একটি **Single Threaded** প্রোগ্রামিং ভাষা, যার মানে এটি একই সময়ে একটি কাজ করতে পারে। **Single Thread** মানে একটি থ্রেড বা ধারা আছে যা একবারে একটি কাজ করে। এক্ষেত্রে JavaScript এর **call stack** এবং **event loop** এর ব্যবহার করা হয়।

🔚 Conclusion

JavaScript একটি শক্তিশালী এবং জনপ্রিয় প্রোগ্রামিং ভাষা যা web ডেভেলপমেন্টে dynamic এবং ইন্টারেক্টিভ ফিচার যোগ করতে ব্যবহৃত হয়। এর মাধ্যমে আপনি web পেজে প্রাণ এনে দিতে পারবেন এবং ব্যবহারকারীদের আরও ইন্টারেক্টিভ অভিজ্ঞতা দিতে পারবেন। JavaScript শেখার মাধ্যমে web ডেভেলপার হিসেবে আপনার দক্ষতা অনেকগুণ বৃদ্ধি পাবে।

↥ Go to Top

Chapter-02: JS with HTML, JS Output, Installing Node, Variable, Data Types and Function

HTML Document এ কোথায় JavaScript কোড লিখতে হয়?

৩ জায়গায় লিখা যায়ঃ

- Body এর মধ্যে।

- Head এর মধ্যে।
- আলাদা ফাইলে। যেমনঃ script.js

JavaScript Can Change The Content of HTML Element

- getElementById বা getElementByClass methods ব্যবহার করে আমরা HTML Element ধরে তার Content পরিবর্তন করে দিতে পারি। তার জন্য আমাদের innerHTML property ব্যবহার করতে হয়। যেমনঃ

```
document.getElementById("codejogot").innerHTML = "Hello JavaScript";
```

JavaScript Can Change The Value of An Attribute

- getElementById বা getElementByClass এর পরে Attribute এর নাম লিখে আমরা চাইলে Attribute এর Value পরিবর্তন করতে পারি। যেমনঃ

```
<button onclick="document.getElementById('myImage').src='pic_bulbon.gif'">
  Change the photo
</button>
```

JavaScript Can Change The CSS Style

যেমনঃ

```
document.getElementById("demo").style.fontSize = "35px";
```

JavaScript Variables

- Variable হলো Data Store করার জন্য Container.
- ৪ ভাবে JavaScript Variable Declare করা যায়ঃ
    - Automatically
    - let keyword
    - var keyword
    - const keyword

Example of Automatic Declaration:

```
x = 5;
y = 6;
z = x + y;
```

Note: Variable Declare করা Good Practice.

```
var x = 5;
```

var y = 6;

var z = x + y;

What is Node

- Node হল একটি **open-source**, **cross-platform**, JavaScript **runtime environment** যা ওয়েব ব্রাউজারের বাইরে জাভাস্ক্রিপ্ট কোড Run করে। আমরা জানি JavaScript Code কেবল Web Browser এই রান করতে পারে। আমরা যাতে Web Browser এর বাইরেও জাভাস্ক্রিপ্ট কোড Run করতে পারি সেজন্যই মূলত Node এর আগমন।
- Node হলো Google Chrome এর V8 Engine দ্বারা তৈরি।
- Node.js এর মাধ্যমে ডেভেলপাররা জাভাস্ক্রিপ্ট ব্যবহার করে সার্ভার-সাইড স্ক্রিপ্ট লিখতে পারে।

Features of Node

- **Asynchronous and Event-Driven**
  - **Non-blocking I/O:** Node.js uses non-blocking, event-driven architecture, making it efficient and suitable for real-time applications. Non-blocking I/O means that Node.js can handle many operations simultaneously without waiting for any single operation to complete.
  - **Event Loop:** Node.js operates on a single-threaded event loop, allowing it to handle multiple connections concurrently. This is particularly useful for I/O-heavy operations.
- **Single Programming Language**
  - With Node.js, developers can use JavaScript for both client-side and server-side programming. This unification of language reduces the learning curve and allows for code reuse across the stack.
- **Package Management with npm**
  - Node.js comes with npm (Node Package Manager), a package manager that provides access to a large ecosystem of reusable libraries and tools. npm makes it easy to manage dependencies and share code with other developers.
- **Scalability**
  - Node.js is designed to build scalable network applications. Its non-blocking I/O and event-driven architecture allow it to handle many concurrent connections with minimal overhead.
- **Performance**

- Built on the V8 JavaScript engine, Node.js provides high performance and fast execution of JavaScript code. V8 compiles JavaScript into native machine code, optimizing it for speed.

Use Cases for Node.js

- **Web Servers:** Node.js is commonly used to build web servers that can handle HTTP requests. It is especially suitable for building RESTful APIs and real-time web applications.
- **Real-Time Applications:** Applications that require real-time communication, such as chat applications, gaming servers, and collaborative tools, benefit from Node.js's event-driven architecture.
- **Single Page Applications (SPAs):** Node.js is often used in conjunction with front-end frameworks (like Angular, React, or Vue.js) to build SPAs, where the application logic is handled on the client-side and the server provides the necessary data via APIs.
- **Command-Line Tools:** Node.js can be used to create command-line tools and scripts that automate tasks, manage systems, and process data.
- **Microservices:** Node.js is well-suited for building microservices architectures due to its lightweight nature and efficient handling of concurrent requests.

Example: Building a Simple Web Server with Node.js

```
// Load the http module to create an HTTP server.
const http = require("http");

// Configure the HTTP server to respond with "Hello, World!" to all requests.
const server = http.createServer((req, res) => {
  // Set the response HTTP header with HTTP status and Content type
  res.writeHead(200, { "Content-Type": "text/plain" });
  // Send the response body "Hello, World!"
  res.end("Hello, World!\n");
});

// Listen on port 3000 and IP address 127.0.0.1
server.listen(3000, "127.0.0.1", () => {
  console.log("Server running at http://127.0.0.1:3000/");
```

});

Difference Between Var, Let and Const

Criteria	**Var**	**Let**	**Const**
Scope	Function-scoped	Block-scoped	Block-scoped
Re-declaration	Re-declare করা যায় এবং Value Update করা যায়	Re-declare করা যায় না কিন্তু Value Update করা যায়। Let অনেকটা C++ এর Variable এর মতো কাজ করে	Re-declare করাও যায় না, Value-ও Update করা যায় না

|

JavaScript Data Types

- JavaScript এ ৮ ধরনের Data Type আছেঃ
    - String
    - Number
    - Bigint
    - Boolean
    - Undefined
    - Null
    - Symbol
    - Object

Examples:

// Numbers:

let length = 16;

let weight = 7.5;

// Strings:

let color = "Yellow";

let lastName = "Johnson";

```
// Booleans
let x = true;
let y = false;

// Object:
const person = { firstName: "John", lastName: "Doe" };

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");
```

- JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
let x = 16 + 4 + "CodeJogot";
Output: 20CodeJogot
let x = "Volvo" + 16 + 4;
Output: Volvo164
```

JavaScript Types are Dynamic

- JavaScript has dynamic types. This means that the same variable can be used to hold different data types.

```
let x; // Now x is undefined
x = 5; // Now x is a Number
x = "John"; // Now x is a String
```

↥ Go to Top

Chapter-02.1: JavaScript Var, Let & Const

JavaScript Var, Let & Const

Table of Contents:

---

1. Introduction to JavaScript Variables

JavaScript এ variables হলো memory locations যেখানে বিভিন্ন data store করা যায়। Variables declare করতে হলে **var**, **let**, এবং **const** keywords ব্যবহার করা হয়। এই তিনটি keyword JavaScript এ variables declare করার বিভিন্ন ধরণ এবং behavior নিয়ে কাজ করে।

---

2. Var

Scope of Var

Var হলো JavaScript এর পুরনো variable declaration method, যা function scope এ কাজ করে। এর মানে হলো, একটি function এর ভেতরে **var** দিয়ে declare করা variable সেই function এর scope এর ভেতরেই accessible। তবে, **var** block scope এ কাজ করে না, অর্থাৎ loop বা conditionals এর মধ্যে declare করা variable বাইরে থেকেও accessible থাকে।

Example:

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

- এখানে **x** variable block এর মধ্যে declare করা হলেও, এটি block এর বাইরে accessible হচ্ছে কারণ **var** block scope maintain করে না।

Hoisting with Var

JavaScript এ **hoisting** এর কারণে **var** দিয়ে declare করা variable এবং function গুলো code এর শুরুতে move হয়। অর্থাৎ, variable declaration top এ চলে আসে কিন্তু তার initialization হয় না।

Example:

```
console.log(a); // Output: undefined
var a = 5;
```

- উপরের উদাহরণে, **a** variable hoist হয়ে যায়, তাই console এ **undefined** দেখায়। তবে, **a** এর মান 5 assign হওয়ার আগেই এটি access করা হয়েছে।

---

3. Let

Scope of Let

Let ES6 (ECMAScript 2015) এ introduce হয় এবং এটি **block-scoped**। অর্থাৎ, একটি block এর মধ্যে **let** দিয়ে declare করা variable সেই block এর বাইরে accessible নয়। এটি **var** এর মতো function scope না মেনে, block scope মেনে চলে।

Example:

```
if (true) {
  let y = 20;
}
console.log(y); // Output: ReferenceError: y is not defined
```

- এখানে, **y** variable block এর মধ্যে declare করা হয়েছে এবং block এর বাইরে এটি accessible নয়। তাই **ReferenceError** দেখায়।

Temporal Dead Zone (TDZ)

Let দিয়ে declare করা variable গুলোতে hoisting হয়, তবে তারা **Temporal Dead Zone** (TDZ) এর কারণে initialization এর আগে access করা যায় না।

Example:

```
console.log(b); // Output: ReferenceError: Cannot access 'b' before initialization
let b = 10;
```

- এখানে **b** variable initialization এর আগে access করা হয়েছে, তাই **ReferenceError** দেখা যায়।

4. Const

Scope and Mutability of Const

Const হলো constant value declare করার জন্য ব্যবহৃত keyword। এটি **let** এর মতো **block-scoped**, তবে এর বিশেষত্ব হলো, **const** দিয়ে declare করা variable এর মান পরবর্তীতে change করা যায় না।

Example:

const z = 30;

z = 40; // Output: TypeError: Assignment to constant variable

- এখানে **z** variable এর মান পরিবর্তন করার চেষ্টা করলে **TypeError** দেখায়।

তবে, **const** দিয়ে object বা array declare করলে, তাদের properties বা elements পরিবর্তন করা যায়। শুধুমাত্র reference (variable এর memory location) অপরিবর্তনীয় থাকে।

Example:

const person = { name: "John" };

person.name = "Doe"; // This is allowed

console.log(person.name); // Output: "Doe"

- **person** object এর property পরিবর্তন করা হয়েছে, যা **const** variable হলেও সম্ভব, কারণ reference একই থাকে।

---

5. Differences Between Var, Let, and Const

Feature	Var	Let	Const
Scope	Function scope	Block scope	Block scope
Hoisting	Hoisted, initialized as undefined	Hoisted, but not initialized	Hoisted, but not initialized
Reassignable	Yes	Yes	No
Re-declaration	Allowed (even within same scope)	Not allowed (within same scope)	Not allowed (within same scope)

Explanation:

1. **Scope**: **Var** function scoped হলেও, **let** এবং **const** block scoped।
2. **Hoisting**: তিনটি keyword ই hoisting করে, কিন্তু **let** এবং **const** TDZ এর কারণে initialization এর আগে access করা যায় না।
3. **Reassignable**: **var** এবং **let** দিয়ে declare করা variables এর মান পরিবর্তন করা যায়, কিন্তু **const** এর মান অপরিবর্তনীয়।
4. **Re-declaration**: **var** একই scope এর মধ্যে বারবার declare করা যায়, কিন্তু **let** এবং **const** একই scope এ একবারই declare করা যায়।

---

6. Best Practices for Using Var, Let, and Const

1. **Use let for variables that may change**: যদি কোনো variable এর মান পরিবর্তন হতে পারে, তবে **let** ব্যবহার করুন। এটি block scope এ থাকে এবং **var** এর মতো hoisting issue তৈরি করে না।
2. **Use const for constant values**: যদি কোনো variable এর মান কখনো পরিবর্তন হবে না, তাহলে **const** ব্যবহার করুন। এটি প্রোগ্রামের readability এবং predictability বাড়ায়।
3. **Avoid using var**: Modern JavaScript development এ **var** ব্যবহার এড়ানো ভালো, কারণ এটি hoisting এবং block scope এর অভাবে bugs তৈরি করতে পারে।

---

7. Conclusion

JavaScript এর **var**, **let**, এবং **const** keyword গুলো variable declare করার জন্য ব্যবহৃত হয়, তবে এদের behavior এবং scope ভিন্ন। **Var** হলো function scoped এবং hoisting এর issue রয়েছে। **Let** হলো block scoped এবং TDZ নিশ্চিত করে। **Const** হলো block scoped এবং constant value declare করার জন্য ব্যবহৃত হয়।

Modern JavaScript development এ **let** এবং **const** বেশি ব্যবহার করা হয়, কারণ এগুলো **var** এর limitations দূর করে এবং কোডকে আরও predictable করে তোলে। Proper use of these keywords ensures cleaner, safer, and more maintainable code.

↥ Go to Top

Chapter-03: JS Operators, Arithmetic, Data Types & Js Functions

- Comparison Operators
- JavaScript String Addition
- Adding Strings and Numbers

JS Operators

- জাভাস্ক্রিপ্ট অপারেটরগুলি বিভিন্ন ধরণের Mathematical এবং Logical Computatioin করতে ব্যবহৃত হয়।

JavaScript Assignment Operator

- Assignment Operator (=) একটি ভেরিয়েবলের জন্য একটি মান নির্ধারণ করে।

Example:

// Assign the value 5 to x

let x = 5;

// Assign the value 2 to y

let y = 2;

// Assign the value x + y to z:

let z = x + y;

JavaScript Addition Operator

- Addition Operator দিয়ে Number যোগ করা হয়।

Example:

let x = 5;

let y = 2;

let z = x + y;

JavaScript Multiplication Operator

- Multiplication Operator দিয়ে Numbers গুন করা হয়।

Example:

let x = 5;

let y = 2;

let z = x * y;

Types of JavaScript Operators

1. Arithmetic Operators
2. Assignment Operators

3. Comparison Operators
4. Logical Operators
5. String Operators
6. Conditional (Ternary) Operator
7. Bitwise Operator
8. Type Operator

Arithmetic Operator

Example:

```
let a = 3;

let x = (100 + 50) * a;
```

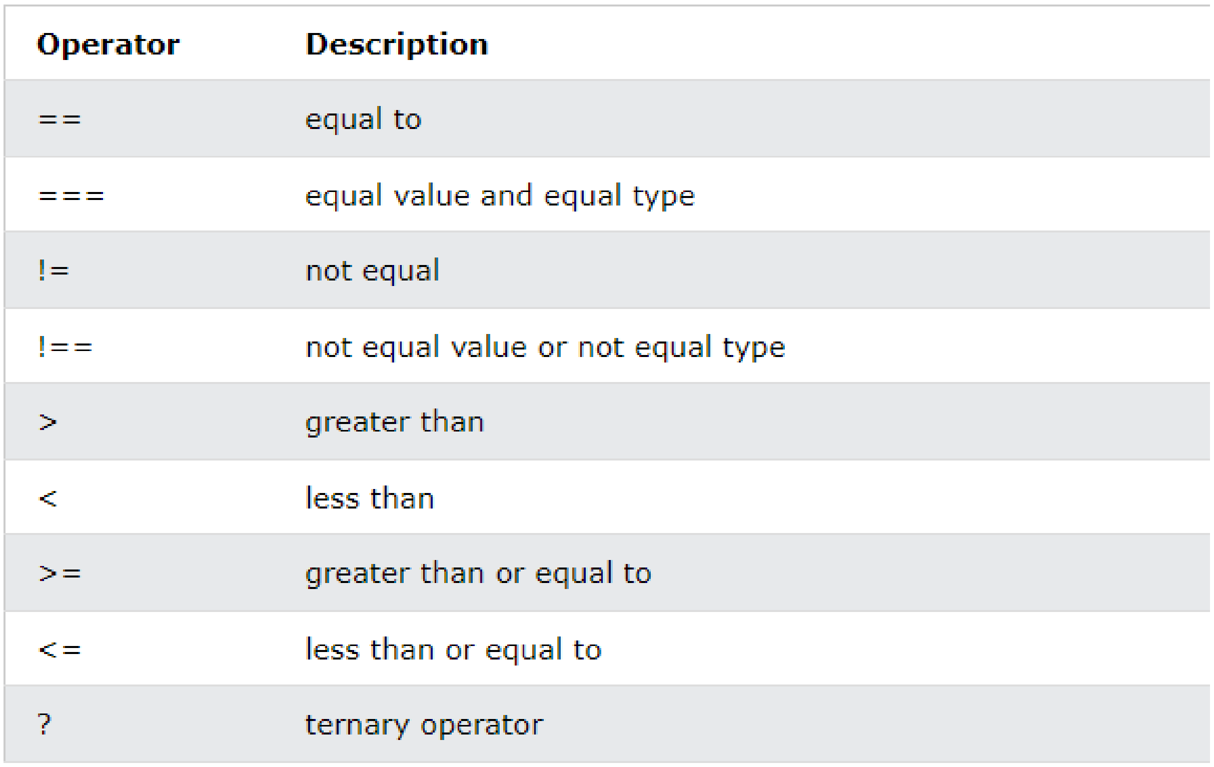
Assignment Operator

Example:

```
let x = 10;

x += 5;
```

Comparison Operators



Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

Credit: W3 School

JavaScript String Addition

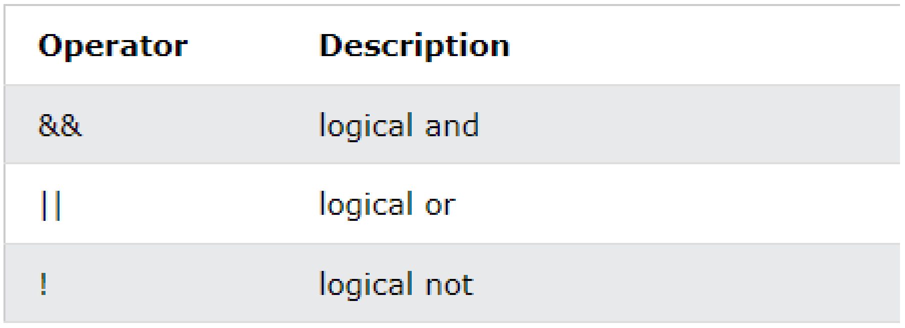
```
let text1 = "John";
let text2 = "Doe";
let text3 = text1 + " " + text2;
```

Adding Strings and Numbers

```
let x = 5 + 5;
let y = "5" + 5;
let z = "Hello" + 5;
```

Output: 10 55 Hello5

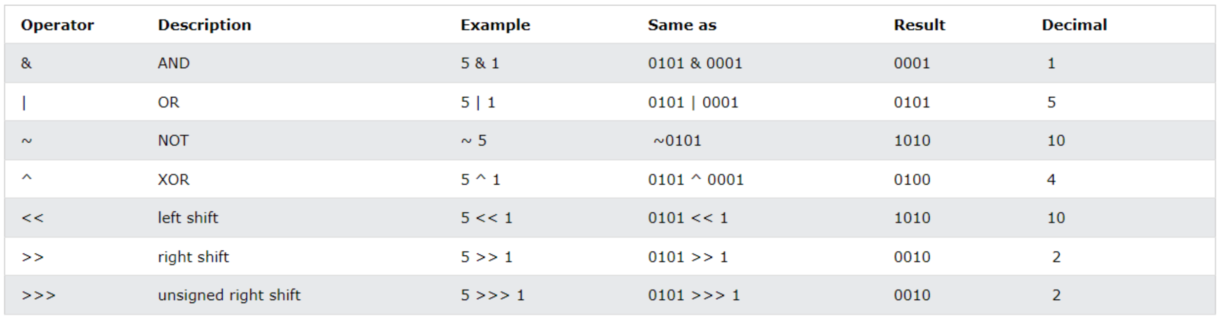
JavaScript Logical Operators



Operator	Description
&&	logical and
||	logical or
!	logical not

JavaScript Bitwise Operators

- Bit operators work on 32 bits numbers.
- Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.



Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	left shift	5 << 1	0101 << 1	1010	10
>>	right shift	5 >> 1	0101 >> 1	0010	2
>>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

Image Credit: W3 School

JavaScript Functions

- JavaScript Function তৈরি করতে গেলে প্রথমে function keyword লিখতে হবে, এরপর Function এর একটা নাম দিতে হবে এবং সবশেষে ()
- Syntax:

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

JavaScript Function যেভাবে Call করা হয়

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function Return Mechanism

```
// Function is called, the return value will end up in x
let x = myFunction(4, 3);

function myFunction(a, b) {
  // Function returns the product of a and b
  return a * b;
}
```

কেন আমরা Function ব্যবহার করি?

- একই কাজ বার বার করাকে Avoid করার জন্য।

How to Handle Standard Input and Output in VS Code for JavaScript

Example:

```
const readline = require("readline");

// Create an interface for reading input and output
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

// Prompt the user for input
rl.question("Enter numbers separated by spaces: ", (input) => {
  // Process the input
  const numbers = input.split(" ").map(Number);

  // For example, using the countEvenNumbers function from earlier
  function countEvenNumbers(arr) {
    let count = 0;
    for (let i = 0; i < arr.length; i++) {
      if (arr[i] % 2 === 0) {
        count++;
      }
    }
    return count;
  }

  const evenCount = countEvenNumbers(numbers);
  console.log(`Number of even numbers: ${evenCount}`);

  // Close the input interface
  rl.close();
});
```

In Console Write:

```
node main.js
```

Input:

```
Enter numbers separated by spaces: 1 2 3 4 5
```

Output:

```
Number of even numbers: 2
```

Solving Some Easy Problems

Problem-01: Write a JavaScript function to find the sum of two numbers.

Problem-02: Write a JavaScript function to find the product of two numbers.

Problem-03: Write a JavaScript function to find the difference of two numbers.

Problem-04: Write a JavaScript function to find the remainder.

Problem-05: Write a JavaScript function to check if a number is positive, negative or zero.

Problem-06: Write a JavaScript function to check if a number is odd or even.

Problem-07: Write a JavaScript function to calculate the square of a number.

Problem-08: Write a JavaScript function to concatenate two strings.

Problem-09: Write a JavaScript function to find the biggest number among 3 numbers.

Problem-10: Write a JavaScript function which returns the number of even numbers in an array.

↥ Go to Top

Chapter-04: JavaScript Object

What is Object

- অবজেক্ট, জাভাস্ক্রিপ্টে, সবচেয়ে গুরুত্বপূর্ণ ডেটা টাইপ এবং আধুনিক জাভাস্ক্রিপ্টের জন্য বিল্ডিং ব্লক তৈরি করে। এই অবজেক্টগুলি জাভাস্ক্রিপ্টের Primitive Data Type(Number, String, Boolean, null, undefined, and symbol) থেকে বেশ ভিন্ন এই অর্থে যে এই Primitive Data Type কেবল একটি Value Store করতে পারে। যেখানে Object একাধিক Value Store করতে পারে। উদাহরনঃ

```
// Create an Object
const student = {
  firstName: "Abdur",
  lastName: "Rahman",
  batch: 5,
  id: "WDB05027",
  marks: 97,
};
```

- In JavaScript, objects are collections of data and functions. This data is stored in the form of **key-value** pairs.
- Keys that store data values are called properties.
- Keys that store functions are called methods.
- It is a common practice to declare objects with the **const** keyword.
- Dot(.) দিয়ে Object এর **Property/Methods** এর Value কে Access করা যায়। যেমনঃ

```
console.log(student.firstName);
```

- Bracket([]) দিয়েও Object এর **Property/Methods** এর Value কে Access করা যায়। যেমনঃ

```
console.log(student["firstName"]);
```

- Object এ Property হিসেবে Function-ও ব্যবহার করা যায়। যেমনঃ

```
// Create an Object
const student = {
  firstName: "Abdur",
  lastName: "Rahman",
  batch: 5,
  id: "WDB05027",
  marks: 97,
  isPassed: function () {
    if (this.marks >= 33) {
      return `${firstName} ${lastName} is passed with ${marks} marks`;
    } else return `${firstName} ${lastName} is failed with ${marks} marks`;
  },
};
//calling function from object
let output = student.isPassed();
```

- এই উদাহরনে *this* keyword student object কে রেফার করছে।
- জাভাস্ক্রিপ্টে একটি অবজেক্ট একটি রেফারেন্স ডেটা টাইপ হিসাবে বিবেচিত হয়। এর মানে হল যে আপনি যখন একটি অবজেক্ট তৈরি করেন এবং এটিকে একটি ভেরিয়েবলে অ্যাসাইন করেন, তখন ভেরিয়েবলটি আসলে অবজেক্টটিকে ধরে রাখে না। Instead, it holds a reference (or pointer) to the location in memory where the object is stored. Memory এর Location Point করে বলে একে Pointer বলে।

উদাহরনঃ

```
let x = { name: "Alice" };
```

Here, **x** is a variable that holds a reference (or pointer or memory address) to the memory location where the object { name: "Alice" } is stored.

- In JavaScript, Objects are King. If you Understand Objects, you Understand JavaScript.
- In JavaScript, **Properties** can be primitive values, functions, or even other objects.
- জাভাস্ক্রিপ্টে, প্রায় "সবকিছুই" একটি Object.
    - Objects are objects
    - Arrays are objects
    - Functions are objects
    - Dates are objects
    - Maths are objects
    - Sets are objects
    - All JavaScript values, except primitives, are objects.

JavaScript Primitives

- A primitive value is a value that has no properties or methods. Example:

let greeting = "Hello";

এখানে "Hello" হলো Primitive Value এবং greeting হলো Primitive Data Type.

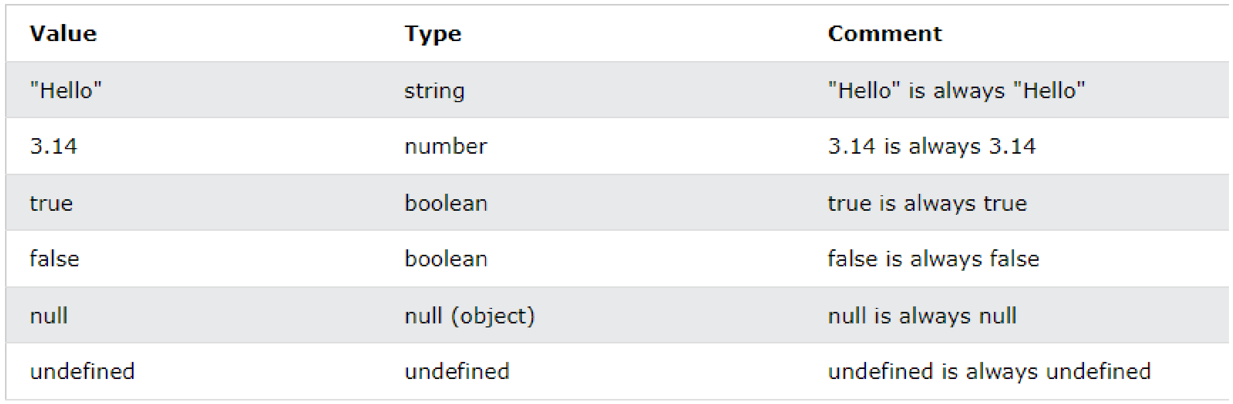
- যেসব Data Type এ কেবল Primitive Value থাকে, তাকে Primitive Data Type বলে। যেমনঃ
    - string
    - number
    - boolean
    - null
    - undefined
    - symbol
    - bigint

Immutable

- Primitive values are immutable অর্থাৎ Primitive Value কে পরিবর্তন করা যায় না। যেমনঃ

let x = 100;

এখানে আপনি x এর Value পরিবর্তন করতে পারবেন, কিন্তু 100 কে পরিবর্তন করতে পারবেন না। 100 Always 100 ই থাকবে।



Value	Type	Comment
"Hello"	string	"Hello" is always "Hello"
3.14	number	3.14 is always 3.14
true	boolean	true is always true
false	boolean	false is always false
null	null (object)	null is always null
undefined	undefined	undefined is always undefined

JavaScript Objects are Mutable

- Objects are mutable: They are addressed by reference, not by value.
- যদি **student** একটি Object হয়, এবং let x = student লিখা হয়, তাহলে object x is **not a copy** of **student**. The object **x** is **student**. Object x এবং object student একই Memory Share করে। তাই যদি কোন কারনে Object x পরিবর্তন করা হয়, তাহলে Object student-ও পরিবর্তন হয়ে যাবে।

```
/Create an Object
const student = {
  firstName:"John",
  lastName:"Doe",
  id:20
}

// Create a copy
const x = student;

// Change id in both
x.age = 10;
```

JavaScript Object Properties

- Properties can be changed, added, deleted, and some are read only.

Adding New Properties

```
// Define the student object
const student = {
  // Properties
  firstName: "Abdur",
  lastName: "Rahim",
  age: 21,
  major: "Computer Science",

  // Method
  getFullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },
};

// Add a new property
student.email = "abdurrahim@gmail.com";
```

Deleting Properties

```
delete student.major;
or;
delete student["major"];
```

Nested Object

```
// Define the student object with nested objects
const student = {
  // Properties
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",

  // Nested object for contact information
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },

  // Nested object for address
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },

  // Method
  getFullName: function () {
    return `${this.firstName} ${this.lastName}`;
  },

  // Method to get full address
  getFullAddress: function () {
    return `${this.address.street}, ${this.address.city}, ${this.address.state}
${this.address.zip}`;
  },
};

// Example usage of the object and its methods
console.log(student.getFullName()); // Output: John Doe
console.log(student.getFullAddress()); // Output: 123 Main St, Anytown, Anystate 12345
console.log(student.contactInfo.email); // Output: john.doe@example.com
console.log(student.contactInfo.phone); // Output: 123-456-7890
```

Looping through an object in JavaScript

- Requirement এর উপর ভিত্তি করে JavaScript এ চারভাবে Object এ Loop করা যায়।

1. Looping through an object with for...in

```
const student = {
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },
};

for (let key in student) {
  if (student.hasOwnProperty(key)) {
    console.log(`${key}: ${student[key]}`);
  }
}
```

Output:

```
firstName: John
lastName
firstName: John
```

```
lastName: Doe
age: 21
major: Computer Science
contactInfo: [object Object]
address: [object Object]
```

2. Looping through an object with forEach

```
const student = {
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },
};

Object.keys(student).forEach((key) => {
  console.log(`${key}: ${student[key]}`);
});
```

Output:

```
firstName: John
lastName: Doe
age: 21
```

```
major: Computer Science
contactInfo: [object Object]
address: [object Object]
```

3. Looping through an object Using Object.entries() with for...of

```
const student = {
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },
};

for (let [key, value] of Object.entries(student)) {
  console.log(`${key}: ${value}`);
}
```

Output:

```
firstName: John
lastName: Doe
age: 21
major: Computer Science
contactInfo: [object Object]
```

```
address: [object Object]
```

4. Looping through an object Using Object.values() with forEach()

- Object.values() creates an array from the property values:

```
const student = {
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },
};

Object.values(student).forEach((value) => {
  console.log(value);
});
```

Output:

```
John
Doe
21
Computer Science
[object Object]
[object Object]
```

What is JSON

- JSON (JavaScript Object Notation) হল একটি লাইটওয়েট **Data Interchange Format** যা মানুষের পক্ষে পড়তে এবং লিখতে সহজ এবং মেশিন সহজে এটিকে Parse করতে পারে।
- JSON একটি Lanuage-independant Format.
- JSON এর key হলো একটি String এবং Value হলো যেকোনো Valid Data Type (String, Array, Number, Boolen etc)
- ওয়েব অ্যাপ্লিকেশনে সার্ভার এবং ক্লায়েন্টের মধ্যে ডেটা প্রেরণের জন্য JSON ব্যাপকভাবে ব্যবহৃত হয়।

Example of JSON:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21,
  "major": "Computer Science",
  "contactInfo": {
    "email": "john.doe@example.com",
    "phone": "123-456-7890"
  },
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "Anystate",
    "zip": "12345"
  },
  "courses": ["CS101", "CS102", "CS103"],
  "graduated": false
}
```

JSON এর ব্যবহার

- **Web APIs:** JSON হল ওয়েব API-এর জন্য সবচেয়ে সাধারণ ফর্ম্যাট। সার্ভার এবং ক্লায়েন্টের মধ্যে ডেটা আদান-প্রদানের একটি সহজ উপায়।

- **Configuration Files:** Many applications use JSON for configuration files due to its readability and ease of use.
- **Data Storage:** Some databases, such as MongoDB, store data in JSON-like formats.

Converting Object to JSON String

- JavaScript এ, Object থেকে JSON এ Convert করার জন্য JSON.stringify() এবং JSON থেকে Object করার জন্য JSON.parse() ব্যবহার করা হয়।

```
// Define a JavaScript object
const student = {
  firstName: "John",
  lastName: "Doe",
  age: 21,
  major: "Computer Science",
  contactInfo: {
    email: "john.doe@example.com",
    phone: "123-456-7890",
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "Anystate",
    zip: "12345",
  },
};

// Convert the JavaScript object to a JSON string
const jsonString = JSON.stringify(student);

// Output the JSON string
console.log(jsonString);
```

Output:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21,
  "major": "Computer Science",
  "contactInfo": { "email": "john.doe@example.com", "phone": "123-456-7890" },
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "Anystate",
    "zip": "12345"
  }
}
```

More Readable:

- To make the JSON string more readable, you can pass additional arguments to JSON.stringify() to include indentation.

```
// Convert the JavaScript object to a pretty-printed JSON string
const prettyJsonString = JSON.stringify(student, null, 2);

// Output the pretty-printed JSON string
console.log(prettyJsonString);
```

Output:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21,
  "major": "Computer Science",
  "contactInfo": {
    "email": "john.doe@example.com",
    "phone": "123-456-7890"
  },
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "Anystate",
    "zip": "12345"
  }
}
```

- The third argument (2) specifies the number of spaces to use as white space for indentation, making the JSON string more readable.

JavaScript Constructor Functions

- অনেক সময় একই Type এর একাধিক Object আমাদের তৈরি করা লাগতে পারে। যেমনঃ

Batch-05 এর Al Amin Student এর জন্য আমরা একটা Object তৈরি করতে পারি।

```
let student = {
  // Properties
  firstName: "Al",
  lastName: "Amin",
  id: "WD05020",
  batch: 5,

  // Method
  getDetails: function () {
    return `Name: ${this.firstName} ${this.lastName}, ID: ${this.id}, Batch: ${this.batch}`;
  },
};
console.log(student.getDetails());
```

আবার Batch-05 এর Sujon Rana এর জন্য আমরা একটা Object তৈরি করতে পারি।

```
let student = {
  // Properties
  firstName: "Sujon",
  lastName: "Rana",
  id: "WD05025",
  batch: 5,

  // Method
  getDetails: function () {
    return `Name: ${this.firstName} ${this.lastName}, ID: ${this.id}, Batch: ${this.batch}`;
  },
};
console.log(student.getDetails());
```

- আপনাদের মনে আছে কি, আমাদের যদি একই Type এর অনেকগুলো Variable Declare করার প্রয়োজন হয়, তাহলে আমরা Array তৈরি করি। একইভাবে একই Type এর Object যদি আমাদের প্রয়োজন হয়, তাহলে আমরা **Costructor Function** তৈরি করতে পারি। এই Function আসলে Object তৈরির Machine এর মতো কাজ করে, যার মাধ্যমে একই Type এর Object যত খুশি তত তৈরি করা যায়।

```
// Define the constructor function
function Student(firstName, lastName, id, batch) {
  // Properties
  this.firstName = firstName;
  this.lastName = lastName;
  this.id = id;
  this.batch = batch;

  // Method
  this.getDetails = function () {
    return `Name: ${this.firstName} ${this.lastName}, ID: ${this.id}, Batch: ${this.batch}`;
  };
}
```

```
// Create student objects using the constructor function
let alamin = new Student("Al", "Amin", "WD05020", 5);
let sujon = new Student("Sujon", "Rana", "WD05025", 5);

// Using the method
console.log(alamin.getDetails()); // Output: Name: Al Amin, ID: WD05020, Batch: 5
console.log(sujon.getDetails()); // Output: Name: Sujon Rana, ID: WD05025, Batch: 5
```

- Constructor Function এর নাম Capital Letter এ লিখতে হয়। JavaScript এ অন্যান্য Function লিখার ক্ষেত্রে Camel Case এবং Constructor Function লিখার ক্ষেত্রে Capital Letter এ লিখতে হয়। যাতে যে কেউ কোড দেখলেই বুঝতে পারে এটা Constructor Function.
- আমরা চাইলে Normal Object এর Property যেভাবে অ্যাড করি, এখানেও সেইভাবে New Property অ্যাড করা যায়। যেমনঃ

```
alamin.score = 100;
```

Built-in JavaScript Constructors

```
new Object(); // A new Object object
new Array(); // A new Array object
new Map(); // A new Map object
new Set(); // A new Set object
new Date(); // A new Date object
new RegExp(); // A new RegExp object
new Function(); // A new Function object
```

- The Math() object is not in the list. Math is a global object.
  The new keyword cannot be used on Math.

JavaScript Event

- HTML Elements এর মাধ্যমে কোন ঘটনা ঘটাকেই **Event** বলে। যখন HTML Page এ JavaScript ব্যবহার করা হয়, তখন JavaScript এই ইভেন্টগুলিতে "React" বা "Listen" করতে পারে। অর্থাৎ Event হয় HTML এ, আর JavaScript সেই Event Listen করে বা React করে। যেমনঃ Button এ ক্লিক করা একটা Event, Mouse Hover করা ইত্যাদি।

Common JavaScript Events

Event Type	**Event**	**Description**
Mouse Events	click	Fires when a mouse button is clicked on an element.
	dblclick	Fires when a mouse button is double-clicked on an element.
	mouseover	Fires when the mouse pointer is moved onto an element.
	mouseout	Fires when the mouse pointer is moved out of an element.
	mousemove	Fires when the mouse pointer is moved within an element.
	mousedown	Fires when a mouse button is pressed on an element.
	mouseup	Fires when a mouse button is released over an element.
Keyboard Events	keydown	Fires when a key is pressed.
	keyup	Fires when a key is released.
	keypress	Fires when a key is pressed and released.
Form Events	submit	Fires when a form is submitted.
	change	Fires when an element's value changes.
	focus	Fires when an element receives focus.
	blur	Fires when an element loses focus.

Event Type	**Event**	**Description**
	input	Fires when the value of an input element changes.
Window Events	load	Fires when the whole page has loaded, including all dependent resources like stylesheets and images.
	resize	Fires when the browser window is resized.
	scroll	Fires when the document view is scrolled.
	unload	Fires when the user navigates away from the page.
Touch Events	touchstart	Fires when a touch point is placed on the touch surface.
	touchmove	Fires when a touch point is moved along the touch surface.
	touchend	Fires when a touch point is removed from the touch surface.
	touchcancel	Fires when a touch point is interrupted.
Drag and Drop Events	drag	Fires when an element is being dragged.
	dragstart	Fires when the user starts dragging an element.
	dragend	Fires when the user has finished dragging the element.
	dragenter	Fires when the dragged element enters a drop target.

Event Type	Event	Description
	dragover	Fires when the dragged element is over a drop target.
	dragleave	Fires when the dragged element leaves a drop target.
	drop	Fires when the dragged element is dropped on a drop target.
Clipboard Events	copy	Fires when content is copied to the clipboard.
	cut	Fires when content is cut from the document and added to the clipboard.
	paste	Fires when content is pasted from the clipboard into the document.

Download the PDF

- Event Syntax:

```
<button onclick="takeAction()">Click me</button>
```

Example:

```
<!DOCTYPE html>
<html>
<head>
   <title>JavaScript Event Example</title>
   <script>
      // JavaScript function to change the text
      function changeText() {
         document.getElementById("myParagraph").innerHTML = "Text has been
changed!";
      }
   </script>
```

```
</head>
<body>

<h2>JavaScript Event Example</h2>

<!-- Button with an onclick event to call the changeText function -->
<button onclick="changeText()">Click me</button>

<!-- Paragraph with an id to target with JavaScript -->
<p id="myParagraph">This is the original text.</p>

</body>
</html>
```

- A JavaScript Counter Demonstrating Events:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript Counter Example</title>
    <script>
      // Initialize the counter variable
      let counter = 0;

      // Function to increase the counter
      function increaseCounter() {
        counter++;
        displayCounter();
      }

      // Function to decrease the counter
      function decreaseCounter() {
        counter--;
        displayCounter();
      }

      // Function to display the counter value
      function displayCounter() {
        document.getElementById("counterDisplay").innerHTML = counter;
      }
    </script>
  </head>
  <body>
    <h2>JavaScript Counter Example</h2>

    <!-- Button to increase the counter -->
    <button onclick="increaseCounter()">Increase</button>

    <!-- Button to decrease the counter -->
    <button onclick="decreaseCounter()">Decrease</button>

    <!-- Paragraph to display the counter value -->
    <p>Counter: <span id="counterDisplay">0</span></p>
  </body>
</html>
```

Explanation:

20. **HTML Structure**:
    - Two <button> elements, one for increasing the counter and one for decreasing the counter.
    - A <p> element with a <span> inside it to display the counter value. The <span> has an id of counterDisplay to target it with JavaScript.
21. **JavaScript Functions**:

        - A counter variable is initialized to 0.
        - increaseCounter and decreaseCounter functions are defined to modify the counter variable and update the display by calling displayCounter.
        - The displayCounter function updates the inner HTML of the <span> element with the current value of the counter.

When you click the "Increase" button, the increaseCounter function is called, which increments the counter and updates the display. Similarly, when you click the "Decrease" button, the decreaseCounter function is called, which decrements the counter and updates the display.

    - নিজের Element এর Content Change করতে চাইলে তাকে id দিয়ে আলাদা করে ধরার কোন দরকার নেইI এর বদলে this.innerHTML ব্যবহার করা যেতে পারেI

<button onclick="this.innerHTML = Date()">The time is?</button>

Chapter-05: JavaScript String

    - String
    - String Methods

String

    - Single এবং Double Quotation এর মধ্যে যা থাকে তাকেই String বলেI Single/Double Quotation এর মধ্যে প্রতিটা Character এর ASCII Value আছেI ASCII = American Standard Code for Information Interchange.
    - String Empty হতে পারেI

Escape Characters

    - JavaScript strings support various escape characters that allow you to include special characters within a string. Here are some common escape characters used in JavaScript:

Escape Character	**Description**	**Example**	**Output**
\'	Single quote	'It\'s a pen'	It's a pen
\"	Double quote	"He said, \"Hi\""	He said, "Hi"

Escape Character	Description	Example	Output
\\	Backslash	"This is a backslash: \\ "	This is a backslash: \
\n	New line	"Line 1\nLine 2"	Line 1 Line 2
\r	Carriage return	"Hello\rWorld"	World
\t	Tab	"Hello\tWorld"	Hello World
\b	Backspace	"ABC\bDEF"	ABDEF
\f	Form feed	"Hello\fWorld"	HelloWorld
Example	Demonstrating multiple escape characters	"She said, \"Hello!\"\nThis is a backslash: \\"	She said, "Hello!" This is a backslash: \

- The backslash escape character (\) turns special characters into string characters.

String Methods

- JavaScript এ সমস্ত String Methods Original String কে পরিবর্তন না করে নতুন String তৈরি করে।

String Methods At A Glance

Method	Description	Example Code	Output
charAt()	Returns the character at a specified index in a string	let str = "Hello"; str.charAt(1);	e

Method	Description	Example Code	Output
concat()	Joins two or more strings and returns a new string	let str1 = "Hello"; let str2 = "World"; str1.concat(" ", str2);	Hello World
includes()	Checks if a string contains a specified substring	let str = "Hello World"; str.includes("World");	true
indexOf()	Returns the index of the first occurrence of a specified value	let str = "Hello World"; str.indexOf("World");	6
slice()	Extracts a part of a string and returns a new string	let str = "Hello World"; str.slice(0, 5);	Hello
split()	Splits a string into an array of substrings	let str = "Hello World"; str.split(" ");	["Hello", "World"]
toLowerCase()	Converts a string to lowercase	let str = "Hello World"; str.toLowerCase();	hello world
toUpperCase()	Converts a string to uppercase	let str = "Hello World"; str.toUpperCase();	HELLO WORLD
trim()	Removes whitespace from both ends of a string	let str = " Hello World "; str.trim();	Hello World

Method	Description	Example Code	Output
replace()	Replaces a specified value with another value in a string	let str = "Hello World"; str.replace("World", "JavaScript");	Hello JavaScript
substring()	Extracts characters from a string, between two specified indices	let str = "Hello World"; str.substring(0, 5);	Hello
startsWith()	Checks if a string starts with a specified value	let str = "Hello World"; str.startsWith("Hello");	true
endsWith()	Checks if a string ends with a specified value	let str = "Hello World"; str.endsWith("World");	true

Extracting String Characters

- চার উপায়ে String এর Characters কে Extract করা যায়। যেমনঃ
  - at(position) ব্যবহার করে।
  - charAt(position) ব্যবহার করে।
  - charCodeAt(position) ব্যবহার করে।
  - Array এর মতো [] ব্যবহার করে।

Method	Description	Example Code	Output
at()	Returns the character at a specified index. Supports negative	let str = "Hello"; str.at(1);	e

Method	Description	Example Code	Output
	indices to count from the end.		
		let str = "Hello"; str.at(-1);	o
charAt()	Returns the character at a specified index. It doesn't support negative index.	let str = "Hello"; str.charAt(1);	e
charCodeAt()	Returns the Unicode value of the character at a specified index.	let str = "Hello"; str.charCodeAt(1);	101

Extracting String Parts

- slice(start, end)
- substring(start, end)
- substr(start, length)

Slice Method

The slice() method in JavaScript is used to extract a portion of an array into a new array. This method does not alter the original array but instead returns a new array containing the selected elements.

Syntax

array.slice(start, end);

- start: Optional. The starting index at which to begin extraction. If negative, it indicates an offset from the end of the array. Default is 0.
- end: Optional. The ending index before which to end extraction (the element at this index is not included). If negative, it indicates an offset from the end of the array. If omitted, it extracts through the end of the array.

Examples

36. **Basic Usage**

```
let fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

let citrus = fruits.slice(1, 3);
```

console.log(citrus); // Output: ["Orange", "Lemon"]

2. **Using Negative Indices**

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

let lastTwo = fruits.slice(-2);

console.log(lastTwo); // Output: ["Apple", "Mango"]

3. **Omitting the end Parameter**

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

let fromSecond = fruits.slice(1);

console.log(fromSecond); // Output: ["Orange", "Lemon", "Apple", "Mango"]

Notes

- The original array remains unchanged.
- If start is greater than the length of the array, an empty array is returned.
- If end is greater than the length of the array, the slice extracts through the end of the array.

Substring Method

- The substring method in JavaScript is used to extract a portion of a string and returns it as a new string, without modifying the original string. Here's a simple explanation of how it works:
- substring() is similar to slice().
- The difference is that start and end values less than 0 are treated as 0 in substring().

Syntax

string.substring(indexStart, indexEnd);

- indexStart (required): The index of the first character to include in the returned substring.
- indexEnd (optional): The index of the first character to exclude from the returned substring. If omitted, substring extracts characters to the end of the string.

Key Points

45. **Zero-based Indexing**: The indices are zero-based, meaning the first character of the string is at index 0.

46. **Index Order**: If indexStart is greater than indexEnd, substring will swap the two arguments.

47. **Negative Indices**: If either indexStart or indexEnd is less than 0, it is treated as 0.

48. **Out of Range**: If any of the indices are greater than the string's length, they are treated as equal to the string's length.

Examples

49. **Basic Usage**

50. let str = "Hello, world!";

51. let result = str.substring(0, 5);

console.log(result); // Outputs: "Hello"

52. **Omitting indexEnd**

53. let str = "Hello, world!";

54. let result = str.substring(7);

console.log(result); // Outputs: "world!"

55. **Swapping Indices**

56. let str = "Hello, world!";

57. let result = str.substring(7, 0);

console.log(result); // Outputs: "Hello, "

58. **Negative Index**

59. let str = "Hello, world!";

60. let result = str.substring(-5, 5);

console.log(result); // Outputs: "Hello"

61. **Index Out of Range**

62. let str = "Hello, world!";

63. let result = str.substring(7, 20);

console.log(result); // Outputs: "world!"

Substr Method

- substr() is similar to slice().
- The difference is that the second parameter specifies the **length** of the extracted part.

let str = "Hello, world!";

let result = str.substr(7, 5);

console.log(result); // Outputs: "world"

Converting Uppercase and Lowercase

toUpperCase Method

The toUpperCase method converts all the characters in a string to uppercase.

Syntax

string.toUpperCase();

Example

let str = "Hello, world!";

let upperStr = str.toUpperCase();

console.log(upperStr); // Outputs: "HELLO, WORLD!"

toLowerCase Method

The toLowerCase method converts all the characters in a string to lowercase.

Syntax

string.toLowerCase();

Example

let str = "Hello, World!";

let lowerStr = str.toLowerCase();

console.log(lowerStr); // Outputs: "hello, world!"

Concat Method

- The concat method in JavaScript is used to merge two or more strings into one. This method does not change the existing strings but returns a new string containing the combined text of the strings provided as arguments.

Syntax

string1.concat(string2, string3, ..., stringN)

- string1, string2, ..., stringN: The strings to be concatenated with the original string.

Key Points

68. **Non-Mutating**: The concat method does not alter the original strings. It returns a new string.

69. **Multiple Arguments**: You can pass multiple strings as arguments to concatenate them all at once.

70. **Alternate Method**: The + operator can also be used to concatenate strings.

Examples

71. **Basic Usage**

72. let str1 = "Hello, ";

73. let str2 = "world!";

74. let result = str1.concat(str2);

console.log(result); // Outputs: "Hello, world!"

75. **Concatenating Multiple Strings**

76. let str1 = "Hello";

77. let str2 = ", ";

78. let str3 = "world";

79. let str4 = "!";

80. let result = str1.concat(str2, str3, str4);

console.log(result); // Outputs: "Hello, world!"

81. **Using concat with an Empty String**

82. let str1 = "Hello, ";

83. let str2 = "";

84. let str3 = "world!";

85. let result = str1.concat(str2, str3);

console.log(result); // Outputs: "Hello, world!"

86. **Alternative Using the + Operator**

87. let str1 = "Hello, ";

88. let str2 = "world!";

89. let result = str1 + str2;

console.log(result); // Outputs: "Hello, world!"

The concat method is a straightforward way to combine strings, and while the + operator is often used for its simplicity, concat can be particularly useful when concatenating multiple strings in a single method call.

Trim Method

- The trim method in JavaScript is used to remove whitespace from both ends of a string. It does not change the original string but returns a new string with the leading and trailing whitespace removed.

Syntax

string.trim();

Key Points

91. **Whitespace Removal**: It removes spaces, tabs, and other whitespace characters from the beginning and end of the string.

92. **Non-Mutating**: The trim method does not alter the original string but returns a new string with the whitespace removed.

Examples

93. **Basic Usage**

94. let str = "   Hello, world!   ";

95. let trimmedStr = str.trim();

console.log(trimmedStr); // Outputs: "Hello, world!"

96. **No Whitespace to Remove**

97. let str = "Hello, world!";

98. let trimmedStr = str.trim();

console.log(trimmedStr); // Outputs: "Hello, world!"

99. **String with Only Whitespace**

100. let str = "   ";

101. let trimmedStr = str.trim();

console.log(trimmedStr); // Outputs: ""

102. **Whitespace in the Middle of the String**

103. let str = "   Hello,   world!   ";

104. let trimmedStr = str.trim();

console.log(trimmedStr); // Outputs: "Hello,   world!"

The trim method is particularly useful for cleaning up user input or processing strings where whitespace at the ends might cause issues.

padStart and padEnd Method

The padStart and padEnd methods in JavaScript are used to pad the current string with another string until the resulting string reaches the given length. The padding is applied from the start or end of the string respectively.

padStart Method

The padStart method pads the current string from the start with another string until the resulting string reaches the specified length.

Syntax

string.padStart(targetLength, padString);

- targetLength (required): The length of the resulting string once the current string has been padded. If this length is less than the length of the original string, no padding is added.
- padString (optional): The string to pad the current string with. If this string is too long, it is truncated. The default value is a space character (" ").

Example

let str = "5";

let paddedStr = str.padStart(3, "0");

console.log(paddedStr); // Outputs: "005"

padEnd Method

The padEnd method pads the current string from the end with another string until the resulting string reaches the specified length.

Syntax

string.padEnd(targetLength, padString);

- targetLength (required): The length of the resulting string once the current string has been padded. If this length is less than the length of the original string, no padding is added.
- padString (optional): The string to pad the current string with. If this string is too long, it is truncated. The default value is a space character (" ").

Example

let str = "5";

let paddedStr = str.padEnd(3, "0");

console.log(paddedStr); // Outputs: "500"

Additional Examples

109. **Using padStart with Default Padding**

110. let str = "42";

111. let paddedStr = str.padStart(5);

console.log(paddedStr); // Outputs: "   42"

112. **Using padEnd with Default Padding**

113. let str = "42";

114. let paddedStr = str.padEnd(5);

console.log(paddedStr); // Outputs: "42   "

115. **Padding with a Custom String**

116. let str = "123";

117. let paddedStrStart = str.padStart(6, "abc");

118. console.log(paddedStrStart); // Outputs: "abc123"

119.

120. let paddedStrEnd = str.padEnd(6, "abc");

console.log(paddedStrEnd); // Outputs: "123abc"

121. **Padding with a Truncated Pad String**

122. let str = "123";

123. let paddedStr = str.padStart(10, "abcdef");

console.log(paddedStr); // Outputs: "abcdefa123"

repeat Method

The repeat method in JavaScript is used to construct and return a new string which contains the specified number of copies of the string on which it was called, concatenated together.

Syntax

string.repeat(count);

- count (required): An integer between 0 and positive infinity, indicating the number of times to repeat the string. If this count is negative or infinity, a RangeError is thrown.

Examples

125. **Basic Usage**

126. let str = "abc";

127. let repeatedStr = str.repeat(3);

console.log(repeatedStr); // Outputs: "abcabcabc"

128. **Zero Count**

129. let str = "abc";

130. let repeatedStr = str.repeat(0);

console.log(repeatedStr); // Outputs: ""

131. **Floating Point Count**

132. let str = "abc";

133. let repeatedStr = str.repeat(2.5);

console.log(repeatedStr); // Outputs: "abcabc" (count is converted to an integer)

134. **RangeError for Negative Count**

135. let str = "abc";

136. try {

137.   let repeatedStr = str.repeat(-1);

138. } catch (e) {

139.   console.log(e); // Outputs: RangeError: Invalid count value

}

140. **Using Repeat for Padding**

141. let str = "abc";

142. let paddedStr = str + " ".repeat(5) + "def";

console.log(paddedStr); // Outputs: "abc   def"

143. **Combining Repeat with Other Methods**

144. let str = "abc";

145. let repeatedUpperStr = str.repeat(2).toUpperCase();

console.log(repeatedUpperStr); // Outputs: "ABCABC"

The repeat method is useful for generating repeated sequences of a string, such as for creating padding, repeating patterns, or generating test data.

replace Method

The replace method in JavaScript is used to return a new string with some or all matches of a pattern replaced by a replacement. The pattern can be a string or a regular expression, and the replacement can be a string or a function to generate the string.

Syntax

string.replace(pattern, replacement);

- pattern (required): The substring or regular expression to be replaced.
- replacement (required): The string or function that replaces the matched substrings.

Key Points

148. **First Match Only**: If pattern is a string, only the first occurrence will be replaced.

149. **Global Replacement**: To replace all occurrences, use a regular expression with the g (global) flag.

150. **Replacement String**: Can include special replacement patterns like $& (the matched substring), $ (the preceding portion), $' (the following portion), and more.

151. **Replacement Function**: Can be used for more complex replacements, where the function's return value replaces the matched substring.

Examples

152. **Basic Usage**

```
153.    let str = "Hello, world!";
154.    let newStr = str.replace("world", "there");
console.log(newStr); // Outputs: "Hello, there!"
```

155. **Global Replacement with Regular Expression**

```
156.    let str = "Hello, world! Hello, everyone!";
157.    let newStr = str.replace(/Hello/g, "Hi");
console.log(newStr); // Outputs: "Hi, world! Hi, everyone!"
```

158. **Using Special Replacement Patterns**

```
159.    let str = "abc123";
160.    let newStr = str.replace(/(\d+)/, "Number: $1");
console.log(newStr); // Outputs: "abcNumber: 123"
```

161. **Replacement Function**

```
162.    let str = "The quick brown fox jumps over the lazy dog.";
163.    let newStr = str.replace(/\b\w+\b/g, function (match) {
```

164. return match.toUpperCase();

165. });

console.log(newStr); // Outputs: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."

166. **Replacing with Empty String**

167. let str = "Hello, world!";

168. let newStr = str.replace("world", "");

console.log(newStr); // Outputs: "Hello, !"

169. **Replacing Multiple Patterns**

170. let str = "Twas the night before Christmas.";

171. let newStr = str.replace(/night|Christmas/g, function (match) {

172. if (match === "night") return "day";

173. if (match === "Christmas") return "New Year";

174. });

console.log(newStr); // Outputs: "Twas the day before New Year."

The replace method is powerful for string manipulation, offering flexibility to replace substrings or patterns with static replacements or dynamically generated content.

replaceAll Method

The replaceAll method in JavaScript is used to replace all occurrences of a specified substring or regular expression within a string with a new substring. It is a more convenient and readable way to perform a global replacement compared to using replace with a regular expression and the global flag (/g).

Syntax

string.replaceAll(pattern, replacement);

- pattern (required): The substring or regular expression to be replaced.
- replacement (required): The string or function that replaces the matched substrings.

Key Points

177. **Global Replacement**: replaceAll automatically replaces all occurrences of the pattern, similar to using replace with the global flag.

178. **String and RegExp Patterns**: The pattern can be a string or a regular expression with the global flag.

179. **Replacement String or Function**: The replacement can be a string or a function that returns the replacement string.

Examples

180. **Basic Usage with a String Pattern**

181. let str = "Hello, world! Hello, everyone!";

182. let newStr = str.replaceAll("Hello", "Hi");

console.log(newStr); // Outputs: "Hi, world! Hi, everyone!"

183. **Using a Regular Expression Pattern**

184. let str = "Hello, world! Hello, everyone!";

185. let newStr = str.replaceAll(/Hello/g, "Hi");

console.log(newStr); // Outputs: "Hi, world! Hi, everyone!"

186. **Replacing with Special Characters**

187. let str = "Hello, world! Hello, everyone!";

188. let newStr = str.replaceAll("!", "?");

console.log(newStr); // Outputs: "Hello, world? Hello, everyone?"

189. **Replacing with a Function**

190. let str = "The quick brown fox jumps over the lazy dog.";

191. let newStr = str.replaceAll(/\b\w+\b/g, function (match) {

192.   return match.toUpperCase();

193. });

console.log(newStr); // Outputs: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."

194. **Replacing Multiple Different Patterns**

195. let str = "Apples are red. Apples are tasty.";

196. let newStr = str.replaceAll("Apples", "Oranges").replaceAll("red", "orange");

console.log(newStr); // Outputs: "Oranges are orange. Oranges are tasty."

197. **Replacing Using Special Replacement Patterns**

198. let str = "abc123abc456";

199. let newStr = str.replaceAll(/abc/g, "XYZ");

console.log(newStr); // Outputs: "XYZ123XYZ456"

Converting a String to an Array

- If you want to work with a string as an array, you can convert it to an array.
- A string can be converted to an array with the split() method.
- If the separator is "", the returned array will be an array of single characters.
- If the separator is omitted, the returned array will contain the whole string in index [0].

text.split(","); // Split on commas

text.split(" "); // Split on spaces

text.split("|"); // Split on pipe

text.split(""); //split by characters

JavaScript String Search

indexOf() Method and lastIndexOf() Method

The indexOf and lastIndexOf methods in JavaScript are used to find the index of a specified substring within a string. The indexOf method returns the first occurrence, while the lastIndexOf method returns the last occurrence.

indexOf Method

The indexOf method returns the index of the first occurrence of a specified substring within the string. If the substring is not found, it returns -1.

Syntax

string.indexOf(searchValue, fromIndex);

- searchValue (required): The substring to search for.
- fromIndex (optional): The index to start the search from. The default is 0.

Examples

206. **Basic Usage**

207. let str = "Hello, world!";

208. let index = str.indexOf("world");

console.log(index); // Outputs: 7

209. **Not Found**

210. let str = "Hello, world!";

211. let index = str.indexOf("planet");

console.log(index); // Outputs: -1

212. **Starting Search from a Specific Index**

213. let str = "Hello, world! Hello again!";

214. let index = str.indexOf("Hello", 10);

console.log(index); // Outputs: 13

lastIndexOf Method

The lastIndexOf method returns the index of the last occurrence of a specified substring within the string. If the substring is not found, it returns -1.

Syntax

string.lastIndexOf(searchValue, fromIndex);

- searchValue (required): The substring to search for.
- fromIndex (optional): The index to start the search backward from. The default is str.length - 1.

Examples

217. **Basic Usage**

218. let str = "Hello, world! Hello again!";

219. let index = str.lastIndexOf("Hello");

console.log(index); // Outputs: 13

220. **Not Found**

221. let str = "Hello, world!";

222. let index = str.lastIndexOf("planet");

console.log(index); // Outputs: -1

223. **Starting Search Backward from a Specific Index**

224. let str = "Hello, world! Hello again!";

225. let index = str.lastIndexOf("Hello", 12);

console.log(index); // Outputs: 0

Additional Examples

226. **Finding All Occurrences Using a Loop**

227. let str = "Hello, world! Hello again!";

228. let searchValue = "Hello";

229. let indices = [];

230. let index = str.indexOf(searchValue);

231.

232. while (index !== -1) {

233.   indices.push(index);

234.   index = str.indexOf(searchValue, index + 1);

235. }

236.

console.log(indices); // Outputs: [0, 13]

237. **Case-Sensitive Search**

238. let str = "Hello, World!";

239. let index = str.indexOf("world");

console.log(index); // Outputs: -1 (case-sensitive)

240. **Using lastIndexOf for Substring Search**

241. let str = "banana";

242. let index = str.lastIndexOf("na");

console.log(index); // Outputs: 4

serach() Method

The search method in JavaScript is used to search a string for a specified value (pattern) and returns the index (position) of the first match. The search value can be a string or a regular expression. This method returns -1 if no match is found.

Syntax

string.search(pattern);

- pattern (required): A string or a regular expression to search for.

Key Points

244. **Search Value**: The pattern can be a string or a regular expression.

245. **Returns Index**: The method returns the index of the first match. If no match is found, it returns -1.

246. **Case-Sensitive**: The search is case-sensitive.

Examples

247. **Basic Usage with a String Pattern**

248. let str = "Hello, world!";

249. let index = str.search("world");

console.log(index); // Outputs: 7

250. **No Match Found**

251. let str = "Hello, world!";

252. let index = str.search("planet");

console.log(index); // Outputs: -1

253. **Using a Regular Expression**

254. let str = "Hello, world!";

255. let index = str.search(/world/);

console.log(index); // Outputs: 7

256. **Case-Sensitive Search**

257. let str = "Hello, World!";

258. let index = str.search(/world/i); // Using 'i' flag for case-insensitive search

console.log(index); // Outputs: 7

259. **Finding a Digit Using Regular Expression**

260. let str = "Hello, world! 2024";

261. let index = str.search(/\d/);

console.log(index); // Outputs: 13

262. **Using Special Characters in Regular Expression**

263. let str = "Hello, world!";

264. let index = str.search(/\W/); // Search for the first non-word character

console.log(index); // Outputs: 5

Additional Considerations

- **Regular Expressions**: When using regular expressions, you can take advantage of various flags and patterns for more complex searches.
- **Difference from indexOf**: Unlike indexOf, which only searches for a simple substring, search can use regular expressions, providing more flexibility in pattern matching.

match() Method

The match method in JavaScript is used to retrieve the result of matching a string against a regular expression. It returns an array containing all the matches, or null if no match is found. This method can be very powerful when combined with regular expressions for pattern matching and extraction.

Syntax

string.match(regexp);

- regexp (required): A regular expression object to match against the string.

Key Points

268. **Regular Expression**: The regexp parameter must be a regular expression.

269. **Return Value**: Returns an array with the matches, or null if no match is found.

270. **Global Flag**: If the regular expression includes the global (g) flag, the method returns an array of all matches. Otherwise, it returns an array with the first match and its capturing groups.

Examples

271. **Basic Usage without Global Flag**

272. let str = "The quick brown fox jumps over the lazy dog.";

273. let result = str.match(/quick/);

console.log(result); // Outputs: ["quick"]

274. **No Match Found**

275. let str = "The quick brown fox jumps over the lazy dog.";

276. let result = str.match(/cat/);

console.log(result); // Outputs: null

277. **Using Global Flag**

278. let str = "The quick brown fox jumps over the lazy dog.";

279. let result = str.match(/\w+/g);

console.log(result); // Outputs: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]

280. **Capturing Groups**

281. let str = "The quick brown fox jumps over the lazy dog.";

282. let result = str.match(/(quick) (brown)/);

console.log(result); // Outputs: ["quick brown", "quick", "brown"]

283. **Using Special Characters**

284. let str = "The quick brown fox jumps over the lazy dog.";

285. let result = str.match(/[aeiou]/g);

console.log(result); // Outputs: ["e", "u", "i", "o", "o", "u", "o", "e", "e", "a", "o"]

286. **Case-Insensitive Search**

287. let str = "The Quick Brown Fox Jumps Over The Lazy Dog.";

288. let result = str.match(/the/gi);

console.log(result); // Outputs: ["The", "The"]

Additional Considerations

- **Global vs. Non-Global**: Without the global flag (g), the match method returns an array with the first match and its capturing groups. With the global flag, it returns an array of all matches without capturing groups.
- **Null Return**: If no matches are found, match returns null, so it's often a good practice to check the result before processing it.

The match method is a versatile tool for pattern matching in strings, allowing for both simple and complex searches with regular expressions.

matchAll() Method

The matchAll method in JavaScript is used to retrieve all matches of a regular expression within a string, along with their capturing groups. This method returns an iterator of all the matches, allowing for comprehensive and detailed pattern matching.

Syntax

string.matchAll(regexp);

- regexp (required): A regular expression object. It must have the global (g) or sticky (y) flag set; otherwise, a TypeError is thrown.

Key Points

292. **Global or Sticky Flag Required**: The regular expression must include the global (g) or sticky (y) flag.

293. **Return Value**: Returns an iterator of arrays containing the matches, capturing groups, and their indices.

294. **Iterator**: The method returns an iterator, which can be converted to an array using the spread operator (...) or Array.from.

Examples

Basic Usage with Global Flag

```
let str = "The quick brown fox jumps over the lazy dog.";
let regexp = /\b\w+\b/g;
let matches = str.matchAll(regexp);

for (let match of matches) {
  console.log(match);
}
// Outputs:
// ["The", index: 0, input: "The quick brown fox jumps over the lazy dog.", groups: undefined]
// ["quick", index: 4, input: "The quick brown fox jumps over the lazy dog.", groups: undefined]
// ["brown", index: 10, input: "The quick brown fox jumps over the lazy dog.", groups: undefined]
// ...
```

Using the Spread Operator to Convert to Array

```
let str = "The quick brown fox jumps over the lazy dog.";
let regexp = /\b\w+\b/g;
let matches = [...str.matchAll(regexp)];
console.log(matches);
// Outputs an array of all match objects
```

Capturing Groups

```
let str = "The quick brown fox jumps over the lazy dog.";
let regexp = /(\w+)\s(\w+)/g;
let matches = str.matchAll(regexp);

for (let match of matches) {
  console.log(match);
```

319. }

320. // Outputs:

321. // ["The quick", "The", "quick", index: 0, input: "The quick brown fox jumps over the lazy dog.", groups: undefined]

322. // ["brown fox", "brown", "fox", index: 10, input: "The quick brown fox jumps over the lazy dog.", groups: undefined]

// ...

323. **Converting Iterator to Array Using Array.from**

324. let str = "The quick brown fox jumps over the lazy dog.";

325. let regexp = /\b\w+\b/g;

326. let matches = Array.from(str.matchAll(regexp));

327. console.log(matches);

// Outputs an array of all match objects

Additional Considerations

- **Iterator Nature**: Since matchAll returns an iterator, you can use it in a for...of loop, spread operator, or Array.from to process the matches.
- **Detailed Match Information**: Each match object includes detailed information such as the match itself, the index where it was found, the input string, and any capturing groups.

The matchAll method is a powerful tool for extracting multiple matches and their details from a string, making it ideal for comprehensive pattern matching with regular expressions.

includes() Method

The includes method in JavaScript is used to determine whether one string contains another substring. It returns true if the substring is found within the string, and false otherwise.

Syntax

string.includes(searchString, position);

- searchString (required): The substring to search for within the string.
- position (optional): The position in the string at which to start searching. The default is 0.

Key Points

332. **Case-Sensitive**: The search is case-sensitive.

333. **Default Position**: If the position parameter is not specified, the search starts from the beginning of the string.

Examples

334. **Basic Usage**

335. let str = "Hello, world!";

336. let result = str.includes("world");

console.log(result); // Outputs: true

337. **Case-Sensitivity**

338. let str = "Hello, world!";

339. let result = str.includes("World");

console.log(result); // Outputs: false

340. **Starting Search from a Specific Position**

341. let str = "Hello, world!";

342. let result = str.includes("world", 8);

console.log(result); // Outputs: false

343. **Using includes with a Long String**

344. let str = "The quick brown fox jumps over the lazy dog.";

345. let result = str.includes("fox");

console.log(result); // Outputs: true

346. **Checking for a Substring Not Present**

347. let str = "Hello, world!";

348. let result = str.includes("planet");

console.log(result); // Outputs: false

Additional Considerations

- **Polyfill for Older Environments**: If you need to support environments that do not have includes, you can use a polyfill:
- if (!String.prototype.includes) {
-   String.prototype.includes = function (search, start) {
-     "use strict";

- if (typeof start !== "number") {
- start = 0;
- }
-
- if (start + search.length > this.length) {
- return false;
- } else {
- return this.indexOf(search, start) !== -1;
- }
- };

}

- **Case-Insensitive Search**: If you need a case-insensitive search, you can convert both strings to the same case (either lower or upper) before using includes:
- let str = "Hello, world!";
- let result = str.toLowerCase().includes("world".toLowerCase());

console.log(result); // Outputs: true

The includes method is a straightforward and effective way to check for the presence of a substring within a string, making it useful for various string manipulation tasks in JavaScript.

startsWith() and endsWith() Methods

The startsWith and endsWith methods in JavaScript are used to check whether a string begins or ends with a specified substring. They both return true if the string matches the specified characters at the start or end, respectively, and false otherwise.

startsWith Method

The startsWith method checks if a string starts with the specified substring.

Syntax

string.startsWith(searchString, position);

- searchString (required): The substring to search for at the start of the string.
- position (optional): The position in the string to start the search. The default is 0.

Examples

368. **Basic Usage**
369. let str = "Hello, world!";

console.log(str.startsWith("Hello")); // Outputs: true

370. **Case-Sensitivity**
371. let str = "Hello, world!";

console.log(str.startsWith("hello")); // Outputs: false

372. **Starting from a Specific Position**
373. let str = "Hello, world!";

console.log(str.startsWith("world", 7)); // Outputs: true

endsWith Method

The endsWith method checks if a string ends with the specified substring.

Syntax

string.endsWith(searchString, length);

- searchString (required): The substring to search for at the end of the string.
- length (optional): The length of the string to consider. The default is the full length of the string.

Examples

376. **Basic Usage**
377. let str = "Hello, world!";

console.log(str.endsWith("world!")); // Outputs: true

378. **Case-Sensitivity**
379. let str = "Hello, world!";

console.log(str.endsWith("World!")); // Outputs: false

380. **Considering a Substring Length**
381. let str = "Hello, world!";

console.log(str.endsWith("Hello", 5)); // Outputs: true

Additional Considerations

382. **Polyfills for Older Environments**

startsWith Polyfill:

```
if (!String.prototype.startsWith) {
  String.prototype.startsWith = function (searchString, position) {
    position = position || 0;
    return this.indexOf(searchString, position) === position;
  };
}
```

endsWith Polyfill:

```
if (!String.prototype.endsWith) {
  String.prototype.endsWith = function (searchString, this_len) {
    if (this_len === undefined || this_len > this.length) {
      this_len = this.length;
    }
    return (
      this.substring(this_len - searchString.length, this_len) ===
      searchString
    );
  };
}
```

383. **Case-Insensitive Search**

For a case-insensitive check, convert both the string and the search substring to the same case:

```
let str = "Hello, world!";
console.log(str.toLowerCase().startsWith("hello".toLowerCase())); // Outputs: true
console.log(str.toLowerCase().endsWith("world!".toLowerCase())); // Outputs: true
```

Both startsWith and endsWith methods are useful for string validation and manipulation, providing a straightforward way to check if a string begins or ends with a specified substring.

JavaScript Template Strings

- এর অনেকগুলো নাম আছে, যেমনঃ

        - String Templates
        - Template Strings
        - Template Literals
    - একটি String Define করার জন্য Template String Double Quotation("") বা Single Quotation('') এর পরিবর্তে Back-tics(``)ব্যবহার করে। যেমনঃ

```
let text = `Hello World!`;
```

Features of Template String

- **Quotes Inside Strings:** Template Strings allow both single and double quotes inside a string. যেমনঃ

```
let text = `He's often called "Johnny"`;
```

- **Multiline Strings:** Template Strings allow multiline strings. যেমনঃ

```
let text = `Bangladesh is
a beautiful
country`;
```

- **Interpolation:** Template String Interpolation Support করে। কোন একটা Variable তার Value দিয়ে Automatically Replace হয়ে যাওয়াকে String Interpolation বলে। Interpolation Syntax: ${expression}, যেমনঃ

```
let x = 10;
let y = 20;
let text = `He has got ${x + y} marks`;
```

- **HTML Template:**

```
<body>
  <div id="app"></div>

  <script src="script.js"></script>
</body>
// JavaScript function to create and return an HTML template
function createTemplate(title, content) {
  return `
```

```
    <div class="container">
      <h2>${title}</h2>
      <p>${content}</p>
    </div>
  `;
}

// JavaScript function to insert the template into the DOM
function insertTemplate(template) {
  const app = document.getElementById("app");
  app.innerHTML += template;
}

// Example usage
const title = "Dynamic HTML Template";
const content =
  "This is an example of creating an HTML template using JavaScript.";
const template = createTemplate(title, content);

insertTemplate(template);
```

JavaScript Numbers

Binary Search Algorithm

```
function binarySearch(arr, value) {
  let low = 0;
  let high = arr.length - 1;

  while (low <= high) {
    let mid = Math.floor((low + high) / 2);
    if (arr[mid] === value) {
      return true;
    } else if (arr[mid] < value) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return false;
}

// Example usage:
let arr = [1, 3, 5, 7, 9, 11, 13, 15];
arr.sort((a, b) => a - b); // Ensure the array is sorted
let value = 7;

if (binarySearch(arr, value)) {
  console.log("Found");
} else {
  console.log("Not Found");
}
```

How arr.sort((a, b) => a - b) line works: The line arr.sort((a, b) => a - b); is used to sort the array arr in ascending order. Here's a detailed explanation:

390. **arr.sort(...):** This is the JavaScript Array.prototype.sort() method, which sorts the elements of an array in place and returns the sorted array. By default, this method sorts elements as strings in ascending order. This can lead to unexpected results when sorting numbers. For example, without a comparison function, the array [10, 2, 30, 4] would be sorted as [10, 2, 30, 4], not [2, 4, 10, 30].

391. **Comparison function:** To correctly sort numbers, you need to provide a comparison function. The comparison function is passed two arguments (often referred to as a and b), which represent two elements in the array that are being compared.

392. **(a, b) => a - b:** This is an arrow function used as the comparison function for sorting. It works as follows:

- If the result of a - b is negative, a is considered smaller than b, and a will be placed before b in the sorted array.
- If the result of a - b is positive, a is considered larger than b, and a will be placed after b in the sorted array.
- If the result of a - b is zero, a and b are considered equal, and their order relative to each other will not be changed.

By using this comparison function, the array elements are sorted numerically in ascending order. Here is a step-by-step example:

For the array [10, 2, 30, 4]:

- When comparing 10 and 2, 10 - 2 is 8 (positive), so 2 comes before 10.
- When comparing 2 and 30, 2 - 30 is -28 (negative), so 2 remains before 30.
- When comparing 10 and 30, 10 - 30 is -20 (negative), so 10 remains before 30.
- When comparing 4 and 2, 4 - 2 is 2 (positive), so 2 remains before 4.
- When comparing 4 and 10, 4 - 10 is -6 (negative), so 4 comes before 10.
- When comparing 4 and 30, 4 - 30 is -26 (negative), so 4 comes before 30.

The sorted array will be [2, 4, 10, 30].

↥ Go to Top

Chapter-06: JavaScript Numbers, Bigint, Number Methods, Number Methods, Number Properties

JavaScript Numbers

- C, C++, Java ইত্যাদি Programming Language এ যেমন পূর্ণ সংখ্যা ও দশমিক সংখ্যার জন্য আলাদা আলাদা Keyword দিয়ে Declare করতে হয়, JavaScript এ এমন নেই। শুধু একটা Keyword দিয়েই পূর্ণ সংখ্যা ও দশমিক সংখ্যা Declare করা যায়। উদাহরণঃ

```
int x = 57; // for integer value
float y = 12.92; // for floating point value
let x = 57; // for integer value
let y = 12.92; // for floating point value
```

- অতিরিক্ত বড় বা অতিরিক্ত ছোট সংখ্যাগুলি scientific (exponent) notation দিয়ে লেখা যেতে পারে। যেমনঃ

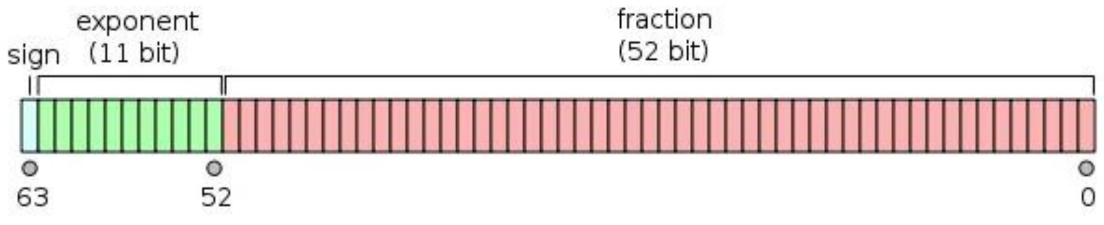
let x = 123e5; // 12300000

let y = 123e-5; // 0.00123

JavaScript Numbers are Always 64-bit Floating Point Numbers

What is 64-bit floating point?

- 64-bit floating point হলো একটা নাম্বারের binary representation. JavaScript এ এই 64-bit Number System কে ৩ ভাগে ভাগ করা হয়েছে। 1 bit for the sign, 11 bits for the exponent and 52 bits for the fraction.



source: wikipedia

- **Sign Bit:** Sign Bit দ্বারা নির্ধারণ হয় সংখ্যাটি Positive নাকি Negative. 0 থাকলে সংখ্যাটি Positive, 1 থাকলে সংখ্যাটি Negative.
- **Exponent:** পরের ১১ টি Bit.
- **Fraction/Mantissa:** পরের ৫২ টি Bit.

NaN (Not a Number) in JavaScript

- NaN is a JavaScript reserved word indicating that a number is not a legal number.

Introduction

NaN stands for "Not-a-Number". It is a property of the global object in JavaScript and indicates a value that is not a legal number. This value is unique in that it is not equal to itself.

In JavaScript, NaN is a property of the global object. The global object is a special object that always exists in the global scope.

When NaN is Returned

NaN is typically returned in situations where a mathematical operation cannot be performed. Here are a few scenarios where NaN might be encountered:

407. **Arithmetic Operations**:

let result = 0 / 0; // NaN

408. **Invalid Number Conversion**:

let result = Number("hello"); // NaN

409. **Math Functions**:

let result = Math.sqrt(-1); // NaN

Checking for NaN

Due to NaN being the only value in JavaScript that is not equal to itself, you cannot use the equality operator to check for NaN. Instead, you should use the isNaN() function or the Number.isNaN() method.

isNaN() Function

The isNaN() function converts the value to a number before testing it, which can lead to some unexpected results:

console.log(isNaN("hello")); // true (string "hello" is converted to NaN)

console.log(isNaN(123)); // false (123 is a number)

Number.isNaN() Method

The Number.isNaN() method is more reliable because it does not convert the value before testing it:

console.log(Number.isNaN("hello")); // false (string "hello" is not converted)

console.log(Number.isNaN(NaN)); // true

Examples of NaN

Let's look at some examples to understand how NaN works in JavaScript:

Example 1: Division by Zero

let result = 0 / 0;

console.log(result); // NaN

Example 2: Invalid Number Conversion

let result = Number("hello");

console.log(result); // NaN

Example 3: Mathematical Functions

let result = Math.sqrt(-1);

console.log(result); // NaN

Example 4: Checking for NaN

Using isNaN():

let value = "hello";

if (isNaN(value)) {

  console.log(value + " is NaN"); // hello is NaN

}

Using Number.isNaN():

let value = NaN;

if (Number.isNaN(value)) {

  console.log("Value is NaN"); // Value is NaN

}

Infinity in JavaScript

- In JavaScript, Infinity is a property of the global object and represents a value that is larger than any other number. It is a special numeric value that behaves mathematically as infinity does. Infinity can be positive or negative.

Positive Infinity

Positive Infinity is the value JavaScript returns when a number exceeds the upper limit of the floating-point numbers. You can get positive Infinity by dividing a positive number by zero or by performing calculations that result in a number too large to represent.

Examples of Positive Infinity

411. **Division by Zero**:
412. let result = 1 / 0;

console.log(result); // Infinity

413. **Exponentiation**:
414. let result = Math.pow(10, 1000);

console.log(result); // Infinity

Checking for Infinity

You can check if a value is Infinity by comparing it to Infinity:

let value = 1 / 0;

if (value === Infinity) {

  console.log("Value is positive Infinity");

}

Negative Infinity

Negative Infinity is the value JavaScript returns when a number exceeds the lower limit of the floating-point numbers in the negative direction. You can get negative Infinity by dividing a negative number by zero or by performing calculations that result in a number too large in the negative direction to represent.

Examples of Negative Infinity

415. **Division by Zero**:
416. let result = -1 / 0;

console.log(result); // -Infinity

417. **Exponentiation**:
418. let result = Math.pow(-10, 1000);

console.log(result); // -Infinity

Checking for Negative Infinity

You can check if a value is negative Infinity by comparing it to -Infinity:

let value = -1 / 0;

if (value === -Infinity) {

  console.log("Value is negative Infinity");

}

Mathematical Operations with Infinity

Infinity can be used in various mathematical operations:

- **Adding Infinity**:
- let result = Infinity + 1;

console.log(result); // Infinity

- **Subtracting Infinity**:
- let result = Infinity - 1;

console.log(result); // Infinity

- **Multiplying Infinity**:
- let result = Infinity * 2;

console.log(result); // Infinity

- **Dividing Infinity**:
- let result = Infinity / 2;

console.log(result); // Infinity

Hexadecimal

- In JavaScript, hexadecimal numbers are prefixed with 0x or 0X. যেমনঃ

let hexNumber = 0x1a; // 1A in hexadecimal is 26 in decimal

console.log(hexNumber); // Output: 26

- By default, JavaScript displays numbers as base 10 decimals. But you can use the **toString()** method to output numbers from **base 2 to base 36**. যেমনঃ

let myNumber = 32;

console.log(myNumber.toString(32)); // Convert to base 32

console.log(myNumber.toString(16)); // Convert to base 16 (hexadecimal)

console.log(myNumber.toString(12)); // Convert to base 12

console.log(myNumber.toString(10)); // Convert to base 10 (decimal)

console.log(myNumber.toString(8)); // Convert to base 8 (octal)

console.log(myNumber.toString(2)); // Convert to base 2 (binary)

Output:

10;

20;

28;

32;

40;

100000;

Bigint in JavaScript

- **BigInt** is a built-in object in JavaScript that provides a way to represent whole numbers larger than the largest number JavaScript can reliably represent with the Number primitive, which is (2^{53} - 1) (Number.MAX_SAFE_INTEGER). BigInt can be used for arbitrarily large integers.

Creating BigInt

You can create a BigInt by appending n to the end of an integer literal or by calling the BigInt function.

Example:

```
// Using the 'n' suffix
let bigInt1 = 1234567890123456789012345678901234567890n;

// Using the BigInt function
let bigInt2 = BigInt("1234567890123456789012345678901234567890");

console.log(bigInt1); // Output: 1234567890123456789012345678901234567890n
console.log(bigInt2); // Output: 1234567890123456789012345678901234567890n
```

Operations with BigInt

BigInts support the standard arithmetic operations such as addition, subtraction, multiplication, division, and exponentiation. However, you cannot mix BigInt with Number types directly. If you need to operate on both, you must convert them to the same type.

Example:

```
let a = 100000000000000000000n;
let b = 200000000000000000000n;

// Addition
let sum = a + b;
console.log(sum); // Output: 300000000000000000000n

// Subtraction
let difference = b - a;
console.log(difference); // Output: 100000000000000000000n
```

```
// Multiplication
let product = a * b;
console.log(product); // Output: 20000000000000000000000000000000000000000n

// Division
let quotient = b / a;
console.log(quotient); // Output: 2n

// Exponentiation
let power = a ** 2n;
console.log(power); // Output: 10000000000000000000000000000000000000000n
```

Mixing BigInt and Number

You cannot directly mix BigInt and Number types in arithmetic operations. You need to convert one type to the other.

Example:

```
let bigIntValue = 100000000000000000000n;
let numberValue = 20;

// Convert Number to BigInt
let result1 = bigIntValue + BigInt(numberValue);
console.log(result1); // Output: 100000000000000000020n

// Convert BigInt to Number
let result2 = Number(bigIntValue) + numberValue;
console.log(result2); // Output: 100000000000000000020
```

Comparisons

BigInts can be compared to other BigInts and Numbers using comparison operators.

Example:

```
let bigIntValue = 100000000000000000000n;
```

let numberValue = 100000000000000000000;

console.log(bigIntValue > 50n); // Output: true

console.log(bigIntValue === BigInt(numberValue)); // Output: true

console.log(bigIntValue === numberValue); // Output: false

console.log(bigIntValue == numberValue); // Output: true (loose equality)

Use Cases for BigInt

- **Cryptography**: Handling large integers is common in cryptographic algorithms.
- **Accurate Large Number Calculations**: Situations where you need precise calculations with very large numbers, such as financial calculations, astronomical calculations, etc.
- **Database IDs**: Some databases use very large integers for unique identifiers.

JavaScript Number Methods

433. **toString()**: Converts a number to a string.
434. let num = 123;

num.toString(); // "123"

435. **toExponential()**: Converts a number to an exponential notation string.
436. let num = 123;

num.toExponential(2); // "1.23e+2"

437. **toFixed()**: Formats a number using fixed-point notation.
438. let num = 123.456;

num.toFixed(2); // "123.46"

439. **toPrecision()**: Formats a number to a specified length.
440. let num = 123.456;

num.toPrecision(4); // "123.5"

441. **valueOf()**: Returns the primitive value of a Number object.
442. let numObj = new Number(123);

numObj.valueOf(); // 123

443. **Number()**: Converts an object to a number.

Number("123"); // 123

444. **parseFloat()**: Parses a string and returns a floating-point number.

parseFloat("123.45"); // 123.45

445. **parseInt()**: Parses a string and returns an integer of the specified radix.

parseInt("101", 2); // 5

446. **isNaN()**: Determines whether a value is NaN.

isNaN("hello"); // true

447. **isFinite()**: Determines whether a value is a finite number.

isFinite(123); // true

JavaScript Number Properties

Property	**Description**	**Example**
Number.EPSILON	The smallest interval between two numbers.	console.log(Number.EPSILON); // 2.220446049250313e-16
Number.MAX_SAFE_INTEGER	Maximum safe integer in JavaScript (2^53 - 1).	console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991
Number.MIN_SAFE_INTEGER	Minimum safe integer in JavaScript (-(2^53 - 1)).	console.log(Number.MIN_SAFE_INTEGER); // -9007199254740991
Number.MAX_VALUE	Largest positive	console.log(Number.MAX_VALUE); // 1.7976931348623157e+308

Property	Description	Example
	representable number.	
Number.MIN_VALUE	Smallest positive representable number.	console.log(Number.MIN_VALUE); // 5e-324
Number.NaN	Represents Not-A-Number.	console.log(Number.NaN); // NaN
Number.NEGATIVE_INFINITY	Represents negative infinity.	console.log(Number.NEGATIVE_INFINITY); // -Infinity
Number.POSITIVE_INFINITY	Represents positive infinity.	console.log(Number.POSITIVE_INFINITY); // Infinity

Quiz

448. What will be the result after the following JavaScript code is executed?

let x = 10;

let y = 20;

let z = "30";

let result = x + y + z;

Click here to see the answer

3030

2. What will be the result after the following JavaScript code is executed?

let x = "100";

let y = "10";

let z = x * y;

Click here to see the answer

1000

3. What will be the result after the following JavaScript code is executed?

let x = "100";

let y = "10";

let z = x - y;

Click here to see the answer

90

4. What will be the result after the following JavaScript code is executed?

let x = "100";

let y = "10";

let z = x + y;

Click here to see the answer

10010

↥ Go to Top

Chapter-07: JavaScript Array, Array Methods, Array Search, Array Sort, Array Iteration

How to Create Array in JavaScript

468. **Using Array Literals**

469. // Creating an array of numbers

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

471.

472. // Creating an array of strings

473. let fruits = ["Apple", "Banana", "Cherry"];

474.

475. // Creating an array with mixed data types

let mixed = [1, "Hello", true, null];

Use Case: This is the most common and straightforward way to create arrays. It's best used when you know the elements in advance and want to create an array quickly.

476. **Using the Array Constructor**

477. // Creating an array with the Array constructor

478. let numbers = new Array(1, 2, 3, 4, 5);

479.

480. // Creating an empty array with a specific length

let emptyArray = new Array(5); // Creates an array with 5 undefined elements

Use Case: Use the Array constructor when you need to create an array with a specified number of elements or when initializing arrays with a known set of elements. Be cautious with the single numeric argument as it sets the array length.

481. **Using Array.of()**

482. // Creating an array with Array.of()

483. let numbers = Array.of(1, 2, 3, 4, 5);

484.

485. // Creating an array with mixed data types

let mixed = Array.of(1, "Hello", true, null);

Use Case: Use Array.of() when you need to create an array from a list of arguments. It's especially useful when you want to ensure a single numeric argument is treated as an array element rather than an array length.

486. **Using Array.from()**

487. // Creating an array from a string

488. let stringArray = Array.from("Hello"); // ['H', 'e', 'l', 'l', 'o']

489.

490. // Creating an array from a Set

491. let set = new Set([1, 2, 3]);

492. let setArray = Array.from(set); // [1, 2, 3]

493.

494. // Creating an array from an array-like object

495. let arrayLike = { length: 3, 0: "a", 1: "b", 2: "c" };

let arrayFromObject = Array.from(arrayLike); // ['a', 'b', 'c']

Use Case: Use Array.from() to create arrays from array-like objects (e.g., NodeLists, arguments objects) or iterables (e.g., strings, sets). It's also useful for transforming data structures into arrays.

496. **Using Spread Operator**

497. // Creating an array by spreading elements of another array

498. let numbers = [1, 2, 3];

499. let newNumbers = [...numbers, 4, 5]; // [1, 2, 3, 4, 5]

500.

501. // Creating an array by spreading elements of a string

let stringArray = [..."Hello"]; // ['H', 'e', 'l', 'l', 'o']

Use Case: Use the spread operator to easily create copies of arrays or to concatenate arrays. It's also helpful for converting iterable objects like strings into arrays.

502. **Using Array Methods (e.g., push)**

503. // Creating an array and adding elements with push

504. let numbers = [];

505. numbers.push(1);

506. numbers.push(2);

507. numbers.push(3);

// numbers is now [1, 2, 3]

Use Case: Use array methods like push() when you need to build an array dynamically by adding elements over time. This is particularly useful in loops or when processing data incrementally.

508. **Using Index Assignment**

509. const course = [];

510. course[0] = "Web Development";

511. course[1] = "Data Science";

512. course[2] = "Machine Learning";

513. course[3] = "Cyber Security";

// course is now ["Web Development", "Data Science", "Machine Learning", "Cyber Security"]

Use Case: Use index assignment when you need to populate an array at specific indices, especially when you might not be able to initialize all elements at once. This method is useful in scenarios where elements are added based on conditions or in non-sequential order.

How To Access The Array Elements

Accessing elements in an array in JavaScript can be done in several ways. Here are the primary methods:

514. **Accessing by Index**

515. const fruits = ["Apple", "Banana", "Cherry"];

516.

517. // Accessing the first element

518. let firstFruit = fruits[0]; // "Apple"

519.

520. // Accessing the second element

521. let secondFruit = fruits[1]; // "Banana"

522.

523. // Accessing the third element

let thirdFruit = fruits[2]; // "Cherry"

Use Case: Use this method when you need to access specific elements in the array by their position. Indexing starts from 0.

524. **Using a Loop (e.g., for loop)**

525. const fruits = ["Apple", "Banana", "Cherry"];

526.

527. // Accessing elements using a for loop

528. for (let i = 0; i < fruits.length; i++) {

529.   console.log(fruits[i]);

}

Use Case: Use loops to iterate over all elements in the array when you need to perform operations on each element.

530. **Using for...of Loop**

531. const fruits = ["Apple", "Banana", "Cherry"];

532.

533. // Accessing elements using for...of loop

534. for (let fruit of fruits) {

535.   console.log(fruit);

}

Use Case: Use the for...of loop for a more readable syntax when you need to iterate over the elements of an array.

536. **Using Array Methods (e.g., forEach)**

537. const fruits = ["Apple", "Banana", "Cherry"];

538.

539. // Accessing elements using forEach method

540. fruits.forEach(function (fruit, index) {

541.   console.log(index, fruit);

```
});
```

Use Case: Use the forEach method to iterate over the array elements with the ability to access the element and its index. This is useful for more complex operations involving each element.

542. **Destructuring Assignment**

543. const fruits = ["Apple", "Banana", "Cherry"];

544.

545. // Accessing elements using destructuring

546. const [firstFruit, secondFruit, thirdFruit] = fruits;

547.

548. console.log(firstFruit); // "Apple"

549. console.log(secondFruit); // "Banana"

```
console.log(thirdFruit); // "Cherry"
```

Use Case: Use destructuring for a concise way to extract multiple elements from an array and assign them to variables in a single statement.

How To Convert An Array Into A String

- The JavaScript method toString() converts an array to a string of (comma separated) array values.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.toString());
```

How to Convert a String Into An Array

551. **Using split() Method**

552. let str = "Hello,World,JavaScript";

553.

554. // Converting string to array using comma as a delimiter

555. let arr = str.split(","); // ["Hello", "World", "JavaScript"]

556.

557. // Converting string to array using space as a delimiter

558. let strWithSpaces = "Hello World JavaScript";

```
let arrWithSpaces = strWithSpaces.split(" "); // ["Hello", "World", "JavaScript"]
```

Use Case: Use split() when you want to split a string into an array based on a specific delimiter, such as commas, spaces, or other characters.

559. **Using Array.from()**

560. let str = "Hello";

561.

562. // Converting string to array of characters

let arr = Array.from(str); // ["H", "e", "l", "l", "o"]

Use Case: Use Array.from() when you want to convert a string into an array of individual characters.

563. **Using Spread Operator**

564. let str = "Hello";

565.

566. // Converting string to array of characters

let arr = [...str]; // ["H", "e", "l", "l", "o"]

Use Case: Use the spread operator for a concise and readable way to convert a string into an array of characters.

567. **Using Object.assign()**

568. let str = "Hello";

569.

570. // Converting string to array of characters

let arr = Object.assign([], str); // ["H", "e", "l", "l", "o"]

Use Case: Use Object.assign() for converting a string to an array of characters in a slightly unconventional way, though it's not as commonly used as the other methods.

Difference Between Array and Object

Arrays and objects are both fundamental data structures in JavaScript, but they have different purposes and characteristics. Here are the key differences, along with examples:

Arrays

- **Purpose**: Arrays are used to store ordered collections of items, typically of the same type, and are accessed by their numeric indices.

- **Characteristics**: Arrays maintain the order of elements and have built-in methods for common operations like adding, removing, and iterating over elements.

Example of Array:

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

// Accessing elements by index
console.log(fruits[0]); // "Apple"
console.log(fruits[1]); // "Banana"

// Adding an element
fruits.push("Date");
console.log(fruits); // ["Apple", "Banana", "Cherry", "Date"]

// Removing an element
fruits.pop();
console.log(fruits); // ["Apple", "Banana", "Cherry"]

// Iterating over elements
fruits.forEach(function (fruit) {
  console.log(fruit);
});
// Output:
// Apple
// Banana
// Cherry
```

Objects

- **Purpose**: Objects are used to store collections of key-value pairs. The keys (also called properties) are strings (or Symbols), and the values can be of any type.

- **Characteristics**: Objects do not maintain order, and each key must be unique within the object. Objects are better suited for storing related data and for representing more complex data structures.

Example of Object:

```
let person = {
  name: "John",
  age: 30,
  job: "Developer",
};

// Accessing properties
console.log(person.name); // "John"
console.log(person["age"]); // 30

// Adding a new property
person.city = "New York";
console.log(person); // { name: "John", age: 30, job: "Developer", city: "New York" }

// Removing a property
delete person.job;
console.log(person); // { name: "John", age: 30, city: "New York" }

// Iterating over properties
for (let key in person) {
  console.log(key + ": " + person[key]);
}
// Output:
// name: John
// age: 30
// city: New York
```

Key Differences

575. **Order**:
    - Arrays maintain the order of elements based on their indices.
    - Objects do not guarantee the order of their properties.
576. **Access**:
    - Arrays are accessed using numeric indices.
    - Objects are accessed using keys (which are usually strings).
577. **Use Case**:
    - Use arrays when the data is an ordered collection or list.
    - Use objects when you need to represent data as key-value pairs or to model more complex structures.
578. **Built-in Methods**:
    - Arrays have methods like push(), pop(), shift(), unshift(), slice(), splice(), forEach(), map(), filter(), etc.
    - Objects do not have these methods, but you can use methods like Object.keys(), Object.values(), Object.entries(), etc.

Combined Example:

Sometimes you might use arrays and objects together to represent complex data structures.

Example:

```
let students = [
  { name: "Alice", age: 20, grade: "A" },
  { name: "Bob", age: 22, grade: "B" },
  { name: "Charlie", age: 23, grade: "C" },
];

// Accessing an object in an array
console.log(students[0].name); // "Alice"

// Iterating over an array of objects
```

```
students.forEach(function (student) {
  console.log(
    student.name +
      " is " +
      student.age +
      " years old and has grade " +
      student.grade
  );
});
// Output:
// Alice is 20 years old and has grade A
// Bob is 22 years old and has grade B
// Charlie is 23 years old and has grade C
```

Array Elements Can Be Anything!

- In JavaScript, arrays are a special type of object. This means that arrays can hold a mix of different types of values.
- JavaScript arrays are flexible and powerful. They can hold any type of value, including numbers, strings, objects, functions, and even other arrays. This makes them very useful for a wide range of applications.

Let's break it down to make it easier to understand:

581. **Arrays Can Hold Different Types**: Unlike some other programming languages, JavaScript allows you to store different types of values in a single array. For example, you can have numbers, strings, objects, functions, and even other arrays in the same array.

582. **Example of Different Types in an Array**:

```
583. let myArray = [];
584. myArray[0] = 42; // A number
585. myArray[1] = "Hello, world!"; // A string
586. myArray[2] = { name: "John" }; // An object
587. myArray[3] = function () {
588.   // A function
```

589.   console.log("I am a function!");

590. };

myArray[4] = [1, 2, 3]; // Another array

591. **Using Functions and Objects in Arrays**:

- **Function in an Array**: You can store a function in an array and call it later.
- let myFunctionArray = [];
- myFunctionArray[0] = function () {
-   console.log("Hello from the function!");
- };
- // Call the function stored in the array

myFunctionArray[0](); // Outputs: Hello from the function!

- **Object in an Array**: You can store an object in an array and access its properties.
- let myObjectArray = [];
- myObjectArray[0] = { name: "Alice", age: 25 };
- // Access properties of the object stored in the array

console.log(myObjectArray[0].name); // Outputs: Alice

592. **Arrays Inside Arrays**: You can also store arrays inside other arrays, creating multi-dimensional arrays.

593. let myNestedArray = [];

594. myNestedArray[0] = [1, 2, 3];

595. myNestedArray[1] = ["a", "b", "c"];

596. // Access elements of the nested array

597. console.log(myNestedArray[0][1]); // Outputs: 2

console.log(myNestedArray[1][2]); // Outputs: c

598. **Special Methods for Arrays**:

- **Date.now**: You can store the current timestamp using Date.now.
- let myArray = [];
- myArray[0] = Date.now;

console.log(myArray[0]()); // Outputs the current timestamp

ForEach On Array

- The Array.forEach() function in JavaScript is used to execute a provided function once for each array element. It is a convenient way to iterate over the elements of an array.
- **Array.forEach()** is used to execute a function on each element of an array.
- It can take three parameters: currentValue, index, and array.
- It does not create or return a new array; it returns undefined.
- It is useful for performing operations like logging, modifying existing arrays, or executing side effects.

If you need to transform an array and get a new array with the results, you should use methods like Array.map() instead.

Here's a simple guide to help you understand and use forEach():

604. **Basic Syntax**:
605. array.forEach(function (currentValue, index, array) {
606.   // code to execute for each element

});

  - currentValue: The current element being processed in the array.
  - index (optional): The index of the current element being processed.
  - array (optional): The array that forEach() is being applied to.

607. **Simple Example**:
608. let numbers = [1, 2, 3, 4, 5];
609.
610. numbers.forEach(function (number) {
611.   console.log(number);
612. });

// Outputs: 1, 2, 3, 4, 5 (each on a new line)

613. **Using the Index Parameter**:
614. let fruits = ["apple", "banana", "cherry"];
615.

```
616.  fruits.forEach(function (fruit, index) {
617.    console.log(index + ": " + fruit);
618.  });
619.  // Outputs:
620.  // 0: apple
621.  // 1: banana
// 2: cherry
```

622. **Using an Arrow Function**: Arrow functions can make the syntax more concise.

```
623.  let letters = ["a", "b", "c"];
624.
625.  letters.forEach((letter, index) => {
626.    console.log(`${index}: ${letter}`);
627.  });
628.  // Outputs:
629.  // 0: a
630.  // 1: b
// 2: c
```

631. **Modifying Elements Inside forEach**: forEach does not return a new array, but you can modify the elements of the array directly.

```
632.  let scores = [10, 20, 30];
633.
634.  scores.forEach((score, index, arr) => {
635.    arr[index] = score + 5;
636.  });
637.
console.log(scores); // Outputs: [15, 25, 35]
```

638. **Practical Example - Logging Object Properties**:

```
639.  let users = [
640.    { name: "Alice", age: 25 },
641.    { name: "Bob", age: 30 },
642.     { name: "Charlie", age: 35 },
643.   ];
644.
645.   users.forEach((user) => {
646.     console.log(`Name: ${user.name}, Age: ${user.age}`);
647.   });
648.   // Outputs:
649.   // Name: Alice, Age: 25
650.   // Name: Bob, Age: 30
// Name: Charlie, Age: 35
```

651. **Limitations**:

- forEach cannot break out of the loop like a regular for loop or for...of loop. If you need to stop iterating based on a condition, consider using a for loop instead.
- It does not return a new array; it returns undefined.

Adding Array Elements in JavaScript

1. Using push()

The push() method adds one or more elements to the end of an array and returns the new length of the array.

```
let fruits = ["apple", "banana"];
fruits.push("cherry");


console.log(fruits); // Outputs: ["apple", "banana", "cherry"]
```

2. Using unshift()

The unshift() method adds one or more elements to the beginning of an array and returns the new length of the array.

```
let fruits = ["apple", "banana"];
fruits.unshift("mango");


console.log(fruits); // Outputs: ["mango", "apple", "banana"]
```

3. Using splice()

The splice() method adds and/or removes elements from an array. To add elements, you specify the start index, the number of elements to remove (in this case, 0), and the elements to add.

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

fruits.splice(1, 0, "orange", "cherry");


console.log(fruits); // Outputs: ["apple", "orange", "cherry", "banana"]
```

4. Direct Assignment by Index

You can add elements by directly assigning a value to a specific index. If the index is beyond the current length of the array, JavaScript will automatically increase the array's size.

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

fruits[5] = "grape";


console.log(fruits); // Outputs: ["apple", "banana", undefined, undefined, undefined,
"grape"]
```

5. Using the concat() Method

The concat() method creates a new array by merging existing arrays or adding new elements to an array.

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

let moreFruits = fruits.concat("cherry", "mango");


console.log(moreFruits); // Outputs: ["apple", "banana", "cherry", "mango"]
```

6. Using the Spread Operator

The spread operator (...) can be used to add elements to an array by creating a new array.

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

let moreFruits = ["cherry", ...fruits, "mango"];


console.log(moreFruits); // Outputs: ["cherry", "apple", "banana", "mango"]
```

Summary

- **push()**: Adds elements to the end of an array.
- **unshift()**: Adds elements to the beginning of an array.
- **splice()**: Adds elements at a specified index.
- **Direct Assignment by Index**: Adds elements at specific positions, expanding the array if necessary.
- **concat()**: Creates a new array by merging existing arrays or adding elements.
- **Spread Operator (...)**: Adds elements to an array by creating a new array.

JavaScript Does Not Support Associative Arrays

What are Associative Arrays?

Arrays with named indexes are called associative arrays (or hashes).

Associative Arrays in JavaScript

Many programming languages support arrays with named indexes. However, JavaScript does not support associative arrays. In JavaScript, arrays always use numbered indexes. If you want to use named indexes, you should use objects instead.

Examples

658. **Numbered Indexes in Arrays**:

659. let fruits = ["apple", "banana", "cherry"];

660.

661. console.log(fruits[0]); // Outputs: apple

662. console.log(fruits[1]); // Outputs: banana

console.log(fruits[2]); // Outputs: cherry

663. **Using Objects for Named Indexes**: If you want to use named indexes, you should use an object.

664. let person = {

665.   firstName: "John",

666.   lastName: "Doe",

667.   age: 30,

668. };

669.

670. console.log(person["firstName"]); // Outputs: John

671. console.log(person["lastName"]); // Outputs: Doe

console.log(person["age"]); // Outputs: 30

When to Use Arrays and When to use Objects

Arrays

Use arrays when:

- You need an ordered collection of items.
- The element names (indexes) are numbers.
- You want to perform array-specific operations like sorting, filtering, and mapping.

Example of an Array:

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

console.log(fruits[0]); // Outputs: apple
console.log(fruits[1]); // Outputs: banana
console.log(fruits[2]); // Outputs: cherry
```

Array-Specific Operations:

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

let doubled = numbers.map((number) => number * 2);
console.log(doubled); // Outputs: [2, 4, 6, 8, 10]

let sorted = numbers.sort((a, b) => b - a);
console.log(sorted); // Outputs: [5, 4, 3, 2, 1]
```

Objects

Use objects when:

- You need a collection of key-value pairs.
- The keys are strings (text).
- You want to store and access data using descriptive keys.

Example of an Object:

```
let person = {
```

  firstName: "John",
  lastName: "Doe",
  age: 30,
};

console.log(person["firstName"]); // Outputs: John
console.log(person["lastName"]); // Outputs: Doe
console.log(person["age"]); // Outputs: 30

Object-Specific Use Cases:

- **Storing configuration settings:**
- let config = {
-   theme: "dark",
-   language: "en",
-   showSidebar: true,
- };
-

console.log(config.theme); // Outputs: dark

- **Representing real-world entities:**
- let car = {
-   make: "Toyota",
-   model: "Corolla",
-   year: 2020,
- };
-

console.log(car.model); // Outputs: Corolla

Summary

- **Use Arrays**:
  - When you need an ordered list.
  - When element names are numbers.
  - For array-specific methods like push, pop, map, filter, and sort.

- **Use Objects**:
    - When you need key-value pairs.
    - When keys are strings.
    - To represent complex entities with properties and methods.

Array Methods

Method	**Description**
concat()	Merges two or more arrays and returns a new array.
every()	Checks if every element in an array passes a test provided as a function.
filter()	Creates a new array with all elements that pass the test implemented by the provided function.
find()	Returns the value of the first element that satisfies the provided testing function.
findIndex()	Returns the index of the first element that satisfies the provided testing function.
forEach()	Executes a provided function once for each array element.
includes()	Determines whether an array includes a certain value, returning true or false.
indexOf()	Returns the first index at which a given element can be found in the array.
join()	Joins all elements of an array into a string.
map()	Creates a new array with the results of calling a provided function on every element in the array.
pop()	Removes the last element from an array and returns that element.

Method	Description
push()	Adds one or more elements to the end of an array and returns the new length of the array.
reduce()	Executes a reducer function on each element of the array, resulting in a single output value.
reduceRight()	Executes a reducer function on each element of the array, from right to left, resulting in a single output value.
reverse()	Reverses the order of the elements in an array.
shift()	Removes the first element from an array and returns that element.
slice()	Returns a shallow copy of a portion of an array into a new array object.
some()	Checks if at least one element in the array passes the test implemented by the provided function.
sort()	Sorts the elements of an array in place and returns the sorted array.
splice()	Changes the contents of an array by removing or replacing existing elements and/or adding new elements.
toString()	Returns a string representing the specified array and its elements.
unshift()	Adds one or more elements to the beginning of an array and returns the new length of the array.
flat()	Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
flatMap()	First maps each element using a mapping function, then flattens the result into a new array.

Method	Description
from()	Creates a new array instance from an array-like or iterable object.
isArray()	Determines whether the passed value is an array.
of()	Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

Here's the detailed explanation of all 27 JavaScript array methods with source code examples:

694. **concat()**

- **Description:** Merges two or more arrays and returns a new array.
- **Explanation:** Combines multiple arrays into one. It does not change the existing arrays but returns a new array.
- **Example:**
- const arr1 = [1, 2, 3];
- const arr2 = [4, 5, 6];
- const newArray = arr1.concat(arr2);

console.log(newArray); // [1, 2, 3, 4, 5, 6]

695. **every()**

- **Description:** Checks if every element in an array passes a test provided as a function.
- **Explanation:** Returns true if all elements pass the test implemented by the provided function; otherwise, it returns false.
- **Example:**
- const arr = [2, 4, 6, 8];
- const allEven = arr.every((num) => num % 2 === 0);

console.log(allEven); // true

696. **filter()**

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

- **Explanation:** Filters the array based on the condition specified in the callback function and returns a new array with the elements that pass the test.
- **Example:**
- const arr = [1, 2, 3, 4, 5];
- const evenNumbers = arr.filter((num) => num % 2 === 0);

console.log(evenNumbers); // [2, 4]

697. **find()**

- **Description:** Returns the value of the first element that satisfies the provided testing function.
- **Explanation:** Searches the array for the first element that meets the condition specified in the callback function and returns that element.
- **Example:**
- const arr = [1, 2, 3, 4, 5];
- const found = arr.find((num) => num > 3);

console.log(found); // 4

698. **findIndex()**

- **Description:** Returns the index of the first element that satisfies the provided testing function.
- **Explanation:** Searches the array for the first element that meets the condition specified in the callback function and returns the index of that element.
- **Example:**
- const arr = [1, 2, 3, 4, 5];
- const index = arr.findIndex((num) => num > 3);

console.log(index); // 3

699. **forEach()**

- **Description:** Executes a provided function once for each array element.
- **Explanation:** Iterates over each element in the array and executes the provided function.

- **Example:**
- const arr = [1, 2, 3];
- arr.forEach((num) => console.log(num));
- // 1
- // 2

// 3

700. **includes()**
   - **Description:** Determines whether an array includes a certain value, returning true or false.
   - **Explanation:** Checks if the array contains a specific value and returns true if it does, otherwise false.
   - **Example:**
   - const arr = [1, 2, 3];
   - const hasValue = arr.includes(2);

console.log(hasValue); // true

701. **indexOf()**
   - **Description:** Returns the first index at which a given element can be found in the array.
   - **Explanation:** Searches the array for a specified value and returns the first index where the value is found. Returns -1 if the value is not found.

1.
   - **Example:**
   - const arr = [1, 2, 3, 2];
   - const index = arr.indexOf(2);

console.log(index); // 1

2. **join()**
   - **Description:** Joins all elements of an array into a string.
   - **Explanation:** Concatenates all elements of the array into a single string, with an optional separator.
   - **Example:**

- const arr = [1, 2, 3];
- const str = arr.join(", ");

console.log(str); // "1, 2, 3"

3. **map()**
   - **Description:** Creates a new array with the results of calling a provided function on every element in the array.
   - **Explanation:** Applies the provided function to each element in the array and returns a new array with the results.
   - **Example:**
   - const arr = [1, 2, 3];
   - const squares = arr.map((num) => num * num);

console.log(squares); // [1, 4, 9]

4. **pop()**
   - **Description:** Removes the last element from an array and returns that element.
   - **Explanation:** Removes the last element from the array, reducing its length by one, and returns the removed element.
   - **Example:**
   - const arr = [1, 2, 3];
   - const lastElement = arr.pop();
   - console.log(lastElement); // 3

console.log(arr); // [1, 2]

5. **push()**
   - **Description:** Adds one or more elements to the end of an array and returns the new length of the array.
   - **Explanation:** Appends new elements to the end of the array and returns the new length of the array.
   - **Example:**
   - const arr = [1, 2];
   - const newLength = arr.push(3, 4);
   - console.log(newLength); // 4

console.log(arr); // [1, 2, 3, 4]

6. **reduce()**
   - **Description:** Executes a reducer function on each element of the array, resulting in a single output value.
   - **Explanation:** Reduces the array to a single value by executing the provided function for each element from left to right.
   - **Example:**
   - const arr = [1, 2, 3, 4];
   - const sum = arr.reduce((acc, num) => acc + num, 0);

console.log(sum); // 10

7. **reduceRight()**
   - **Description:** Executes a reducer function on each element of the array, from right to left, resulting in a single output value.
   - **Explanation:** Similar to reduce(), but processes the array elements from right to left.
   - **Example:**
   - const arr = [1, 2, 3, 4];
   - const sum = arr.reduceRight((acc, num) => acc + num, 0);

console.log(sum); // 10

8. **reverse()**
   - **Description:** Reverses the order of the elements in an array.
   - **Explanation:** Modifies the array in place by reversing the order of its elements.
   - **Example:**
   - const arr = [1, 2, 3];
   - arr.reverse();

console.log(arr); // [3, 2, 1]

9. **shift()**
   - **Description:** Removes the first element from an array and returns that element.

- **Explanation:** Removes the first element from the array, reducing its length by one, and returns the removed element.
- **Example:**
- const arr = [1, 2, 3];
- const firstElement = arr.shift();
- console.log(firstElement); // 1

console.log(arr); // [2, 3]

10. **slice()**
    - **Description:** Returns a shallow copy of a portion of an array into a new array object.
    - **Explanation:** Extracts a section of the array and returns it as a new array without modifying the original array.
    - **Example:**
    - const arr = [1, 2, 3, 4, 5];
    - const newArray = arr.slice(1, 3);

console.log(newArray); // [2, 3]

11. **some()**
    - **Description:** Checks if at least one element in the array passes the test implemented by the provided function.
    - **Explanation:** Returns true if at least one element passes the test implemented by the provided function; otherwise, it returns false.
    - **Example:**
    - const arr = [1, 2, 3];
    - const hasEven = arr.some((num) => num % 2 === 0);

console.log(hasEven); // true

12. **sort()**
    - **Description:** Sorts the elements of an array in place and returns the sorted array.
    - **Explanation:** Sorts the elements of the array according to the provided compare function. If no function is provided, elements are sorted as strings.
    - **Example:**

- const arr = [3, 1, 4, 2];
- arr.sort((a, b) => a - b);

console.log(arr); // [1, 2, 3, 4]

13. **splice()**
    - **Description:** Changes the contents of an array by removing or replacing existing elements and/or adding new elements.
    - **Explanation:** Modifies the array by removing, replacing, or adding elements at a specified index.
    - **Example:**
    - const arr = [1, 2, 3, 4];
    - arr.splice(2, 1, "newElement");

console.log(arr); // [1, 2, 'newElement', 4]

14. **toString()**
    - **Description:** Returns a string representing the specified array and its elements.
    - **Explanation:** Converts the array to a string, with each element separated by commas.
    - **Example:**
    - const arr = [1, 2, 3];
    - const str = arr.toString();

console.log(str); // "1, 2, 3"

15. **unshift()**
    - **Description:** Adds one or more elements to the beginning of an array and returns the new length of the array.
    - **Explanation:** Prepends new elements to the beginning of the array and returns the new length of the array.
    - **Example:**
    - const arr = [2, 3];
    - const newLength = arr.unshift(1);
    - console.log(newLength); // 3

console.log(arr); // [1, 2, 3]

16. **flat()**
   - **Description:** Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
   - **Explanation:** Flattens nested arrays into a single array up to the specified depth.
   - **Example:**
   - const arr = [1, [2, [3, [4]]]];
   - const flatArray = arr.flat(2);

console.log(flatArray); // [1, 2, 3, [4]]

17. **flatMap()**
   - **Description:** First maps each element using a mapping function, then flattens the result into a new array.
   - **Explanation:** Applies the provided function to each element and flattens the result into a new array.
   - **Example:**
   - const arr = [1, 2, 3];
   - const newArray = arr.flatMap((num) => [num, num * 2]);

console.log(newArray); // [1, 2, 2, 4, 3, 6]

18. **from()**
   - **Description:** Creates a new array instance from an array-like or iterable object.
   - **Explanation:** Converts an array-like or iterable object into a new array instance.
   - **Example:**
   - const arrayFromStr = Array.from("hello");

console.log(arrayFromStr); // ['h', 'e', 'l', 'l', 'o']

19. **isArray()**
   - **Description:** Determines whether the passed value is an array.
   - **Explanation:** Checks if the provided value is an array and returns true if it is, otherwise false.
   - **Example:**

- const isArray = Array.isArray([1, 2, 3]);

console.log(isArray); // true

20. **of()**
    - **Description:** Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.
    - **Explanation:** Creates a new array instance with the provided arguments as elements.
    - **Example:**
    - const newArray = Array.of(1, 2, 3);

console.log(newArray); // [1, 2, 3]

JavaScript Sort

Here's the markdown table with the provided list of sorting-related functions and methods:

Method	Description
`sort()`	Sorts the elements of an array in place and returns the sorted array.
`reverse()`	Reverses the order of the elements in an array.
`toSorted()`	Returns a new array with the elements sorted, without modifying the original array.
`toReversed()`	Returns a new array with the elements reversed, without modifying the original array.
`Math.min()`	Returns the smallest of zero or more numbers.
`Math.max()`	Returns the largest of zero or more numbers.

Explanations with Examples

1. **sort()**
    - **Description:** Sorts the elements of an array in place and returns the sorted array.
    - **Explanation:** Sorts the elements of the array according to the provided compare function. If no function is provided, elements are sorted as strings.

- If the return value is negative, a is sorted before b.
- If the return value is positive, b is sorted before a.
- If the return value is 0, no changes are done with the sort order of the two values.
- **Example:**
- // Sort numbers in ascending order
- const arr = [3, 1, 4, 2];
- arr.sort((a, b) => a - b);
- console.log(arr); // [1, 2, 3, 4]
-
- // Sort strings alphabetically
- const words = ["banana", "apple", "cherry"];
- words.sort();

console.log(words); // ['apple', 'banana', 'cherry']

2. **reverse()**
   - **Description:** Reverses the order of the elements in an array.
   - **Explanation:** Modifies the array in place by reversing the order of its elements.
   - **Example:**
   - const arr = [1, 2, 3];
   - arr.reverse();
   - console.log(arr); // [3, 2, 1]
   -
   - // Combining with sort
   - const words = ["banana", "apple", "cherry"];
   - words.sort();
   - words.reverse();

console.log(words); // ['cherry', 'banana', 'apple']

3. **toSorted()**

- **Description:** Returns a new array with the elements sorted, without modifying the original array.
- **Explanation:** Creates a sorted copy of the array, leaving the original array unchanged.
- **Example:**
- const arr = [3, 1, 4, 2];
- const sortedArr = arr.toSorted((a, b) => a - b);
- console.log(sortedArr); // [1, 2, 3, 4]

console.log(arr); // [3, 1, 4, 2] (original array remains unchanged)

4. **toReversed()**
    - **Description:** Returns a new array with the elements reversed, without modifying the original array.
    - **Explanation:** Creates a reversed copy of the array, leaving the original array unchanged.
    - **Example:**
    - const arr = [1, 2, 3];
    - const reversedArr = arr.toReversed();
    - console.log(reversedArr); // [3, 2, 1]

console.log(arr); // [1, 2, 3] (original array remains unchanged)

5. **Sorting Objects**
    - **Description:** Custom sorting of array elements based on object properties.
    - **Explanation:** Sorts objects within an array using a custom comparator function that compares specific properties.
    - **Example:**
    - const items = [
    -   { name: "apple", price: 50 },
    -   { name: "banana", price: 30 },
    -   { name: "cherry", price: 20 },
    -];
    - items.sort((a, b) => a.price - b.price);

- console.log(items);
- // [
- //   { name: 'cherry', price: 20 },
- //   { name: 'banana', price: 30 },
- //   { name: 'apple', price: 50 }

//]

6. **Numeric Sort**
   - **Description:** Sorting numbers in an array in numerical order.
   - **Explanation:** Sorts numeric values in ascending or descending order using a comparator function.
   - **Example:**
   - const numbers = [4, 2, 5, 1, 3];
   - numbers.sort((a, b) => a - b);

console.log(numbers); // [1, 2, 3, 4, 5]

7. **Random Sort**
   - **Description:** Randomly shuffling the elements of an array.
   - **Explanation:** Uses a comparator function to shuffle array elements in random order.
   - **Example:**
   - const arr = [1, 2, 3, 4, 5];
   - arr.sort(() => Math.random() - 0.5);

console.log(arr); // Random order, e.g., [3, 1, 4, 2, 5]

8. **Math.min()**
   - **Description:** Returns the smallest of zero or more numbers.
   - **Explanation:** Finds the minimum value among given numbers.
   - **Example:**
   - const min = Math.min(3, 1, 4, 2);

console.log(min); // 1

   - Math.min.apply(null, [1, 2, 3]) is equivalent to Math.min(1, 2, 3)
   - Example:

9. const min = Math.min.apply(null, [3, 1, 4, 2]);

console.log(min); // 1

10. **Math.max()**
    - **Description:** Returns the largest of zero or more numbers.
    - **Explanation:** Finds the maximum value among given numbers.
    - **Example:**
    - const max = Math.max(3, 1, 4, 2);

console.log(max); // 4

- Math.max.apply(null, [1, 2, 3]) is equivalent to Math.max(1, 2, 3).
- Example:

11. const max = Math.max.apply(null, [3, 1, 4, 2]);

console.log(max); // 4

12. **Homemade Min**
    - **Description:** Finding the minimum value in an array using a custom function.
    - **Explanation:** Uses array methods to find the smallest number in an array.
    - **Example:**
    - const arr = [3, 1, 4, 2];
    - const min = arr.reduce((a, b) => (a < b ? a : b));

console.log(min); // 1

13. **Homemade Max**
    - **Description:** Finding the maximum value in an array using a custom function.
    - **Explanation:** Uses array methods to find the largest number in an array.
    - **Example:**
    - const arr = [3, 1, 4, 2];
    - const max = arr.reduce((a, b) => (a > b ? a : b));

console.log(max); // 4

Sorting Object Arrays

- **Description:** Sorting arrays of objects based on one or more object properties.

- **Explanation:** To sort an array of objects, you provide a comparator function to the sort() method. The comparator function defines the sort order based on the properties of the objects. The function should return:
  - A negative value if the first argument is less than the second (should appear before the second in the sorted array).
  - Zero if the two arguments are considered equal (their order doesn't change).
  - A positive value if the first argument is greater than the second (should appear after the second in the sorted array).

Examples

1. **Sorting by a Single Property (e.g., by price):**

```
const items = [
  { name: "apple", price: 50 },
  { name: "banana", price: 30 },
  { name: "cherry", price: 20 },
];

// Sort by price (ascending)
items.sort((a, b) => a.price - b.price);
console.log(items);
// [
//   { name: 'cherry', price: 20 },
//   { name: 'banana', price: 30 },
//   { name: 'apple', price: 50 }
// ]

// Sort by price (descending)
items.sort((a, b) => b.price - a.price);
console.log(items);
// [
//   { name: 'apple', price: 50 },
//   { name: 'banana', price: 30 },
//   { name: 'cherry', price: 20 }
// ]
```

Sorting by Multiple Properties (e.g., by price, then by name):

```
const items = [
  { name: "apple", price: 30 },
  { name: "banana", price: 30 },
  { name: "cherry", price: 20 },
  { name: "apple", price: 20 },
];

// Sort by price first (ascending), then by name (ascending)
items.sort((a, b) => {
  if (a.price === b.price) {
    return a.name.localeCompare(b.name); // Sort by name if prices are equal
  }
  return a.price - b.price; // Otherwise, sort by price
});

console.log(items);
// [
//   { name: 'apple', price: 20 },
//   { name: 'cherry', price: 20 },
//   { name: 'apple', price: 30 },
//   { name: 'banana', price: 30 }
// ]
```

Explanation of the Code

The line return a.name.localeCompare(b.name); is used to compare two strings (in this case, the name properties of the objects a and b). This method is part of

JavaScript's String prototype and returns a number that indicates the order of the two strings relative to each other. Specifically, it works as follows:

- **Returns -1** if a.name comes before b.name in lexicographical order (i.e., alphabetical order).
- **Returns 1** if a.name comes after b.name in lexicographical order.
- **Returns 0** if a.name and b.name are identical.

Explanation in the Context of Sorting:

- **If the price values of a and b are equal** (checked by a.price === b.price), the sort function doesn't know which object should come first based on price. So, it then compares the name properties.
- The localeCompare method ensures that the names are sorted alphabetically. If a.name comes before b.name, the method returns -1, meaning a should come before b in the sorted array. If a.name comes after b.name, it returns 1, meaning b should come before a.

In your example, when the prices are the same, the sorting falls back to comparing the names alphabetically:

- For { name: 'apple', price: 30 } and { name: 'banana', price: 30 }, since apple comes before banana alphabetically, localeCompare returns -1, so apple stays before banana.
- For { name: 'apple', price: 20 } and { name: 'cherry', price: 20 }, since apple comes before cherry alphabetically, localeCompare returns -1, so apple stays before cherry.

This is why the final sorted array has items ordered by price first and, for equal prices, by name alphabetically.

JavaScript Array Map

- **Description:** The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.
- **Explanation:** The map() method iterates over each element in the original array, applies the provided function to each element, and returns a new array containing the results of those function calls. The original array remains unchanged. It's a powerful method when you want to transform elements of an array, such as applying a mathematical operation to each element or converting data formats.
- **Syntax:**
- let newArray = array.map(function (currentValue, index, array) {

- // return element for newArray

});

  - **currentValue:** The current element being processed in the array.
  - **index (optional):** The index of the current element being processed.
  - **array (optional):** The array map was called upon.

Examples

1. **Basic Example: Squaring Numbers**
2. const numbers = [1, 2, 3, 4];
3. const squared = numbers.map((num) => num * num);

console.log(squared); // [1, 4, 9, 16]

Explanation: Here, the map() method takes each element from the numbers array, squares it, and returns a new array squared with the squared values.

4. **Converting an Array from Objects**
5. const users = [
6. { firstName: "John", lastName: "Doe" },
7. { firstName: "Jane", lastName: "Smith" },
8. { firstName: "Emily", lastName: "Jones" },
9.];
10.
11. const fullNames = users.map((user) => `${user.firstName} ${user.lastName}`);

console.log(fullNames); // ['John Doe', 'Jane Smith', 'Emily Jones']

Explanation: The map() method is used here to transform an array of user objects into an array of full names by concatenating the firstName and lastName properties of each object.

12. **Mapping Indexes**
13. const numbers = [10, 20, 30];
14. const indices = numbers.map((num, index) => index);

console.log(indices); // [0, 1, 2]

Explanation: In this example, map() is used to create a new array of the indices of the original array. The function returns the index for each element.

15. **Transforming Data**

```
const temperaturesInCelsius = [0, 10, 20, 30];
const temperaturesInFahrenheit = temperaturesInCelsius.map(
  (celsius) => (celsius * 9) / 5 + 32
);
console.log(temperaturesInFahrenheit); // [32, 50, 68, 86]
```

Explanation: This example demonstrates using map() to convert an array of temperatures from Celsius to Fahrenheit.

Key Points

- map() always returns a new array without modifying the original array.
- The number of elements in the new array will always match the number of elements in the original array.
- It's commonly used for data transformation tasks where you need to perform the same operation on every element of an array and store the results in a new array.

map() is widely used in functional programming paradigms in JavaScript, especially in scenarios involving React or other libraries where immutability and transformations of data are common.

JavaScript Array Filter

- **Description:** The filter() method creates a new array with all elements that pass the test implemented by the provided function. In other words, it filters out elements from the original array that do not satisfy the condition specified in the callback function.
- **Explanation:** The filter() method is used to create a subset of an array based on a condition. The callback function you provide should return true to keep the element in the new array, or false to exclude it. The original array remains unchanged.
- **Syntax:**

```
let newArray = array.filter(function (currentValue, index, array) {
  // return true to keep the element, false otherwise
});
```

  - **currentValue:** The current element being processed in the array.
  - **index (optional):** The index of the current element being processed.
  - **array (optional):** The array filter was called upon.

Examples

1. **Filtering Even Numbers**
2. const numbers = [1, 2, 3, 4, 5, 6];
3. const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // [2, 4, 6]

Explanation: The filter() method is used here to create a new array evenNumbers that contains only the elements from the original array numbers that are even.

4. **Filtering Objects Based on a Property**
5. const people = [
6.   { name: "John", age: 20 },
7.   { name: "Jane", age: 25 },
8.   { name: "Emily", age: 30 },
9.];
10.
11. const adults = people.filter((person) => person.age >= 25);
12. console.log(adults);
13. // [
14. //   { name: 'Jane', age: 25 },
15. //   { name: 'Emily', age: 30 }

//]

Explanation: In this example, filter() is used to create a new array adults that includes only the people who are 25 years old or older.

16. **Filtering Out Falsy Values**
17. const mixedArray = [0, "hello", false, 42, "", null, "world"];
18. const truthyValues = mixedArray.filter(Boolean);

console.log(truthyValues); // ['hello', 42, 'world']

Explanation: By passing the built-in Boolean function to filter(), the method filters out all falsy values (false, 0, '', null, undefined, and NaN) from the array.

19. **Filtering Unique Values**
20. const numbers = [1, 2, 2, 3, 4, 4, 5];

```
const uniqueNumbers = numbers.filter(
  (num, index, arr) => arr.indexOf(num) === index
);
console.log(uniqueNumbers); // [1, 2, 3, 4, 5]
```

Explanation: This example uses filter() to remove duplicate values from an array, resulting in an array of unique numbers. The indexOf method returns the index of the **first occurrence** of the element num in the array arr.

Detailed Explanation:

i. **Array numbers**:
This is the original array containing the numbers, including duplicates: [1, 2, 2, 3, 4, 4, 5].

ii. **filter Method**:
The filter method creates a new array with all elements that pass the test implemented by the provided function. In this case, the function checks if the current element's index matches the first occurrence of that element in the array.

iii. **Callback Function**:

```
(num, index, arr) => arr.indexOf(num) === index;
```

This is the callback function passed to the filter method. It takes three arguments:

- num: The current element being processed in the array.
- index: The index of the current element.
- arr: The array that filter was called on (in this case, numbers).

iv. **indexOf(num)**: The indexOf method returns the index of the **first occurrence** of the element num in the array arr.

v. **Comparison arr.indexOf(num) === index**:

- For each element in the array, the code checks if the index of its first occurrence (arr.indexOf(num)) is the same as its current index (index).
- If they match, it means that the element has not appeared earlier in the array, so it is considered unique up to this point.
- If they don't match, it means the element has already appeared earlier in the array, so it's a duplicate and is filtered out.

Step-by-Step Execution:

  - **Iteration 1**: num = 1, index = 0, arr.indexOf(1) = 0
    0 === 0 → true → include 1 in uniqueNumbers.
  - **Iteration 2**: num = 2, index = 1, arr.indexOf(2) = 1
    1 === 1 → true → include 2 in uniqueNumbers.
  - **Iteration 3**: num = 2, index = 2, arr.indexOf(2) = 1
    1 === 2 → false → exclude this 2 from uniqueNumbers.
  - **Iteration 4**: num = 3, index = 3, arr.indexOf(3) = 3
    3 === 3 → true → include 3 in uniqueNumbers.
  - **Iteration 5**: num = 4, index = 4, arr.indexOf(4) = 4
    4 === 4 → true → include 4 in uniqueNumbers.
  - **Iteration 6**: num = 4, index = 5, arr.indexOf(4) = 4
    4 === 5 → false → exclude this 4 from uniqueNumbers.
  - **Iteration 7**: num = 5, index = 6, arr.indexOf(5) = 6
    6 === 6 → true → include 5 in uniqueNumbers.

Result:

After filtering, uniqueNumbers contains [1, 2, 3, 4, 5], which is the original array without any duplicates.

Key Points

- filter() returns a new array containing only the elements that satisfy the condition specified in the callback function.
- The original array is not modified.
- The new array may contain fewer elements than the original array, or it may be empty if no elements pass the test.
- Commonly used in scenarios where you need to extract or exclude specific elements from an array based on a certain condition.

filter() is a powerful and versatile method for creating subsets of data, often used in data processing, search functionalities, and conditions-based rendering in web development frameworks like React.

JavaScript Array Reduce

- **Description:** The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. This method can be used to aggregate array data, like summing all elements, calculating a product, or flattening an array.

- **Explanation:** The reduce() method processes each element of the array (from left to right) and accumulates the result into a single value. The callback function you pass to reduce() receives two main arguments:
  - The **accumulator**, which accumulates the callback's return values.
  - The **current value**, which is the current element being processed in the array.

Additionally, reduce() can take an initial value as a second argument, which is used as the initial accumulator value. If no initial value is provided, reduce() will use the first element of the array as the initial accumulator and start the iteration from the second element.

- **Syntax:**
- array.reduce(function (accumulator, currentValue, index, array) {
-   // return the new accumulator value

}, initialValue);

  - **accumulator:** The accumulated result from the previous iteration or the initial value if provided.
  - **currentValue:** The current element being processed in the array.
  - **index (optional):** The index of the current element being processed.
  - **array (optional):** The array reduce was called upon.
  - **initialValue (optional):** The initial value to start the accumulation.

Examples

1. **Summing an Array of Numbers**
2. const numbers = [1, 2, 3, 4, 5];
3. const sum = numbers.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // 15

Explanation: Here, reduce() adds each number in the numbers array to the accumulator (acc), starting from 0, to produce the total sum of the array.

4. **Flattening a Nested Array**
5. const nestedArray = [
6.   [1, 2],
7.   [3, 4],
8.   [5, 6],

9.];

10. const flatArray = nestedArray.reduce((acc, curr) => acc.concat(curr), []);

console.log(flatArray); // [1, 2, 3, 4, 5, 6]

Explanation: This example uses reduce() to flatten a 2D array into a single array by concatenating each sub-array to the accumulator.

11. **Counting Instances of Values in an Array**

12. const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];

13. const count = fruits.reduce((acc, fruit) => {

14.   acc[fruit] = (acc[fruit] || 0) + 1;

15.   return acc;

16. }, {});

console.log(count); // { apple: 3, banana: 2, orange: 1 }

Explanation: In this example, reduce() is used to create an object that counts the occurrences of each fruit in the fruits array.

17. **Finding the Maximum Value in an Array**

18. const numbers = [10, 5, 20, 3, 100];

19. const max = numbers.reduce(

20.   (acc, curr) => (curr > acc ? curr : acc),

21.   numbers[0]

22.);

console.log(max); // 100

Explanation: The reduce() method iterates through the numbers array, comparing each value to the current maximum (acc) and updating it if a larger value is found.

23. **Grouping Objects by a Property**

24. const people = [

25.   { name: "Alice", age: 21 },

26.   { name: "Bob", age: 25 },

27.   { name: "Charlie", age: 21 },

28.];

29.

```
const groupedByAge = people.reduce((acc, person) => {
  const age = person.age;
  if (!acc[age]) {
    acc[age] = [];
  }
  acc[age].push(person);
  return acc;
}, {});

console.log(groupedByAge);
// {
//   21: [{ name: 'Alice', age: 21 }, { name: 'Charlie', age: 21 }],
//   25: [{ name: 'Bob', age: 25 }]
// }
```

Explanation: This example uses reduce() to group an array of objects by a specific property (age in this case). The result is an object where the keys are ages and the values are arrays of people who have that age.

Key Points

- reduce() is a versatile method that can be used to perform complex data transformations and aggregations.
- It always returns a single value, which can be of any type (number, string, object, array, etc.).
- It is particularly useful for operations like summing arrays, counting elements, flattening arrays, and more.
- You should always provide an initial value for the accumulator to avoid unexpected results, especially with empty arrays.

reduce() is a powerful method in JavaScript, especially in functional programming paradigms. It can significantly reduce the amount of code needed to perform aggregations and transformations on arrays.

JavaScript Array Every

- **Description:** The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value (true or false).
- **Explanation:** The every() method iterates over each element of the array and applies the provided callback function to test a condition. If the callback function returns true for every element, every() returns true. If the callback returns false for any element, every() immediately returns false and stops iterating. This method is useful for checking if all elements in an array meet a certain condition.
- **Syntax:**
- array.every(function (currentValue, index, array) {
-   // return true or false based on the condition

});

  - **currentValue:** The current element being processed in the array.
  - **index (optional):** The index of the current element being processed.
  - **array (optional):** The array every was called upon.

Examples

1. **Checking if All Elements are Even**
2. const numbers = [2, 4, 6, 8];
3. const allEven = numbers.every((num) => num % 2 === 0);

console.log(allEven); // true

Explanation: The every() method is used here to check if all elements in the numbers array are even. Since all elements pass the test (num % 2 === 0), every() returns true.

4. **Checking if All Elements are Greater Than a Value**
5. const numbers = [10, 20, 30, 40];
6. const allGreaterThanFive = numbers.every((num) => num > 5);

console.log(allGreaterThanFive); // true

Explanation: In this example, every() checks if all elements in the numbers array are greater than 5. Since all elements satisfy the condition, every() returns true.

7. **Checking if All Elements are Strings**
8. const mixedArray = ["hello", "world", 42, "JavaScript"];

9. const allStrings = mixedArray.every((item) => typeof item === "string");

console.log(allStrings); // false

Explanation: Here, every() checks if all elements in mixedArray are strings. Since one element is a number (42), every() returns false.

10. **Using every() with an Empty Array**
11. const emptyArray = [];
12. const result = emptyArray.every((item) => item > 0);

console.log(result); // true

Explanation: When every() is called on an empty array, it returns true by default. This is because there are no elements in the array to fail the test, so it is considered that "all" elements (which are none) pass the test.

Key Points

- every() returns true only if the provided function returns true for every element in the array.
- If the function returns false for any element, every() immediately stops and returns false.
- It does not modify the original array.
- It is often used for validation or condition checks across all elements in an array.
- If every() is called on an empty array, it will always return true.

The every() method is useful in scenarios where you need to ensure that all elements in an array meet a particular condition, such as validating data or checking constraints.

JavaScript Array Some

- **Description:** The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value (true or false).
- **Explanation:** The some() method iterates over each element in the array and applies the provided callback function to test a condition. If the callback function returns true for any element, some() immediately returns true and stops further iterations. If no elements pass the test, it returns false. This method is useful for checking if any element in an array meets a specific condition.
- **Syntax:**
- array.some(function (currentValue, index, array) {
-   // return true or false based on the condition

});

- **currentValue:** The current element being processed in the array.
- **index (optional):** The index of the current element being processed.
- **array (optional):** The array some was called upon.

Examples

1. **Checking if Any Element is Even**
2. const numbers = [1, 3, 5, 7, 8];
3. const hasEven = numbers.some((num) => num % 2 === 0);

console.log(hasEven); // true

Explanation: The some() method checks if any element in the numbers array is even. Since the number 8 is even, some() returns true.

4. **Checking if Any Element is Greater Than a Value**
5. const numbers = [1, 2, 3, 4, 5];
6. const anyGreaterThanThree = numbers.some((num) => num > 3);

console.log(anyGreaterThanThree); // true

Explanation: Here, some() checks if any element in the numbers array is greater than 3. Since there are elements that satisfy this condition, some() returns true.

7. **Checking if Any Element is a String**
8. const mixedArray = [10, "hello", 42, "world"];
9. const hasString = mixedArray.some((item) => typeof item === "string");

console.log(hasString); // true

Explanation: The some() method is used here to check if any element in mixedArray is a string. Since the array contains strings like 'hello' and 'world', some() returns true.

10. **Using some() with an Empty Array**
11. const emptyArray = [];
12. const result = emptyArray.some((item) => item > 0);

console.log(result); // false

Explanation: When some() is called on an empty array, it returns false because there are no elements to test, meaning no element can pass the test.

Key Points

- some() returns true if at least one element passes the test implemented by the provided function.
- If no elements pass the test, some() returns false.
- It does not modify the original array.
- some() is often used when you need to check for the existence of at least one element that meets a certain condition.
- If some() is called on an empty array, it will always return false.

The some() method is particularly useful in scenarios where you want to verify that at least one element in an array meets a specific condition, such as checking for the presence of a specific value or type within an array.

JavaScript Array.from()

- **Description:** The Array.from() method creates a new, shallow-copied Array instance from an array-like or iterable object. It is particularly useful for converting structures like NodeList, arguments, or any iterable (like a Set or Map) into a true array.
- **Explanation:** Array.from() takes an array-like or iterable object as its first argument and returns a new array. Optionally, it can take a second argument, which is a mapping function that will be applied to each element before being added to the array. This method is especially useful when working with DOM elements, strings, or any other iterable that you need to convert into an array for further processing.
- **Syntax:**

Array.from(arrayLike, mapFn, thisArg);

  - **arrayLike:** The array-like or iterable object to convert to an array.
  - **mapFn (optional):** A function to call on every element of the array before adding it to the new array.
  - **thisArg (optional):** A value to use as this when executing mapFn.

Examples

1. **Converting a String to an Array**
2. const str = "hello";
3. const strArray = Array.from(str);

console.log(strArray); // ['h', 'e', 'l', 'l', 'o']

Explanation: The Array.from() method converts the string 'hello' into an array of characters.

4. **Converting a Set to an Array**
5. const set = new Set(["apple", "banana", "cherry"]);
6. const array = Array.from(set);

console.log(array); // ['apple', 'banana', 'cherry']

Explanation: Here, Array.from() is used to convert a Set into an array. This is useful when you need to manipulate or access the elements of the Set as an array.

7. **Mapping with Array.from()**
8. const numbers = [1, 2, 3];
9. const doubled = Array.from(numbers, (num) => num * 2);

console.log(doubled); // [2, 4, 6]

Explanation: In this example, Array.from() not only converts the array-like numbers into an array but also applies a mapping function that doubles each number.

10. **Converting a NodeList to an Array**
11. const nodeList = document.querySelectorAll("p");
12. const nodesArray = Array.from(nodeList);

console.log(nodesArray); // Array of <p> elements

Explanation: Array.from() is used here to convert a NodeList (which is returned by querySelectorAll) into an array, allowing for array methods like map, filter, or forEach to be used on the DOM elements.

13. **Using Array.from() with Arguments Object**
14. function sum() {
15.   const argsArray = Array.from(arguments);
16.   return argsArray.reduce((sum, num) => sum + num, 0);
17. }
18.

console.log(sum(1, 2, 3)); // 6

Explanation: This example demonstrates how Array.from() can be used to convert the arguments object, which is array-like but not an actual array, into a true array. This allows you to then apply array methods like reduce to calculate the sum of all arguments.

Key Points

- Array.from() is a static method of Array, meaning it is called on the Array class itself, not on an instance of an array.
- It is often used to convert array-like objects or iterables into a real array.
- You can also use it with a mapping function, similar to how Array.prototype.map() works, to transform each element in the array-like object before adding it to the new array.
- Useful for working with DOM collections, strings, and other iterable objects that are not true arrays.

Array.from() is a versatile and powerful method in JavaScript, especially when working with data structures that are not inherently arrays but need to be treated as arrays for further manipulation.

JavaScript Array keys()

The keys() method in JavaScript returns a new Array Iterator object that contains the keys (or indices) for each element in the array. This method is useful when you want to iterate over the indices of an array.

Syntax

```
array.keys();
```

Example

Here's a simple example to demonstrate how keys() works:

```
const fruits = ["apple", "banana", "mango"];
const iterator = fruits.keys();

for (let key of iterator) {
  console.log(key);
}
```

Explanation

- In the example above, we have an array fruits with three elements.
- The keys() method returns an iterator object that contains the keys (indices) of the array elements.
- We then use a for...of loop to iterate over the keys and log them to the console.

Output

The output of the code will be:

```
0

1

2
```

Each number corresponds to the index of each element in the fruits array.

Additional Example

You can also use the keys() method in combination with other array methods, like forEach:

```
const colors = ["red", "green", "blue"];

const keys = colors.keys();


keys.forEach((key) => {

  console.log(`Index: ${key}, Value: ${colors[key]}`);

});
```

Output

```
Index: 0, Value: red

Index: 1, Value: green

Index: 2, Value: blue
```

In this example, we used forEach to iterate through the keys and access the corresponding values in the colors array.

JavaScript Array Entries

The entries() method in JavaScript returns a new Array Iterator object that contains key/value pairs for each index in the array. Each pair consists of the index (the key) and the value at that index in the array. This method is useful when you want to iterate over both the indices and the values of an array.

Syntax

```
array.entries();
```

Example

Here's a simple example to demonstrate how entries() works:

```
const fruits = ["apple", "banana", "mango"];

const iterator = fruits.entries();
```

```
for (let entry of iterator) {
  console.log(entry);
}
```

Explanation

- In the example above, we have an array fruits with three elements.
- The entries() method returns an iterator object that contains key/value pairs for each element in the array.
- We then use a for...of loop to iterate over these key/value pairs and log them to the console.

Output

The output of the code will be:

```
[0, "apple"]
[1, "banana"]
[2, "mango"]
```

Each pair represents the index (first value) and the corresponding array element (second value).

Using Destructuring

You can also use destructuring to make it easier to work with the key/value pairs:

```
const colors = ["red", "green", "blue"];
const iterator = colors.entries();

for (let [index, value] of iterator) {
  console.log(`Index: ${index}, Value: ${value}`);
}
```

Output

```
Index: 0, Value: red
Index: 1, Value: green
Index: 2, Value: blue
```

In this example, we use array destructuring within the for...of loop to directly extract the index and value from each entry.

Practical Example: Looping Over Entries

The entries() method can be particularly useful when you need to loop over both the index and value of array elements:

```
const students = ["Alice", "Bob", "Charlie"];
const studentEntries = students.entries();


for (let [index, name] of studentEntries) {
  console.log(`Student ${index + 1}: ${name}`);
}
```

Output

```
Student 1: Alice
Student 2: Bob
Student 3: Charlie
```

This example shows how you can use the index and value to create more informative output.

The entries() method is useful when both the index and value are needed simultaneously during iteration.

JavaScript Array with() Method

- Array with() method as a safe way to update elements in an array without altering the original array. Example:

```
const months = ["Januar", "Februar", "Mar", "April"];
const myMonths = months.with(2, "March");
```

↥ Go to Top

Chapter-08: JavaScript Date Object and Math Object

JavaScript Date Object: An In-Depth Guide with Real Use Cases

- The JavaScript Date object is used to work with dates and times. It provides methods for getting and setting year, month, day, hour, minute, second, and millisecond values of a date object. It also has methods to format and manipulate dates.

Common Uses

- Displaying the current date and time.
- Calculating time intervals (e.g., age, countdowns).
- Scheduling events.
- Formatting dates for user interfaces.

Creating Date Objects

There are four main ways to create a Date object in JavaScript:

1. Creating a New Date Object

Description: This method creates a new Date object representing the current date and time.

Code:

```
let currentDate = new Date();
console.log("Current Date and Time:", currentDate);
```

Explanation:

- The new Date() constructor creates a new Date object with the current date and time.
- The currentDate object holds the exact date and time at the moment of creation.

Use Case: This can be used to display the current date and time on a webpage, such as in a footer or a header.

2. Creating a Date Object with a Specific Date and Time

Description: This method creates a Date object for a specific date and time using a date string.

Code:

```
let specificDate = new Date("August 13, 2024 15:30:00");
console.log("Specific Date and Time:", specificDate);
```

Explanation:

- You can create a Date object using a date string.
- The string must be in a format recognized by the JavaScript Date object, such as "August 13, 2024 15:30:00".

Use Case: This method is useful when scheduling events or reminders on specific dates and times.

3. Creating a Date Object Using Date Components

Description: This method allows you to create a Date object by specifying the year, month, day, hour, minute, second, and millisecond components. You can use between two and seven parameters.

Code:

- **Seven Parameters:**
- let fullDate = new Date(2018, 11, 24, 10, 33, 30, 0);

console.log("Full Date:", fullDate);

- **Six Parameters:**
- let dateWithoutMilliseconds = new Date(2018, 11, 24, 10, 33, 30);

console.log("Date Without Milliseconds:", dateWithoutMilliseconds);

- **Five Parameters:**
- let dateWithoutSeconds = new Date(2018, 11, 24, 10, 33);

console.log("Date Without Seconds:", dateWithoutSeconds);

- **Four Parameters:**
- let dateWithoutMinutes = new Date(2018, 11, 24, 10);

console.log("Date Without Minutes:", dateWithoutMinutes);

- **Three Parameters:**
- let dateWithoutHours = new Date(2018, 11, 24);

console.log("Date Without Hours:", dateWithoutHours);

- **Two Parameters:**
- let dateWithoutDay = new Date(2018, 11);

console.log("Date Without Day:", dateWithoutDay);

Explanation:

- The Date constructor can accept from two to seven parameters, representing year, month (0-indexed), day, hour, minute, second, and millisecond.
- The example code demonstrates how omitting certain parameters defaults them to 0 or 1, depending on the position (e.g., day defaults to 1, hour defaults to 0).

Use Case: This approach is useful for programmatically generating dates based on user inputs or specific criteria, with flexibility in how much detail you need to specify.

4. Creating a Date Object Using Timestamps

Description: This method creates a Date object from a timestamp, which is the number of milliseconds since January 1, 1970.

Code:

```
let timestamp = 1658329800000;
let dateFromTimestamp = new Date(timestamp);
console.log("Date from Timestamp:", dateFromTimestamp);
```

Explanation:

- A timestamp represents the number of milliseconds since January 1, 1970 (the Unix Epoch).
- The Date constructor can convert a timestamp into a human-readable date.

Use Case: This is commonly used in databases and APIs where dates are stored as timestamps.

Date Formats

JavaScript supports multiple formats for dates, but the most reliable is the ISO 8601 format.

1. ISO Date Format

Description: The ISO 8601 format is a standardized way to represent dates and times, often used in APIs and databases.

Code:

```
let isoDate = new Date("2024-08-13T15:30:00Z");
console.log("ISO Date:", isoDate);
```

Explanation:

- The ISO 8601 format (YYYY-MM-DDTHH:MM:SSZ) is the most universally recognized format.
- T separates the date and time, and Z denotes UTC time.

Use Case: ISO format is ideal for passing dates between systems or when consistency is critical, such as in API responses.

2. Short Date Format

Description: The short date format is commonly used in the United States and follows the MM/DD/YYYY pattern.

Code:

```
let shortDate = new Date("08/13/2024");
```

```
console.log("Short Date:", shortDate);
```

Explanation:

- The short date format MM/DD/YYYY is common in the U.S.
- It can be convenient for local applications, but less ideal for global use due to format variations.

Use Case: Use short dates in user interfaces where the format is clearly understood by your audience.

3. Long Date Format

Description: The long date format uses the full name of the month and day, making it easy to read.

Code:

```
let longDate = new Date("August 13, 2024");
console.log("Long Date:", longDate);
```

Explanation:

- The long date format (Month DD, YYYY) is user-friendly and easy to read.
- It can be used in contexts where dates need to be clearly communicated to users.

Use Case: This format is perfect for displaying dates in reports, articles, or any content aimed at human readers.

Getting and Setting Date Components

1. Getting Date Components

Description: These methods allow you to retrieve individual components of a date, such as the year, month, or day.

Code:

```
let date = new Date("August 13, 2024 15:30:00");
let year = date.getFullYear();
let month = date.getMonth() + 1; // Month is zero-indexed
let day = date.getDate();
console.log(`Year: ${year}, Month: ${month}, Day: ${day}`);
```

Explanation:

- getFullYear() returns the year.

- getMonth() returns the month (0-indexed).
- getDate() returns the day of the month.

Use Case: These methods are useful for extracting specific components from a date, such as in applications where you need to display or process individual date parts.

2. Setting Date Components

Description: These methods allow you to set individual components of a date, modifying the year, month, or day as needed.

Code:

```
let date = new Date("August 13, 2024 15:30:00");
date.setFullYear(2025);
date.setMonth(11); // December (0-indexed)
date.setDate(25);
console.log("Updated Date:", date);
```

Explanation:

- setFullYear(), setMonth(), and setDate() allow you to modify the respective components of a date object.
- **Note:** Changing one component does not affect others unless the new value overflows (e.g., setting the day to 32 will change the month).

Use Case: Use these methods when you need to adjust dates based on user input or specific logic, such as rescheduling events.

Date Calculations

1. Adding or Subtracting Days

Description: This method allows you to add or subtract days from a date object.

Code:

```
let today = new Date();
let fiveDaysLater = new Date();
fiveDaysLater.setDate(today.getDate() + 5);
console.log("Five Days Later:", fiveDaysLater);
```

Explanation:

- setDate() can add days by modifying the current date.
- This method automatically handles month and year transitions.

Use Case: This is useful for creating deadlines, reminders, or countdowns.

2. Date Difference

Description: This method calculates the difference between two dates, typically to determine the number of days between them.

Code:

```
let startDate = new Date("August 1, 2024");
let endDate = new Date("August 13, 2024");
let differenceInTime = endDate.getTime() - startDate.getTime();
let differenceInDays = differenceInTime / (1000 * 3600 * 24);
console.log(`Difference in Days: ${differenceInDays}`);
```

Explanation:

- getTime() returns the time in milliseconds.
- Subtracting two dates gives the difference in milliseconds.
- Dividing by (1000 * 3600 * 24) converts milliseconds to days.

Use Case: This is essential for applications that require date comparisons, such as tracking project timelines or calculating the duration of an event.

Formatting Dates

1. toLocaleDateString()

Description: This method formats a date according to the locale-specific conventions, allowing for customized date displays.

Code:

```
let currentDate = new Date();
let formattedDate = currentDate.toLocaleDateString("en-US", {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
});
console.log("Formatted Date:", formattedDate);
```

Explanation:

- toLocaleDateString() formats the date according to locale-specific conventions.
- The options object allows customization of the output (e.g., full weekday name, numeric year, etc.).
- ১ম Parameter **en-BD** দিলে Bangladesh Time Zone এ দেখাবে, কিন্তু English Font এ দেখাবে। আর **bn-BD** দিলে Bangla Font এ দেখাবে।

Use Case: This method is ideal for presenting dates in a user-friendly and localized manner, especially in international applications.

2. toISOString()

Description: This method returns a date object as a string in the ISO 8601 format, which is commonly used for data storage and transfer.

Code:

```
let currentDate = new Date();
let isoString = currentDate.toISOString();
console.log("ISO String:", isoString);
```

Explanation:

- toISOString() returns a date as a string in ISO 8601 format (YYYY-MM-DDTHH:MM:SSZ).
- It's particularly useful for storing dates in databases or transferring data between systems.

Use Case: Use this method when you need a consistent and standard format for dates across different environments or systems.

Date Comparisons

1. Comparing Dates

Description: This method demonstrates how to compare two dates to determine which is earlier, later, or if they are the same.

Code:

```
let date1 = new Date("August 13, 2024");
let date2 = new Date("August 14, 2024");

if (date1 > date2) {
  console.log("Date1 is later than Date2");
} else if (date1 < date2) {
  console.log("Date1 is earlier than Date2");
} else {
  console.log("Date1 is the same as Date2");
}
```

Explanation:

- Date objects can be compared directly using comparison operators (>, <, ===).
- The comparison is based on the internal timestamp value.

Use Case: This is useful for sorting dates, checking deadlines, or determining if a specific date has passed.

Working with UTC Dates

1. Creating and Converting UTC Dates

Description: This method shows how to create a date object in UTC and convert it to a readable UTC string.

Code:

```
let utcDate = new Date(Date.UTC(2024, 7, 13, 12, 0, 0));
console.log("UTC Date:", utcDate.toUTCString());
```

Explanation:

- Date.UTC() creates a date in UTC time.
- toUTCString() converts a date object to a string, representing the date in UTC format.

Use Case: This is essential for applications that operate across different time zones, ensuring that dates and times are consistent globally.

JavaScript Math Object

Table of Contents

- The Math object in JavaScript is a built-in object that provides a range of mathematical functions and constants. It is not a constructor, meaning you don't need to create an instance of Math. Instead, you can directly use its methods and properties to perform various mathematical operations.

- The Math object in JavaScript is a global object that provides various properties and methods for mathematical constants and functions. You don't need to instantiate the Math object, as it is available globally. You can use its properties to access constants like Math.PI, and methods like Math.abs() to perform operations.

Example:

// Accessing the value of PI

console.log(Math.PI); // Output: 3.141592653589793

Explanation:

- **Math.PI**: This property returns the value of π (Pi), which is approximately 3.14159.

Commonly Used Math Methods

Here are some of the most commonly used methods provided by the Math object:

1. **Math.abs(x)** - Returns the absolute value of x.
2. **Math.ceil(x)** - Rounds x up to the nearest integer.
3. **Math.floor(x)** - Rounds x down to the nearest integer.
4. **Math.max(x, y, z, ...)** - Returns the largest of zero or more numbers.
5. **Math.min(x, y, z, ...)** - Returns the smallest of zero or more numbers.
6. **Math.random()** - Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).
7. **Math.round(x)** - Rounds x to the nearest integer.
8. **Math.sqrt(x)** - Returns the square root of x.

Explanation:

- **Math.abs(x)**: Converts any negative number to positive. If the number is already positive, it returns the number as it is.
- **Math.ceil(x)**: Rounds a number upward to its nearest integer.
- **Math.floor(x)**: Rounds a number downward to its nearest integer.
- **Math.max(x, y, z, ...)**: Takes any number of arguments and returns the maximum value.

- **Math.min(x, y, z, ...)**: Takes any number of arguments and returns the minimum value.
- **Math.random()**: Generates a random floating-point number between 0 (inclusive) and 1 (exclusive).
- **Math.round(x)**: Rounds a number to the nearest integer.
- **Math.sqrt(x)**: Computes the square root of a given number.

Examples with Code

Let's dive into some examples to understand how these methods work:

1. Math.abs(x)

```
let negativeNumber = -10;
let absoluteValue = Math.abs(negativeNumber);
console.log(absoluteValue); // Output: 10
```

Explanation: The Math.abs() method converts -10 into 10 by removing the negative sign.

2. Math.ceil(x)

```
let number = 4.3;
let roundedUp = Math.ceil(number);
console.log(roundedUp); // Output: 5
```

Explanation: The Math.ceil() method rounds 4.3 up to 5.

3. Math.floor(x)

```
let number = 4.8;
let roundedDown = Math.floor(number);
console.log(roundedDown); // Output: 4
```

Explanation: The Math.floor() method rounds 4.8 down to 4.

4. Math.random()

```
let randomNum = Math.random();
console.log(randomNum); // Output: A random number between 0 and 1 (e.g., 0.5343)
```

Explanation: The Math.random() method generates a random floating-point number between 0 and 1.

5. Math.sqrt(x)

let squareRoot = Math.sqrt(16);

console.log(squareRoot); // Output: 4

Explanation: The Math.sqrt() method returns 4 as the square root of 16.

Comprehensive Table of Math Methods

Below is a table summarizing some key methods of the JavaScript Math object:

Method	Description	Example	Output
Math.abs(x)	Returns the absolute value of x	Math.abs(-5)	5
Math.ceil(x)	Rounds x up to the nearest integer	Math.ceil(4.2)	5
Math.floor(x)	Rounds x down to the nearest integer	Math.floor(4.8)	4
Math.max(x, y)	Returns the highest value among the arguments	Math.max(1, 2, 3)	3
Math.min(x, y)	Returns the lowest value among the arguments	Math.min(1, 2, 3)	1
Math.pow(x, y)	Returns x to the power of y	Math.pow(2, 3)	8
Math.random()	Returns a random number between 0 (inclusive) and 1 (exclusive)	Math.random()	Varies
Math.round(x)	Rounds x to the nearest integer	Math.round(4.5)	5
Math.sqrt(x)	Returns the square root of x	Math.sqrt(25)	5
Math.trunc(x)	Returns the integer part of x by removing the fractional part	Math.trunc(4.9)	4

↥ Go to Top

Chapter-09: JavaScript Iterables, Sets, Set Methods, Map and Map Methods

JavaScript Iterables

Table of Contents

What Are Iterables?

- In JavaScript, an iterable is an object that can be iterated over, meaning you can loop through its elements one by one. Iterables are a fundamental concept in JavaScript, allowing for easy access and manipulation of sequences of data, such as arrays, strings, and more.
- An iterable is any object that has a method with the key [Symbol.iterator]. This method returns an iterator, which is an object that defines how to iterate over the iterable's elements. The most common iterables in JavaScript are arrays and strings, but other data structures like Maps and Sets are also iterable.

Example:

```
let array = [1, 2, 3];
let string = "Hello";

for (let element of array) {
  console.log(element);
}

for (let char of string) {
  console.log(char);
}
```

Explanation:

- The for...of loop is used to iterate over iterable objects like arrays and strings.

Common JavaScript Iterables

Here is a table summarizing some of the most commonly used iterables in JavaScript:

Iterable Type	Description	Explanation
Arrays	A collection of elements that can be iterated over.	Each element in an array can be accessed in sequence using iteration.
Strings	A sequence of characters that can be iterated one character at a time.	Each character in a string can be accessed in sequence using iteration.
Maps	A collection of key-value pairs where each key is unique.	Iteration over a Map gives you access to both keys and values.
Sets	A collection of unique values.	Iteration over a Set gives you access to each unique value.

Using the for...of Loop

The for...of loop is specifically designed for iterating over iterable objects. It provides a simple and clean syntax for accessing each element in an iterable.

Example:

```
let numbers = [10, 20, 30];


for (let number of numbers) {
  console.log(number);
}


// Output:
// 10
// 20
// 30
```

Explanation:

- The for...of loop iterates over each element in the numbers array and prints it to the console.

Custom Iterables

You can create your own iterable objects by defining the [Symbol.iterator] method in an object. This method should return an iterator that follows a specific protocol.

Example:

```
let customIterable = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    return {
      current: this.from,
      last: this.to,

      next() {
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      },
    };
  },
};
```

```
for (let value of customIterable) {
  console.log(value);
}

// Output:
// 1
// 2
// 3
// 4
// 5
```

Explanation:

- The custom object customIterable has a [Symbol.iterator] method, making it iterable.
- The next() method returns an object with done (a boolean indicating if the iteration is complete) and value (the current value in the iteration).

Built-in Iterators

JavaScript provides built-in iterators for its iterables. These iterators can be accessed using methods like .entries(), .keys(), and .values() for objects like arrays, Maps, and Sets.

Example:

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

let iterator = fruits.entries();

for (let entry of iterator) {
  console.log(entry);
}

// Output:
// [0, "Apple"]
// [1, "Banana"]
// [2, "Cherry"]
```

Explanation:

- The .entries() method returns an iterator that contains key-value pairs for each index and element in the array.

Examples with Code

Let's explore some practical examples of using iterables in JavaScript:

1. Iterating Over a String

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

for (let char of message) {
  console.log(char);
}

// Output:
// H
// e
// l
// l
// o
// ,
// 
// W
// o
// r
// l
// d
// !
```

Explanation: The for...of loop iterates over each character in the message string.

2. Iterating Over a Set

```
let uniqueNumbers = new Set([1, 2, 3, 3, 4]);

for (let num of uniqueNumbers) {
  console.log(num);
}

// Output:
// 1
// 2
// 3
// 4
```

Explanation: The for...of loop iterates over the unique values in the Set.

3. Iterating Over a Map

```
let userMap = new Map();
userMap.set("name", "John");
userMap.set("age", 30);

for (let [key, value] of userMap) {
  console.log(`${key}: ${value}`);
}

// Output:
// name: John
// age: 30
```

Explanation: The for...of loop iterates over the key-value pairs in the Map.

JavaScript Sets

Table of Contents

What is a Set?

A Set in JavaScript is a collection of unique values. Unlike arrays, a Set doesn't allow duplicate values, making it useful for storing collections where each item must be unique. The values in a Set can be of any data type, such as numbers, strings, or objects.

Sets are also ordered based on the insertion order, meaning elements are iterated over in the order they were added. This characteristic makes it easy to loop through the values in the sequence of their addition.

Example:

```
let uniqueNumbers = new Set([1, 2, 3, 4, 4, 5]);


console.log(uniqueNumbers); // Output: Set(5) { 1, 2, 3, 4, 5 }
```

Explanation:

- In the example, even though 4 is repeated in the array, the Set only keeps one instance of it, ensuring all values are unique.

Creating a Set

You can create a Set using the Set constructor, which can take an iterable object (like an array) as an argument.

Syntax:

```
let mySet = new Set([iterable]);
```

Example:

```
let fruits = new Set(["Apple", "Banana", "Cherry"]);
console.log(fruits); // Output: Set(3) { 'Apple', 'Banana', 'Cherry' }
```

Explanation:

- The Set is created with three unique elements: "Apple," "Banana," and "Cherry."

Adding and Removing Elements

You can add elements to a Set using the add() method and remove elements using the delete() method.

Example:

```
let mySet = new Set();


mySet.add(1);

mySet.add(2);

mySet.add(3);


console.log(mySet); // Output: Set(3) { 1, 2, 3 }


mySet.delete(2);


console.log(mySet); // Output: Set(2) { 1, 3 }
```

Explanation:

- The add() method adds elements to the Set, and the delete() method removes a specified element.

Working with Set Methods

Here are some common methods used with Sets:

1. **add(value)** - Adds a value to the Set.
2. **delete(value)** - Removes a value from the Set.
3. **has(value)** - Returns true if the value is in the Set, otherwise false.
4. **clear()** - Removes all elements from the Set.
5. **size** - Returns the number of elements in the Set.

Example:

```
let mySet = new Set(["Apple", "Banana"]);

mySet.add("Cherry");
console.log(mySet.has("Banana")); // Output: true

mySet.delete("Banana");
console.log(mySet.size); // Output: 2

mySet.clear();
console.log(mySet.size); // Output: 0
```

Explanation:

- has() checks if "Banana" is in the Set.
- delete() removes "Banana" from the Set.
- size gives the number of elements in the Set.
- clear() removes all elements from the Set.

Iterating Over a Set

You can iterate over the elements of a Set using the for...of loop or the forEach() method.

Example 1: Using for...of

```
let fruits = new Set(["Apple", "Banana", "Cherry"]);

for (let fruit of fruits) {
  console.log(fruit);
}

// Output:
// Apple
// Banana
// Cherry
```

Example 2: Using forEach()

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

// Output:
// Apple
// Banana
// Cherry
```

Explanation:

- Both for...of and forEach() methods allow you to loop through each element in the Set.

Examples with Code

Let's explore some practical examples of using Sets in JavaScript:

1. Removing Duplicates from an Array

```
let numbers = [1, 2, 2, 3, 4, 4, 5];
let uniqueNumbers = new Set(numbers);

console.log(uniqueNumbers); // Output: Set(5) { 1, 2, 3, 4, 5 }
```

Explanation: The Set automatically removes duplicate values from the array.

2. Converting a Set Back to an Array

```
let uniqueNumbersArray = Array.from(uniqueNumbers);

console.log(uniqueNumbersArray); // Output: [1, 2, 3, 4, 5]
```

Explanation: Array.from() converts the Set back to an array.

Use Cases for Sets

Sets are particularly useful in scenarios where you need to ensure that a collection of values is unique. Here are some common use cases:

1. **Removing Duplicates**: Automatically remove duplicates from an array.
2. **Tracking Unique Values**: Keep track of unique values without worrying about duplicates.
3. **Set Operations**: Perform operations like union, intersection, and difference between sets of data.

Example: Set Operations

```
let setA = new Set([1, 2, 3]);
let setB = new Set([3, 4, 5]);

// Union
let union = new Set([...setA, ...setB]);
console.log(union); // Output: Set(5) { 1, 2, 3, 4, 5 }

// Intersection
let intersection = new Set([...setA].filter((x) => setB.has(x)));
console.log(intersection); // Output: Set(1) { 3 }

// Difference
let difference = new Set([...setA].filter((x) => !setB.has(x)));
console.log(difference); // Output: Set(2) { 1, 2 }
```

Explanation:

- **Union**: Combines all elements from setA and setB.
- **Intersection**: Returns the common elements between setA and setB.
- **Difference**: Returns the elements that are in setA but not in setB.

JavaScript Set Methods

Table of Contents

List of Set Methods

Method	**Description**	**Example**
add(value)	Adds a new element to the Set.	mySet.add(1)
delete(value)	Removes the specified element from the Set.	mySet.delete(2)
has(value)	Checks if the Set contains the specified value.	mySet.has(2)
clear()	Removes all elements from the Set.	mySet.clear()
size	Returns the number of elements in the Set.	mySet.size
entries()	Returns an iterator with [value, value] pairs for each element in the Set.	mySet.entries()
forEach(callback)	Executes a function for each element in the Set.	mySet.forEach(value => {...})
keys() and values()	Return an iterator with the values of the Set.	mySet.keys() or mySet.values()

Detailed Explanation with Examples

add(value)

The add(value) method adds a new element with the specified value to a Set. If the value already exists in the Set, it won't be added again, ensuring all values are unique.

Syntax:

```
mySet.add(value);
```

Example:

```
let mySet = new Set();
mySet.add(1);
mySet.add(2);
mySet.add(1); // This will not be added as 1 already exists

console.log(mySet); // Output: Set(2) { 1, 2 }
```

Explanation:

- The add() method adds unique values to the Set. Attempting to add a duplicate value has no effect.

---

delete(value)

The delete(value) method removes the specified value from a Set. If the value is not found, the Set remains unchanged.

Syntax:

```
mySet.delete(value);
```

Example:

```
let mySet = new Set([1, 2, 3]);
mySet.delete(2);

console.log(mySet); // Output: Set(2) { 1, 3 }
```

Explanation:

- The delete() method removes the value 2 from the Set.

---

has(value)

The has(value) method checks if a Set contains a specific value. It returns true if the value exists in the Set, otherwise false.

Syntax:

```
mySet.has(value);
```

Example:

```
let mySet = new Set([1, 2, 3]);

console.log(mySet.has(2)); // Output: true
console.log(mySet.has(4)); // Output: false
```

Explanation:

- The has() method checks for the existence of a value in the Set.

---

clear()

The clear() method removes all elements from a Set, leaving it empty.

Syntax:

```
mySet.clear();
```

Example:

```
let mySet = new Set([1, 2, 3]);
mySet.clear();

console.log(mySet); // Output: Set(0) {}
```

Explanation:

- The clear() method removes all values from the Set.

---

size

The size property returns the number of elements in a Set.

Syntax:

```
mySet.size;
```

Example:

```
let mySet = new Set([1, 2, 3]);

console.log(mySet.size); // Output: 3
```

Explanation:

- The size property provides the count of elements in the Set.

---

entries()

The entries() method returns a new Iterator object that contains an array of [value, value] for each element in the Set, in insertion order. This method is primarily used for compatibility with the Map object.

Syntax:

```
mySet.entries();
```

Example:

```
let mySet = new Set(["a", "b", "c"]);
let iterator = mySet.entries();

for (let entry of iterator) {
  console.log(entry);
}

// Output:
// ["a", "a"]
// ["b", "b"]
// ["c", "c"]
```

Explanation:

- The entries() method returns an iterator where each element is represented as an array of the form [value, value].

---

forEach(callback)

The forEach(callback) method executes a provided function once for each value in the Set, in insertion order.

Syntax:

```
mySet.forEach(callback);
```

Example:

```
let mySet = new Set([1, 2, 3]);

mySet.forEach((value) => {
  console.log(value * 2);
});

// Output:
// 2
// 4
// 6
```

Explanation:

- The forEach() method applies the provided function to each element in the Set.

---

keys() and values()

The keys() and values() methods both return a new Iterator object containing the values for each element in the Set. Since a Set's elements are unique, and there are no keys in the usual sense, keys() and values() return the same values.

Syntax:

```
mySet.keys();
mySet.values();
```

Example:

```
let mySet = new Set([1, 2, 3]);
let keyIterator = mySet.keys();
let valueIterator = mySet.values();

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

Explanation:

- Both keys() and values() return an iterator containing the values of the Set elements.

JavaScript Map and Map Methods

Table of Contents

What is a Map?

A Map in JavaScript is a collection of key-value pairs where both keys and values can be of any data type. Maps are similar to plain JavaScript objects ({}), but with some key differences:

- **Maps maintain the order of elements**: Unlike objects, Maps preserve the order of insertion, meaning that when you iterate over a Map, the elements are returned in the order they were added.
- **Keys can be of any type**: While object keys are typically strings, Map keys can be any value (including functions, objects, or any primitive).

Example:

let myMap = new Map();

myMap.set("name", "John");

```
myMap.set(42, "The answer to life");

console.log(myMap.get("name")); // Output: John
console.log(myMap.get(42)); // Output: The answer to life
```

Explanation:

- The set() method adds key-value pairs to the Map, and the get() method retrieves the value associated with a key.

List of Map Methods

Method	**Description**	**Example**
set(key, value)	Adds or updates a key-value pair in the Map.	myMap.set('name', 'John')
get(key)	Retrieves the value associated with the specified key.	myMap.get('name')
has(key)	Checks if the Map contains the specified key.	myMap.has('name')
delete(key)	Removes the specified key and its associated value.	myMap.delete('name')
clear()	Removes all key-value pairs from the Map.	myMap.clear()
size	Returns the number of key-value pairs in the Map.	myMap.size
keys()	Returns an iterator for all keys in the Map.	myMap.keys()
values()	Returns an iterator for all values in the Map.	myMap.values()

Method	Description	Example
entries()	Returns an iterator for all key-value pairs in the Map.	myMap.entries()
forEach(callback)	Executes a function for each key-value pair in the Map.	myMap.forEach((value, key) => {...})

Detailed Explanation with Examples

set(key, value)

The set(key, value) method adds a new key-value pair to the Map or updates the value if the key already exists.

Syntax:

```
myMap.set(key, value);
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

console.log(myMap); // Output: Map(2) { 'name' => 'John', 'age' => 30 }
```

Explanation:

- The set() method adds the key 'name' with the value 'John' and the key 'age' with the value 30 to the Map.

---

get(key)

The get(key) method returns the value associated with the specified key. If the key is not found, it returns undefined.

Syntax:

```
myMap.get(key);
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
```

console.log(myMap.get("name")); // Output: John

console.log(myMap.get("age")); // Output: undefined

Explanation:

- The get() method retrieves the value associated with the key 'name', which is 'John'.

---

has(key)

The has(key) method checks whether the Map contains a specified key. It returns true if the key is found, otherwise false.

Syntax:

myMap.has(key);

Example:

let myMap = new Map();

myMap.set("name", "John");

console.log(myMap.has("name")); // Output: true

console.log(myMap.has("age")); // Output: false

Explanation:

- The has() method checks if the key 'name' exists in the Map and returns true.

---

delete(key)

The delete(key) method removes the specified key and its associated value from the Map.

Syntax:

myMap.delete(key);

Example:

let myMap = new Map();

myMap.set("name", "John");

myMap.delete("name");

```
console.log(myMap.has("name")); // Output: false
```

Explanation:

- The delete() method removes the key 'name' and its value from the Map.

---

clear()

The clear() method removes all key-value pairs from the Map, leaving it empty.

Syntax:

```
myMap.clear();
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);
myMap.clear();

console.log(myMap.size); // Output: 0
```

Explanation:

- The clear() method removes all key-value pairs from the Map.

---

size

The size property returns the number of key-value pairs currently in the Map.

Syntax:

```
myMap.size;
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

console.log(myMap.size); // Output: 2
```

Explanation:

- The size property gives the total count of key-value pairs in the Map.

---

keys()

The keys() method returns a new iterator object that contains the keys for each element in the Map, in insertion order.

Syntax:

```
myMap.keys();
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

for (let key of myMap.keys()) {
  console.log(key);
}

// Output:
// name
// age
```

Explanation:

- The keys() method returns an iterator for the keys in the Map.

---

values()

The values() method returns a new iterator object that contains the values for each element in the Map, in insertion order.

Syntax:

```
myMap.values();
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

for (let value of myMap.values()) {
  console.log(value);
}

// Output:
// John
// 30
```

Explanation:

- The values() method returns an iterator for the values in the Map.

---

entries()

The entries() method returns a new iterator object that contains an array of [key, value] for each element in the Map, in insertion order.

Syntax:

```
myMap.entries();
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

for (let entry of myMap.entries()) {
  console.log(entry);
}

// Output:
// ["name", "John"]
// ["age", 30]
```

Explanation:

- The entries() method returns an iterator with [key, value] pairs for each element in the Map.

---

forEach(callback)

The forEach(callback) method executes a provided function once for each key-value pair in the Map, in insertion order.

Syntax:

```
myMap.forEach((value, key) => {
  // code to execute
});
```

Example:

```
let myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

myMap.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});

// Output:
// name: John
// age: 30
```

Explanation:

- The forEach() method iterates over each key-value pair in the Map and executes the provided function.

How To find the frequency of elements in an array using a JavaScript Map

Example:

Let's say you have an array:

```
const array = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4];
```

Steps:

1. **Create a Map:** Initialize an empty Map to store the frequency of each element.
2. **Iterate over the Array:** Loop through each element in the array.
3. **Update the Map:** For each element, check if it already exists in the Map. If it does, increment its value by 1. If it doesn’t, set its value to 1.
4. **Result:** The Map will have the element as the key and its frequency as the value.

Implementation:

```
const array = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4];

// Create an empty Map
const frequencyMap = new Map();

// Iterate over the array
array.forEach((element) => {
  if (frequencyMap.has(element)) {
    // If the element exists in the map, increment its value
    frequencyMap.set(element, frequencyMap.get(element) + 1);
  } else {
    // If the element doesn't exist in the map, add it with a value of 1
    frequencyMap.set(element, 1);
  }
});

// To display the frequency of elements
frequencyMap.forEach((value, key) => {
  console.log(`Element: ${key}, Frequency: ${value}`);
});
```

Explanation:

- frequencyMap.set(element, frequencyMap.get(element) + 1) increments the count for the element if it already exists in the Map.

- frequencyMap.set(element, 1) adds the element to the Map with an initial count of 1 if it doesn't exist yet.

Output:

For the array [1, 2, 2, 3, 3, 3, 4, 4, 4, 4], the output will be:

Element: 1, Frequency: 1

Element: 2, Frequency: 2

Element: 3, Frequency: 3

Element: 4, Frequency: 4

This approach effectively counts the frequency of each element using a Map in JavaScript.

↥ Go to Top

Chapter-10: JavaScript Type Conversion, Destructuring, Bitwise Operations and Regular Expressions

JavaScript Type Conversion

Table of Contents

What is Type Conversion?

Type conversion in JavaScript refers to the process of converting a value from one data type to another. This can happen automatically (known as type coercion) or explicitly through functions and methods. Common conversions include converting between strings, numbers, and booleans.

Example:

let num = 10;

let str = String(num); // Converts number to string "10"

let bool = Boolean(num); // Converts number to boolean true

Explanation:

- The String() function converts a number to a string, and the Boolean() function converts it to a boolean.

Type Conversion Methods Overview

Method	**Description**	**Example**
String(value)	Converts a value to a string.	String(123) ➜ "123"
Number(value)	Converts a value to a number.	Number("123") ➜ 123
Boolean(value)	Converts a value to a boolean.	Boolean(1) ➜ true
parseInt(value)	Parses a string and returns an integer.	parseInt("123") ➜ 123
parseFloat(value)	Parses a string and returns a floating-point number.	parseFloat("12.34") ➜ 12.34

Method	Description	Example
value.toString()	Converts a value to a string using the toString() method.	(123).toString() ➜ "123"
value.toFixed(n)	Formats a number using fixed-point notation.	(12.3456).toFixed(2) ➜ "12.35"
value.toPrecision(n)	Formats a number to a specified precision.	(12.3456).toPrecision(3) ➜ "12.3"

Detailed Explanation with Examples

String()

The String() function converts a given value to a string.

Syntax:

```
String(value);
```

Example:

```
let num = 123;
let str = String(num);

console.log(str); // Output: "123"
console.log(typeof str); // Output: "string"
```

Explanation:

- The String() function converts the number 123 to the string "123".

Number()

The Number() function converts a given value to a number. If the value cannot be converted, it returns NaN (Not-a-Number).

Syntax:

```
Number(value);
```

Example:

```
let str = "123";
let num = Number(str);

console.log(num); // Output: 123
console.log(typeof num); // Output: "number"
```

Explanation:

- The Number() function converts the string "123" to the number 123.

Boolean()

The Boolean() function converts a given value to a boolean. Values like 0, null, undefined, NaN, "" (empty string) are converted to false, while all other values are converted to true.

Syntax:

```
Boolean(value);
```

Example:

```
let num = 0;
let bool = Boolean(num);

console.log(bool); // Output: false
console.log(typeof bool); // Output: "boolean"
```

Explanation:

- The Boolean() function converts the number 0 to false.

---

parseInt()

The parseInt() function parses a string and returns an integer of the specified radix (base). The function ignores leading whitespace and stops at the first character that cannot be converted to a number.

Syntax:

```
parseInt(string, radix);
```

Example:

```
let str = "123px";
let num = parseInt(str);

console.log(num); // Output: 123
console.log(typeof num); // Output: "number"
```

Explanation:

- The parseInt() function parses the string "123px" and returns the integer 123.

---

parseFloat()

The parseFloat() function parses a string and returns a floating-point number.
Like parseInt(), it stops parsing at the first character that is not part of the number.

Syntax:

```
parseFloat(string);
```

Example:

```
let str = "12.34px";
let num = parseFloat(str);

console.log(num); // Output: 12.34
console.log(typeof num); // Output: "number"
```

Explanation:

- The parseFloat() function parses the string "12.34px" and returns the floating-point number 12.34.

---

toString()

The toString() method converts a number, boolean, or other data types to a string. This method is commonly used on numbers to convert them to strings.

Syntax:

```
value.toString();
```

Example:

```
let num = 123;
let str = num.toString();

console.log(str); // Output: "123"
console.log(typeof str); // Output: "string"
```

Explanation:

- The toString() method converts the number 123 to the string "123".

---

toFixed()

The toFixed() method formats a number using fixed-point notation. It returns the string representation of the number rounded to the specified number of decimal places.

Syntax:

```
value.toFixed(digits);
```

Example:

```
let num = 12.3456;
let str = num.toFixed(2);

console.log(str); // Output: "12.35"
console.log(typeof str); // Output: "string"
```

Explanation:

- The toFixed() method rounds the number 12.3456 to two decimal places, resulting in the string "12.35".

---

toPrecision()

The toPrecision() method formats a number to the specified precision (number of significant digits). It returns the string representation of the number.

Syntax:

```
value.toPrecision(precision);
```

Example:

```
let num = 12.3456;
let str = num.toPrecision(3);
```

console.log(str); // Output: "12.3"

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

Explanation:

- The toPrecision() method formats the number 12.3456 to three significant digits, resulting in the string "12.3".

JavaScript Destructuring

Table of Contents

---

1. 📘 What is JavaScript Destructuring?

Destructuring হলো JavaScript এর একটি syntax যা arrays বা objects থেকে data সহজেই extract করে একাধিক variable এ assign করতে সাহায্য করে।

Syntax:

const [a, b] = [1, 2]; // Array Destructuring

const { x, y } = { x: 10, y: 20 }; // Object Destructuring

---

2. 🛠️ How to Use Destructuring?

Destructuring ব্যবহার করতে হলে:

1. প্রথমে একটি array বা object থাকা প্রয়োজন।
2. Variable গুলো তৈরি করুন, এবং array বা object থেকে data assign করুন।

---

3. 🔄 Array Destructuring with Explanation

Basic Example:

```
const colors = ["red", "green", "blue"];
const [first, second, third] = colors;

console.log(first); // red
console.log(second); // green
console.log(third); // blue
```

Explanation:

1. **Array Creation**:
   - colors নামে একটি array তৈরি করা হয়েছে, যার মধ্যে তিনটি color আছে।
2. **Destructuring Syntax**:
   - [first, second, third] array এর elements কে variables এ assign করেছে।
   - first = "red", second = "green", এবং third = "blue"।

Skipping Values:

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

console.log(second); // 20
console.log(fourth); // 40
```

Explanation:

1. **Skipping**:

   - , ব্যবহার করে প্রথম এবং তৃতীয় element skip করা হয়েছে।
2. **Variable Assignment**:
   - second = 20 এবং fourth = 40।

Swapping Variables:

```
let a = 1, b = 2;
[a, b] = [b, a];

console.log(a); // 2
console.log(b); // 1
```

Explanation:

1. **Swap Logic**:
   - [b, a] নতুন array তৈরি করেছে যা a এবং b এর মান swap করেছে।
2. **Destructuring**:
   - a = b এর মান এবং b = a এর মান পেয়েছে।

---

4. 📝 Object Destructuring with Explanation

Basic Example: {#object-basic-example}

```
const user = { name: "John", age: 30, country: "USA" };
const { name, age, country } = user;

console.log(name); // John
console.log(age);  // 30
console.log(country); // USA
```

Explanation:

1. **Object Creation**:
   - user নামে একটি object তৈরি করা হয়েছে।
2. **Destructuring**:

- { name, age, country } object এর properties কে variables এ assign করেছে।
- name = "John", age = 30, এবং country = "USA"।

Nested Destructuring: {#nested-destructuring}

```
const user = { name: "Alice", address: { city: "New York", zip: 10001 } };
const { address: { city, zip } } = user;

console.log(city); // New York
console.log(zip);  // 10001
```

Explanation:

1. **Nested Object**:
   - user এর মধ্যে address নামে nested object আছে।
2. **Destructuring**:
   - address এর ভেতরের city এবং zip properties destructure করা হয়েছে।

Renaming Variables: {#renaming-variables}

```
const product = { id: 101, name: "Laptop" };
const { id: productId, name: productName } = product;

console.log(productId);   // 101
console.log(productName); // Laptop
```

Explanation:

1. **Renaming**:
   - id property কে productId এবং name কে productName নামে assign করা হয়েছে।
2. **Output**:
   - New variable names অনুযায়ী data access করা হয়েছে।

---

5. 🏗️ Destructuring in Function Parameters

Example:

```
function greet({ name, age }) {
  console.log(`Hello, ${name}! You are ${age} years old.`);
}

const user = { name: "John", age: 30 };
greet(user); // Hello, John! You are 30 years old.
```

Explanation:

1. **Function Parameter Destructuring**:
   - Function এর parameter এর মধ্যেই destructuring করা হয়েছে।
2. **Cleaner Code**:
   - সরাসরি object properties access করা হয়েছে।

---

6. 🛠️ Default Values in Destructuring

Example:

```
const user = { name: "Alice" };
const { name, age = 25 } = user;

console.log(name); // Alice
console.log(age);  // 25
```

Explanation:

1. **Default Values**:
   - age property না থাকলে default value 25 assign হবে।
2. **Fallback Mechanism**:
   - Missing properties এর জন্য fallback value নির্ধারণ করা হয়েছে।

---

7. 📖 Real-Life Examples with Detailed Explanation

Fetch API Response:

```
const response = {
  data: { user: { name: "Alice", age: 25 } },
  status: 200,
};

const {
  data: { user: { name, age } },
  status,
} = response;

console.log(name); // Alice
console.log(age);  // 25
console.log(status); // 200
```

Explanation:

1. **Nested Object**:
   - response object থেকে nested properties destructure করা হয়েছে।
2. **Rename and Extract**:
   - data.user.name এবং data.user.age সরাসরি extract হয়েছে।

React Props:

```
function Profile({ name, age }) {
  return <p>{name} is {age} years old.</p>;
}

// Usage
<Profile name="John" age={30} />;
```

Explanation:

1. **Props Destructuring**:
   - Profile function এর parameter এর মধ্যেই props destructure করা হয়েছে।

2. **Cleaner JSX**:
    - Props সরাসরি ব্যবহার করা হয়েছে।

---

8. 🛡️ Best Practices

1. **Use Default Values**:
2. const { age = 25 } = { name: "John" };

console.log(age); // 25

3. **Destructure Function Parameters**:
4. function greet({ name, age }) {
5.   console.log(`Hello, ${name}.`);

}

6. **Meaningful Variable Names**:
7. const { id: productId } = { id: 101 };

console.log(productId); // 101

Summary:

Destructuring হলো JavaScript এর একটি গুরুত্বপূর্ণ feature যা code readability এবং efficiency বৃদ্ধি করে। এটি arrays এবং objects থেকে data extract করা সহজ করে এবং modern JavaScript development এর জন্য অপরিহার্য।

JavaScript Bitwise Operations

JavaScript এর **Bitwise Operations** এমন অপারেশন যা **bit-level manipulation** এর মাধ্যমে কাজ করে। এটি integer numbers কে binary আকারে ব্যবহার করে কাজ করে। Bitwise operations high-performance computations এর জন্য ব্যবহৃত হয়।

Table of Contents

---

1. 📘 What Are Bitwise Operations?

Bitwise operations **binary representation** এ numbers এর উপর কাজ করে। অর্থাৎ, এটি numbers কে bits (0 এবং 1) হিসেবে গণনা করে।

Example:

5 & 3; // Binary: 0101 & 0011 = 0001 (Decimal: 1)

---

2. 🧐 Why Use Bitwise Operations?

1. **High Performance**:
   - Bitwise operations খুব দ্রুত কাজ করে কারণ এটি সরাসরি binary data নিয়ে কাজ করে।
2. **Memory Efficiency**:
   - Bitwise operations কম memory ব্যবহার করে।
3. **Advanced Algorithms**:
   - Cryptography, image processing, এবং gaming algorithms-এ ব্যবহৃত হয়।
4. **Custom Flags**:
   - Custom flags বা options পরিচালনা করতে ব্যবহার করা হয়।

---

3. 🔄 Types of Bitwise Operators

JavaScript এ সাত ধরণের bitwise operator আছে:

---

1. AND (&)

Functionality:

- দুটি bit উভয়ই 1 হলে result 1 হবে। অন্যথায়, 0।

Truth Table:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Example:

const a = 5; // Binary: 0101

const b = 3; // Binary: 0011

console.log(a & b); // Binary: 0001 => Decimal: 1

---

2. OR (|)

Functionality:

- যেকোনো একটি bit 1 হলে result 1 হবে।

Truth Table:

| A | B | A | B | |---|---|-------| | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 |

Example:

const a = 5; // Binary: 0101

const b = 3; // Binary: 0011

console.log(a | b); // Binary: 0111 => Decimal: 7

---

3. XOR (^)

Functionality:

- Bits ভিন্ন হলে result 1, একই হলে result 0।

Truth Table:

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Example:

const a = 5; // Binary: 0101

const b = 3; // Binary: 0011

console.log(a ^ b); // Binary: 0110 => Decimal: 6

---

4. NOT (~)

Functionality:

- Bit কে invert করে (0 থেকে 1 এবং 1 থেকে 0)।

Example:

const a = 5; // Binary: 0101

console.log(~a); // Binary: 1010 => Decimal: -6

Explanation:

- JavaScript এ NOT অপারেটর 32-bit signed integer return করে।

---

5. Left Shift (<<)

Functionality:

- Bits নির্দিষ্ট সংখ্যক স্থান বামে সরায়।

Example:

const a = 5; // Binary: 0101

console.log(a << 1); // Binary: 1010 => Decimal: 10

Explanation:

- << 1 মানে binary value বামে 1 স্থান সরানো এবং শেষে 0 যোগ করা।

---

6. Right Shift (>>)

Functionality:

- Bits নির্দিষ্ট সংখ্যক স্থান ডানে সরায়। Sign bit ধরে রাখে।

Example:

const a = -5; // Binary: 111...1011 (32 bits)

console.log(a >> 1); // Binary: 111...1101 => Decimal: -3

Explanation:

- Negative numbers এর ক্ষেত্রে sign bit (1) ধরে রাখা হয়।

---

7. Unsigned Right Shift (>>>)

Functionality:

- Bits নির্দিষ্ট সংখ্যক স্থান ডানে সরায়, কিন্তু sign bit বাদ দেয়।

Example:

const a = -5; // Binary: 111...1011

console.log(a >>> 1); // Binary: 011...1101 => Decimal: 2147483645

---

4. 📖 Examples and Explanation

Example 1: Check if a Number is Even or Odd

```
const isEven = (num) => (num & 1) === 0;


console.log(isEven(4)); // true (Even)

console.log(isEven(5)); // false (Odd)
```

Explanation:

- num & 1: Binary AND অপারেশন। Odd numbers এর শেষ bit 1 হয়, আর Even numbers এর শেষ bit 0 হয়।

Example 2: Toggle a Bit

```
const toggleBit = (num, bitPosition) => num ^ (1 << bitPosition);

console.log(toggleBit(5, 1)); // Binary: 0101 => 0111 => Decimal: 7
console.log(toggleBit(7, 1)); // Binary: 0111 => 0101 => Decimal: 5
```

Explanation:

- (1 << bitPosition) মানে নির্দিষ্ট স্থানে 1 সেট করা।
- ^ দিয়ে toggle করা হয়।

Example 3: Set a Bit

```
const setBit = (num, bitPosition) => num | (1 << bitPosition);

console.log(setBit(5, 1)); // Binary: 0101 => 0111 => Decimal: 7
```

Explanation:

- (1 << bitPosition) নির্দিষ্ট স্থানে 1 সেট করে এবং | দিয়ে original number এর সাথে combine করে।

Example 4: Clear a Bit

```
const clearBit = (num, bitPosition) => num & ~(1 << bitPosition);

console.log(clearBit(7, 1)); // Binary: 0111 => 0101 => Decimal: 5
```

Explanation:

- (1 << bitPosition) নির্দিষ্ট স্থানে 1 সেট করে।
- ~ দিয়ে invert করে, এবং & দিয়ে bit clear করে।

---

5. ✅ Use Cases of Bitwise Operations

1. **Performance Optimization**:
   - Cryptography এবং image processing algorithms-এ ব্যবহৃত হয়।
2. **Flags Management**:
   - Multiple boolean states পরিচালনা করতে।
3. **Gaming**:
   - Binary masks ব্যবহার করে game states পরিচালনা করতে।
4. **Low-Level Programming**:
   - Memory এবং hardware-level কাজ করার জন্য।

---

6. ⚠ Common Mistakes and Best Practices

Mistakes:

1. **Negative Number Handling**:
   - Negative numbers এর binary representation signed format এ থাকে।
2. **Overusing Bitwise Operations**:
   - Code readability কমে যেতে পারে।

Best Practices:

1. **Use Comments**:
   - Bitwise logic বোঝানোর জন্য comments ব্যবহার করুন।
2. **Test Edge Cases**:
   - Negative numbers এবং large values test করুন।

Summary

JavaScript এর **Bitwise Operations** একটি শক্তিশালী tool যা high-performance computations এর জন্য উপযুক্ত। এটি low-level data manipulation এর জন্য ব্যবহৃত হয় এবং advanced algorithms তৈরিতে গুরুত্বপূর্ণ ভূমিকা পালন করে।

JavaScript Regular Expressions

JavaScript এর **Regular Expressions (RegEx)** হলো text বা string এর মধ্যে pattern matching এবং manipulation এর জন্য একটি শক্তিশালী টুল। এটি বিভিন্ন text matching tasks যেমন validation, searching, এবং replacing এর জন্য ব্যবহার করা হয়।

Table of Contents

---

1. 📘 What is a Regular Expression?

A **Regular Expression (RegEx)** হলো একটি pattern যা string এর মধ্যে searching, matching এবং replacing এর কাজ করে।

Basic Examples:

- Match "abc" in a string: /abc/
- Match a digit: /\d/
- Match an email address: /^\w+@\w+\.\w+$/

---

2. 🧐 Why Use Regular Expressions?

1. **Efficient Searching**:
   - Text এর মধ্যে complex patterns match করতে পারে।
2. **Validation**:
   - Input validation এর জন্য ব্যবহৃত হয় (e.g., email, phone number, password)।
3. **Text Manipulation**:

- String থেকে unwanted characters সরাতে বা replace করতে।

---

3. 🔄 Syntax of Regular Expressions

A regular expression একটি pattern এবং optional **flags** দিয়ে গঠিত হয়।

General Syntax:

const regex = /pattern/flags;

Example:

const regex = /hello/i;

- **Pattern**: /hello/ -> "hello" শব্দটি match করবে।
- **Flags**: i -> Case insensitive।

---

4. 🛠 Creating a Regular Expression

JavaScript এ RegEx তৈরি করার দুটি পদ্ধতি আছে:

1. Literal Notation:

const regex = /pattern/flags;

Example:

const regex = /abc/i;

2. RegExp Constructor:

const regex = new RegExp('pattern', 'flags');

Example:

const regex = new RegExp('abc', 'i');

---

5. 📖 Common Metacharacters

Metacharacters RegEx এর প্রধান অংশ। এগুলো বিশেষ অর্থ বহন করে।

Metacharacter	Description	Example
.	Any single character except newline	/a.b/ -> "acb"

Metacharacter	Description	Example
\d	Any digit (0-9)	/\d/ -> "5"
\D	Non-digit character	/\D/ -> "a"
\w	Any word character (alphanumeric + _)	/\w/ -> "a"
\W	Non-word character	/\W/ -> "@"
\s	Whitespace (space, tab, newline)	/\s/ -> " "
\S	Non-whitespace character	/\S/ -> "a"
^	Start of string	/^a/ -> "abc"
$	End of string	/a$/ -> "cba"
*	Zero or more occurrences	/a*/ -> "aaa"
+	One or more occurrences	/a+/ -> "aaa"
?	Zero or one occurrence	/a?/ -> "a"
{n}	Exactly n occurrences	/a{2}/ -> "aa"
[]	Matches any character inside brackets	/[abc]/
`	`	OR operation

---

6. 🏷️ Flags in Regular Expressions

Flags RegEx এর behavior নিয়ন্ত্রণ করে।

Flag	Description	Example
i	Case-insensitive search	/abc/i
g	Global search	/abc/g

Flag	Description	Example
m	Multiline search	/^a/m
s	Dot matches newline as well	/a.b/s
u	Unicode support	/\u{1F600}/u
y	Sticky search	/abc/y

---

7. ⚙️ RegEx Methods in JavaScript

JavaScript এ Regular Expressions এর জন্য কিছু built-in methods আছে।

String Methods:

1. **match**:
   - Pattern match করার জন্য।
2. const str = "hello world";
3. const result = str.match(/hello/);

console.log(result); // ["hello"]

4. **matchAll**:
   - সব matches return করে।
5. const str = "cat, bat, rat";
6. const result = str.matchAll(/at/g);
7. for (const match of result) {
8.   console.log(match[0]);

}

9. **replace**:
   - Match করা string কে replace করার জন্য।
10. const str = "hello world";
11. const result = str.replace(/world/, "JavaScript");

console.log(result); // "hello JavaScript"

12. **split**:
    - String কে RegEx এর মাধ্যমে ভাগ করার জন্য।
13. const str = "apple,banana,cherry";
14. const result = str.split(/,/);

console.log(result); // ["apple", "banana", "cherry"]

RegExp Methods:

1. **test**:
    - Pattern match হলে true return করে।
2. const regex = /hello/;

console.log(regex.test("hello world")); // true

3. **exec**:
    - Match করা data return করে।
4. const regex = /world/;

console.log(regex.exec("hello world")); // ["world"]

---

8. 📖 Examples and Use Cases

Example 1: Validate an Email

const email = "test@example.com";

const regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

console.log(regex.test(email)); // true

Example 2: Extract Numbers from a String

const str = "My phone number is 12345";

const regex = /\d+/g;

console.log(str.match(regex)); // ["12345"]

Example 3: Replace Multiple Spaces with a Single Space

```
const str = "This   is   a   test";
const regex = /\s+/g;
console.log(str.replace(regex, " ")); // "This is a test"
```

Example 4: Validate a Password

```
const password = "P@ssw0rd123";
const regex = /^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/;
console.log(regex.test(password)); // true
```

Explanation:

- At least one uppercase letter.
- At least one lowercase letter.
- At least one digit.
- At least one special character.
- Minimum 8 characters.

Example 5: Find All Words Starting with a Specific Letter

```
const str = "Apple Banana Apricot Berry";
const regex = /\bA\w*/g;
console.log(str.match(regex)); // ["Apple", "Apricot"]
```

---

9. ✅ Best Practices

1. **Use Escaping (\)**:
   - Special characters যেমন . বা ? এর মতো literal match করতে হলে escape করতে হবে।

```
const regex = /\./;
```

2. **Use Descriptive Patterns**:
   - Complex RegEx সহজে পড়ার জন্য comments বা whitespace ব্যবহার করুন।
3. **Avoid Overuse**:
   - Simple string methods (like includes) ব্যবহার করুন যেখানে সম্ভব।

4. **Test Your Patterns**:
    - RegEx patterns validate করার জন্য tools যেমন regex101 ব্যবহার করুন।

Summary

JavaScript এর **Regular Expressions** একটি শক্তিশালী টুল যা text এর মধ্যে pattern match এবং manipulate করতে ব্যবহৃত হয়। এটি validation, searching, এবং string manipulation এর জন্য অপরিহার্য।

আপনার যদি কোনো অংশ বিস্তারিতভাবে জানতে চান বা প্রশ্ন থাকে, আমাকে জানাবেন! 😊

↥ Go to Top

Chapter-11: JavaScript Errors, Use Strict, This Keyword and Arrow Function

JavaScript Errors

JavaScript এ **Errors** হলো programming mistakes বা runtime exceptions যা কোডের execution কে ব্যাহত করে। Errors শনাক্ত করে debugging এবং handling করার জন্য JavaScript এর built-in error objects এবং handling mechanisms রয়েছে।

Table of Contents

---

1. 📘 What Are JavaScript Errors?

JavaScript Errors এমন অবস্থা যা কোডের execution ব্যাহত করে। এগুলো হতে পারে:

- **Syntax Errors**: Invalid JavaScript syntax।
- **Runtime Errors**: কোড চলাকালীন সময়ে সমস্যা।
- **Logical Errors**: ভুল লজিকের কারণে ভুল ফলাফল।

---

2. 🧐 Why Do Errors Occur?

Errors এর কিছু সাধারণ কারণ:

1. **Syntax Mistakes**:
    - ভুলভাবে কোড লেখা।

console.log("Hello World" // Missing closing parenthesis

2. **Undefined Variables**:
    - Variable declare না করেই access করার চেষ্টা।

console.log(a); // ReferenceError: a is not defined

3. **Invalid Function Usage**:
    - Function কে ভুল data দিয়ে call করা।
4. const num = 10;

num.toUpperCase(); // TypeError: num.toUpperCase is not a function

---

3. 🔄 Types of Errors in JavaScript

---

1. SyntaxError

- যখন কোডে ভুল syntax থাকে।

Example:

```
console.log("Hello World"; // SyntaxError: Unexpected token ';'
```

---

2. ReferenceError

- যখন একটি variable declare করা হয়নি, অথচ access করার চেষ্টা করা হয়।

Example:

```
console.log(a); // ReferenceError: a is not defined
```

---

3. TypeError

- যখন একটি variable বা property তার expected type এর সাথে match করে না।

Example:

```
const num = 10;
num.toUpperCase(); // TypeError: num.toUpperCase is not a function
```

---

4. RangeError

- যখন একটি value তার valid range এর বাইরে যায়।

Example:

```
function factorial(num) {
  if (num > 1000) throw new RangeError("Number is too large!");
}
factorial(1001); // RangeError: Number is too large!
```

---

5. URIError

- Invalid URI (Uniform Resource Identifier) এর কারণে।

Example:

```
decodeURIComponent('%'); // URIError: URI malformed
```

---

6. EvalError

- eval function ব্যবহার সংক্রান্ত error (ব্যবহার খুব বিরল)।

Example:

```
throw new EvalError("Eval error occurred!");
```

---

7. AggregateError

- যখন একটি operation multiple errors return করে।

Example:

```
Promise.any([
  Promise.reject(new Error("Error 1")),
  Promise.reject(new Error("Error 2")),
]).catch((err) => console.log(err)); // AggregateError: All Promises were rejected
```

---

4. 🛠️ Error Handling in JavaScript

Errors handle করার জন্য JavaScript এর built-in mechanisms রয়েছে।

Using try-catch

Syntax:

```
try {
  // Code that might throw an error
} catch (error) {
  // Handle the error
}
```

Example:

```
try {
  const num = undefined;
  console.log(num.toString());
} catch (error) {
  console.error("An error occurred:", error.message);
}
```

Output:

```
An error occurred: Cannot read properties of undefined (reading 'toString')
```

Throwing Custom Errors

Syntax:

```
throw new Error("Custom error message");
```

Example:

```
function divide(a, b) {
  if (b === 0) throw new Error("Cannot divide by zero!");
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (error) {
  console.error(error.message);
}
```

Output:

```
Cannot divide by zero!
```

Finally Block

finally block সব সময় execute হয়, error হোক বা না হোক।

Example:

```
try {
  console.log("Trying...");
  throw new Error("An error occurred!");
} catch (error) {
  console.log(error.message);
} finally {
  console.log("This will always execute.");
}
```

Output:

```
Trying...
An error occurred!
This will always execute.
```

---

5. 📖 Examples and Explanation

Example 1: Nested try-catch

```
try {
  try {
    throw new Error("Inner error!");
  } catch (innerError) {
    console.error("Caught inner error:", innerError.message);
    throw new Error("Outer error!");
  }
} catch (outerError) {
  console.error("Caught outer error:", outerError.message);
}
```

Output:

```
Caught inner error: Inner error!
Caught outer error: Outer error!
```

Example 2: Validating User Input

```
function validateAge(age) {
  if (typeof age !== "number") throw new TypeError("Age must be a number");
  if (age < 0) throw new RangeError("Age cannot be negative");
  return "Valid age";
}

try {
  console.log(validateAge(-5));
} catch (error) {
  console.error(error.name + ": " + error.message);
}
```

Output:

```
RangeError: Age cannot be negative
```

---

6. ✅ Best Practices for Error Handling

1. **Use Specific Errors**:
   - Use TypeError, RangeError ইত্যাদি specific error types।
2. **Don’t Suppress Errors**:
   - Errors কে silent করে না রেখে meaningful ভাবে handle করুন।
3. **Log Errors**:
   - Errors log করুন debugging এর জন্য।

```
console.error("Error:", error.message);
```

4. **Avoid Overusing try-catch**:
   - শুধু critical এবং unpredictable code এ ব্যবহার করুন।
5. **Graceful Fallbacks**:
   - Errors থাকলে fallback values বা alternative logic ব্যবহার করুন।

Summary

JavaScript Errors handle করার জন্য একটি শক্তিশালী এবং structured approach দেয়। **try-catch-finally**, **custom errors**, এবং built-in error types ব্যবহার করে runtime issues সমাধান করা সহজ। Proper error handling application এর stability এবং usability বাড়াতে গুরুত্বপূর্ণ ভূমিকা রাখে।

JavaScript Use Strict

"use strict" হলো JavaScript এর একটি **directive** যা কোড execution কে **strict mode** এ নিয়ে যায়। এটি কিছু **common coding mistakes** এড়াতে সাহায্য করে এবং JavaScript কোডকে আরও নিরাপদ ও নির্ভরযোগ্য করে তোলে।

Table of Contents

---

1. 📘 What is "use strict"?

"use strict" একটি JavaScript directive যা strict mode enable করে।

- **Strict Mode**: এটি একটি **restricted variant** যা JavaScript এর silent errors (যেগুলো naturally ignore করা হয়) detect করে এবং errors হিসেবে throw করে।
- এটি ES5 (ECMAScript 5) থেকে চালু হয়েছে।

---

2. 🧐 Why Use "use strict"?

1. **Prevent Common Mistakes**:
   - ভুলভাবে variables declare না করা।
2. **Improves Performance**:

    - Strict mode code দ্রুত execute হয় কারণ এটি optimized হয়।
3. **Makes Debugging Easier**:
    - Silent errors detect এবং throw করে।
4. **Avoid Deprecated Features**:
    - JavaScript এর কিছু old বা unsafe features strict mode এ নিষিদ্ধ।

---

3. 🛠️ How to Enable Strict Mode?

Strict Mode enable করার জন্য "use strict" একটি string হিসেবে function বা script এর শুরুতে রাখতে হয়।

---

1. For the Entire Script

```
"use strict";

x = 10; // ReferenceError: x is not defined
```

---

2. For Specific Functions

```
function strictFunction() {
  "use strict";
  y = 20; // ReferenceError: y is not defined
}

function nonStrictFunction() {
  z = 30; // No error (not in strict mode)
}

strictFunction();
nonStrictFunction();
```

---

3. In ES6 Modules

- ES6 Modules (e.g., import/export) automatically strict mode এ থাকে।

```
export function myFunction() {
  x = 40; // ReferenceError: x is not defined
}
```

---

4. 🔄 Changes in Strict Mode

1. Prevents Undeclared Variables

```
"use strict";
x = 10; // ReferenceError: x is not defined
```

---

2. Disallows Duplicates

```
"use strict";
function duplicateArgs(a, a) {
  // SyntaxError: Duplicate parameter name not allowed in strict mode
}
```

---

3. Eliminates this Default to Global Object

```
"use strict";
function showThis() {
  console.log(this); // undefined
}
showThis();
```

- Non-strict mode এ this global object (window or global) reference করে।

---

4. Throws Errors for Read-Only Properties

```
"use strict";
const obj = {};
Object.defineProperty(obj, "readOnly", { value: 42, writable: false });
```

obj.readOnly = 99; // TypeError: Cannot assign to read only property

---

5. Silent Errors Become Explicit

"use strict";

delete Object.prototype; // TypeError: Cannot delete property

---

6. Reserved Keywords are Restricted

"use strict";

const public = 42; // SyntaxError: Unexpected strict mode reserved word

---

5. 📖 Examples of "use strict"

Example 1: Variable Declaration

"use strict";

x = 10; // ReferenceError: x is not defined

Without strict mode, এটি silently var x = 10 হিসেবে কাজ করবে। কিন্তু strict mode এ, এটি error throw করে।

Example 2: Duplicate Parameter Names

```
"use strict";
function sum(a, a) {
  // SyntaxError: Duplicate parameter name not allowed
}
```

Example 3: Prevent Deleting Properties

"use strict";

const obj = { name: "John" };

delete obj.name; // TypeError: Cannot delete property 'name'

Example 4: Reserved Keywords

```
"use strict";
const let = 10; // SyntaxError: Unexpected strict mode reserved word
```

Example 5: Default this Value

```
"use strict";
function showThis() {
  console.log(this); // undefined
}
showThis();
```

---

6. ✅ Benefits of Using "use strict"

1. **Error Detection**:
   - Silent errors detect এবং handle করা সহজ।
2. **Better Optimization**:
   - Modern JavaScript engines strict mode কে optimize করে।
3. **Improved Code Security**:
   - Global scope এর misuse রোধ করে।
4. **Readable Code**:
   - Code clear এবং predictable হয়।

---

7. ⚠️ Limitations of "use strict"

1. **Backward Compatibility**:
   - Strict mode পুরোনো JavaScript কোডে সমস্যা তৈরি করতে পারে।
2. **Not Suitable for All Scripts**:
   - ছোট বা simple script এ strict mode অতিরিক্ত মনে হতে পারে।

---

8. 🛡️ Best Practices

1. **Always Use Strict Mode**:
    - Bugs এড়াতে নতুন JavaScript project এ strict mode ব্যবহার করুন।
2. **Enable for Specific Functions**:
    - পুরোনো কোডে সমস্যা এড়াতে শুধুমাত্র নতুন functions এ strict mode enable করুন।
3. **Avoid Using Deprecated Features**:
    - JavaScript এর modern syntax এবং features ব্যবহার করুন।
4. **Test Legacy Code**:
    - পুরোনো কোড strict mode এর সাথে চলবে কিনা যাচাই করুন।

Summary

"use strict" JavaScript কোডের quality এবং debugging improve করার জন্য অপরিহার্য। এটি common mistakes এড়িয়ে কোড clean এবং optimized রাখতে সাহায্য করে। Modern JavaScript development এ strict mode সর্বদা enable করা উচিত।

JavaScript this Keyword

JavaScript এর **this keyword** এমন একটি গুরুত্বপূর্ণ feature, যা execution context এর উপর ভিত্তি করে বিভিন্ন মান ধারণ করে। এটি কোডে কোন object থেকে একটি function call হচ্ছে তা নির্দেশ করে। এই Documentation এ this keyword এর ব্যবহার, কাজ করার পদ্ধতি, বিভিন্ন context-এ এর আচরণ, এবং এটি সঠিকভাবে ব্যবহার করার পদ্ধতি বিস্তারিতভাবে ব্যাখ্যা করা হয়েছে।

Table of Contents

---

1. 📘 What is this in JavaScript?

this হলো একটি special keyword, যা function execution context এর উপর ভিত্তি করে বিভিন্ন মান ধারণ করে। সহজ কথায়, **this** সেই object কে নির্দেশ করে, যা থেকে functionটি call হয়েছে।

Example:

```
const person = {
  name: "John",
  greet: function () {
    console.log(this.name);
  },
};
person.greet(); // Output: John
```

ব্যাখ্যা:

- this.name এখানে person object এর name property কে নির্দেশ করছে কারণ function টি person.greet() এর মাধ্যমে call হয়েছে।

---

2. 🧐 Why is this Important?

JavaScript এর **dynamic context** এবং object-oriented features এর জন্য this খুবই গুরুত্বপূর্ণ। এটি বিভিন্নভাবে কোড উন্নত এবং modular করতে সাহায্য করে।

Key Reasons to Use this:

1. **Dynamic Context Management**:
    - একই function বিভিন্ন object এর context অনুযায়ী ব্যবহার করা যায়।

2. const obj1 = { name: "Alice" };
3. const obj2 = { name: "Bob" };
4. function sayName() {
5.   console.log(this.name);
6. }
7. sayName.call(obj1); // Alice

sayName.call(obj2); // Bob

8. **Code Reusability**:
   - Functions বা methods reusable রাখা সহজ হয়।
9. **Event Handling**:
   - DOM elements এর event handlers এ this ব্যবহার হয়।
10. document.querySelector("button").addEventListener("click", function () {
11.   console.log(this); // The button element

});

---

3. 🔄 How this Works in Different Contexts

---

1. Global Context

Explanation:

Global context এ, this এর মান নির্ভর করে:

- Non-strict mode: this global object (window বা global in Node.js) নির্দেশ করে।
- Strict mode: this undefined হয়ে যায়।

উদাহরণ:

```
console.log(this); // Non-strict mode: window object
"use strict";
console.log(this); // Strict mode: undefined
```

---

2. Function Context

Explanation:

একটি function এর মধ্যে, this নির্ধারিত হয় **function কিভাবে call করা হয়েছে** তার উপর ভিত্তি করে।

উদাহরণ:

```
function showThis() {
  console.log(this);
}
showThis(); // Non-strict mode: window object
"use strict";
function showThis() {
  console.log(this);
}
showThis(); // Strict mode: undefined
```

---

3. Object Context

Explanation:

যখন একটি function একটি object এর method হিসেবে call হয়, তখন this keyword সেই object কে নির্দেশ করে।

উদাহরণ:

```
const obj = {
  name: "Alice",
  greet: function () {
    console.log(this.name);
  },
};
obj.greet(); // Output: Alice
```

---

4. Class Context

Explanation:

Class এর মধ্যে, this সেই object কে নির্দেশ করে যা class এর একটি instance তৈরি করে।

উদাহরণ:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}
const person = new Person("Alice");
person.greet(); // Output: Hi, I'm Alice
```

---

5. Arrow Functions

Explanation:

Arrow functions এর নিজস্ব this থাকে না। এটি lexical scope থেকে this inherit করে।

উদাহরণ:

```
const obj = {
  name: "Alice",
  greet: function () {
    const arrow = () => console.log(this.name);
    arrow();
  },
};
obj.greet(); // Output: Alice
```

---

4. 🛠️ Explicitly Binding this

Using call

Explanation:

call method একটি function এর this এর মান নির্ধারণ করতে ব্যবহৃত হয় এবং সাথে সাথে function invoke করে।

উদাহরণ:

```
function greet() {
  console.log(`Hello, ${this.name}`);
}
const obj = { name: "Alice" };
greet.call(obj); // Output: Hello, Alice
```

Using apply

Explanation:

apply method call এর মতোই কাজ করে, কিন্তু arguments একটি array আকারে পাঠানো হয়।

উদাহরণ:

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}
const obj = { name: "Alice" };
greet.apply(obj, ["Hi"]); // Output: Hi, Alice
```

Using bind

Explanation:

bind একটি নতুন function তৈরি করে যেখানে this স্থায়ীভাবে সেট করা থাকে।

উদাহরণ:

```
function greet() {
  console.log(`Hello, ${this.name}`);
}
const obj = { name: "Alice" };
const boundGreet = greet.bind(obj);
boundGreet(); // Output: Hello, Alice
```

---

5. 📖 Common Mistakes and Pitfalls

1. **Losing this in Callbacks**:
2. const obj = {
3.   name: "Alice",
4.   greet: function () {
5.   console.log(this.name);
6.   },
7. };

setTimeout(obj.greet, 1000); // Output: undefined

Solution:

setTimeout(obj.greet.bind(obj), 1000); // Output: Alice

---

2. **Arrow Functions as Methods**:
3. const obj = {
4.   name: "Alice",
5.   greet: () => {
6.   console.log(this.name);
7.   },
8. };

obj.greet(); // Output: undefined

Solution:

```
const obj = {
  name: "Alice",
  greet: function () {
    console.log(this.name);
  },
};
obj.greet(); // Output: Alice
```

---

6. ✅ Best Practices

1. **Use Arrow Functions for Lexical this**: Arrow functions ব্যবহার করুন যেখানে parent scope থেকে this inherit করতে হবে।
2. **Explicit Binding**: call, apply, এবং bind ব্যবহার করে this সঠিকভাবে handle করুন।
3. **Avoid Overusing this**: যেখানে প্রয়োজন সেখানে this ব্যবহার করুন। Overuse করলে debugging কঠিন হয়ে যায়।
4. **Prefer Classes for Object-Oriented Code**: Object-oriented code এর জন্য ES6 classes ব্যবহার করুন। এতে this এর behavior সহজ হয়।

Summary

JavaScript এর **this keyword** dynamic এবং execution context এর উপর নির্ভর করে। এটি mastering করলে JavaScript এর object-oriented programming সহজ এবং কার্যকর হয়।

JavaScript Arrow Function

JavaScript এর **Arrow Functions** হলো একটি **concise syntax** যা ES6 (ECMAScript 2015) এ প্রবর্তিত হয়। এটি **function expression** লিখতে সহজ করে এবং **lexical this binding** (parent scope থেকে this inherit করা) এর একটি গুরুত্বপূর্ণ feature প্রদান করে।

Table of Contents

---

1. 📘 What is an Arrow Function?

Arrow Function হলো JavaScript এর একটি shorthand syntax যা সাধারণ function expression এর চেয়ে ছোট এবং readable। এটি function keyword বাদ দিয়ে, => (arrow) ব্যবহার করে function লেখার একটি উপায়।

---

2. 🛠️ Syntax of Arrow Functions

Basic Syntax:

```
const functionName = (parameters) => expression;
```

Example:

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

When No Parameters:

```
const greet = () => "Hello!";
console.log(greet()); // Output: Hello!
```

Single Parameter (Parentheses Optional):

```
const square = x => x * x;
console.log(square(4)); // Output: 16
```

Multiple Parameters (Parentheses Required):

```
const multiply = (a, b) => a * b;
console.log(multiply(3, 5)); // Output: 15
```

Multi-line Arrow Functions:

```
const sum = (a, b) => {
  const result = a + b;
  return result;
};
console.log(sum(4, 6)); // Output: 10
```

---

3. 🧐 Why Use Arrow Functions?

1. **Concise Syntax**:
   - Function expressions ছোট এবং cleaner হয়।
2. **Lexical this Binding**:
   - Arrow functions automatically parent scope থেকে this কে inherit করে।
3. **No arguments Object**:
   - Regular functions এর মতো arguments object নেই।
4. **Improved Readability**:
   - Shorter syntax কোডকে সহজে পড়তে সহায়তা করে।

---

4. 🔄 Key Features of Arrow Functions

1. **Lexical this Binding**:
   - Arrow functions এর this parent scope থেকে inherit করে।
2. **No arguments Object**:
   - Arrow functions এর নিজস্ব arguments object থাকে না।
3. **Cannot be Used as Constructors**:
   - Arrow functions new keyword দিয়ে instantiate করা যায় না।
4. **Implicit Return**:
   - Single-line expressions এ return keyword না দিলেও value return হয়।

5. 📖 Examples with Explanation

---

1. Basic Syntax

```
const greet = (name) => `Hello, ${name}!`;
console.log(greet("Alice")); // Output: Hello, Alice!
```

Explanation:

- এখানে single-line arrow function ব্যবহার করা হয়েছে।
- return keyword প্রয়োজন হয়নি কারণ এটি implicit return করছে।

---

2. Single-line Arrow Functions

```
const isEven = num => num % 2 === 0;
console.log(isEven(4)); // Output: true
console.log(isEven(7)); // Output: false
```

Explanation:

- Single parameter থাকলে parentheses (()) প্রয়োজন হয় না।
- return keyword বাদ দিয়ে সরাসরি expression লেখা হয়েছে।

---

3. Multi-line Arrow Functions

```
const findMax = (a, b) => {
  if (a > b) return a;
  return b;
};
console.log(findMax(10, 5)); // Output: 10
```

Explanation:

- Multi-line logic এর জন্য curly braces {} ব্যবহার করা হয়েছে।
- Explicit return ব্যবহার করা হয়েছে।

---

4. Lexical this Binding

```
const obj = {
  name: "Alice",
  greet: function () {
    const arrow = () => console.log(`Hello, ${this.name}`);
    arrow();
  },
};
obj.greet(); // Output: Hello, Alice
```

Explanation:

- Arrow function তার parent scope (greet function) থেকে this inherit করেছে।
- Regular functions এর ক্ষেত্রে this undefined হতে পারে।

---

6. 🔄 Arrow Functions vs Regular Functions

Feature	**Arrow Function**	**Regular Function**
Syntax	Concise	Verbose
this Binding	Lexical (parent scope থেকে নেয়)	Dynamic (যেখানে call হয়, সেখানে সেট হয়)
arguments Object	No	Yes
Constructors	Cannot be used as constructors	Used as constructors

---

Example of this Difference:

```
// Regular Function
function regularFunc() {
  console.log(this);
}

// Arrow Function
const arrowFunc = () => {
  console.log(this);
};

regularFunc(); // Depends on the caller
arrowFunc();   // Lexical parent scope's `this`
```

---

7. 🚫 When Not to Use Arrow Functions

1. **Object Methods**:
   - Arrow functions এর this parent scope থেকে inherit করে, যার ফলে object methods এ এর ব্যবহার সমস্যার সৃষ্টি করতে পারে।

Example:

```
const obj = {
  name: "Alice",
  greet: () => {
    console.log(this.name); // Output: undefined
  },
};
obj.greet();
```

Solution:

```
const obj = {
  name: "Alice",
  greet: function () {
    console.log(this.name); // Output: Alice
  },
};
obj.greet();
```

---

2. **Dynamic this Required**:
   - যখন this dynamically call context এর উপর নির্ভর করবে।

Example:

```
function Timer() {
  this.seconds = 0;

  setInterval(() => {
    this.seconds++;
    console.log(this.seconds);
  }, 1000);
}

new Timer();
```

---

3. **Constructors**:
   - Arrow functions constructor হিসেবে কাজ করে না।

Example:

```
const Person = (name) => {
  this.name = name;
};
const john = new Person("John"); // Error: Person is not a constructor
```

---

8. ✅ Best Practices

1. **Use for Short and Simple Functions**:
   - ছোট function expressions এর জন্য arrow functions আদর্শ।
2. **Avoid in Object Methods**:
   - Object methods এর জন্য regular function ব্যবহার করুন।
3. **Use in Callbacks**:
   - Callbacks বা event handlers এ arrow functions ব্যবহার করুন।
4. const numbers = [1, 2, 3];

const squares = numbers.map((n) => n * n);

5. **Readable Code**:
    - Multi-line logic এর জন্য curly braces এবং explicit return ব্যবহার করুন।

Summary

JavaScript এর **Arrow Functions** কোড লেখাকে সহজ, সংক্ষিপ্ত এবং readable করে তোলে।
এটি **lexical this binding**, **implicit return**, এবং **modern syntax** এর কারণে React, Node.js, এবং অন্যান্য frameworks এ অত্যন্ত জনপ্রিয়।

↥ Go to Top

Chapter-12: JavaScript Classes, JSON and Debugging

JavaScript Classes

📋 Table of Contents

---

1. What is a JavaScript Class? 🎯

JavaScript এ **Class** হলো একটা **Blueprint** বা **Template**, যেটা দিয়ে আমরা **Object** তৈরি করি।
একটা Class নির্ধারণ করে — কিভাবে একটা Object behave করবে এবং তার কী কী Properties এবং Methods থাকবে।

🔵 Example:

যেমন ধরুন, একটা **Car Factory** আছে যেখানে সব গাড়ির Structure একই হলেও Color, Model আলাদা হয়।
এই Factory-এর নকশাটাই হচ্ছে **Class**।

---

2. Why use Classes? ❓

- Code কে **organized** ও **manageable** করতে সাহায্য করে।
- **Reusability** বাড়ায় — বারবার একই জিনিস না লিখে আগের Class use করা যায়।
- **Big Projects** এ কোড Maintain করা সহজ হয়।
- নতুন Object তৈরি করা হয় একদম **clear and efficient** way-তে।

---

3. How to Create a Class 🏗️

JavaScript এ class keyword ব্যবহার করে ক্লাস তৈরি করা হয়:

```
class Car {
   constructor(name, model) {
      this.name = name;
      this.model = model;
   }
}
```

👉 এখানে:

- Car হলো Class-এর নাম।
- constructor() হলো Special Method যা Object তৈরি সময় অটো-কল হয়।
- this.name ও this.model হচ্ছে Object-এর Property।

---

4. Constructor Method 🚗

constructor() হলো Special Method যেটা নতুন Object তৈরি করার সময় অটোমেটিক চালু হয়।

🔵 Example:

```
class Student {
   constructor(name, age) {
      this.name = name;
      this.age = age;
   }
}

const student1 = new Student("Rahim", 20);
console.log(student1);
// Output: Student { name: 'Rahim', age: 20 }
```

---

5. Adding Methods inside a Class 🔧

আমরা ক্লাসের ভিতরে নতুন নতুন Methods তৈরি করতে পারি:

```
class Car {
   constructor(name) {
      this.name = name;
   }

   start() {
      console.log(`${this.name} has started.`);
   }
}

const myCar = new Car("Toyota");
myCar.start();
// Output: Toyota has started.
```

---

6. Creating Objects from Class 🧩

new keyword দিয়ে আমরা Class থেকে Object তৈরি করি:

```
const car1 = new Car("Honda");
const car2 = new Car("BMW");

console.log(car1.name); // Honda
console.log(car2.name); // BMW
```

---

7. Class Inheritance (Extending Classes) 🧬

Inheritance মানে হলো — একটা ক্লাসের Property এবং Methods অন্য ক্লাসে **reuse** করা।
JavaScript এ extends keyword দিয়ে এটা করা হয়।

🔵 Example:

```
class Vehicle {
  constructor(brand) {
    this.brand = brand;
  }

  showBrand() {
    console.log(`Brand is: ${this.brand}`);
  }
}

class Bike extends Vehicle {
  constructor(brand, model) {
    super(brand);
    this.model = model;
  }

  showDetails() {
        console.log(`Brand: ${this.brand}, Model: ${this.model}`);
    }
}

const myBike = new Bike("Yamaha", "R15");
myBike.showBrand();   // Brand is: Yamaha
myBike.showDetails(); // Brand: Yamaha, Model: R15
```

---

8. Super Keyword Explained 🏎️

super() keyword ব্যবহার করে আমরা Parent Class-এর Constructor অথবা Method call করতে পারি।

উদাহরণঃ উপরের কোডে super(brand) দিয়ে **Vehicle** ক্লাসের constructor call করা হয়েছে।

---

9. Getters and Setters 💥

Getter দিয়ে আমরা Object-এর Property সহজে পড়তে পারি আর **Setter** দিয়ে সেট করতে পারি।

🔵 Example:

```
class Person {
    constructor(name) {
        this._name = name;
    }

    get name() {
        return this._name;
    }

    set name(newName) {
        this._name = newName;
    }
}
```

```
const person1 = new Person("Alim");
console.log(person1.name); // Alim

person1.name = "Hasan";
console.log(person1.name); // Hasan
```

---

10. Real Life Examples (2 Practical Examples) 🌍

Example 1: Bank Account Management

```
class BankAccount {
  constructor(owner, balance) {
    this.owner = owner;
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
    console.log(`Deposited ${amount}. New balance is ${this.balance}.`);
  }

  withdraw(amount) {
    if (amount > this.balance) {
      console.log("Insufficient Balance!");
    } else {
      this.balance -= amount;
      console.log(`Withdrew ${amount}. New balance is ${this.balance}.`);
    }
  }
}
```

```
const account = new BankAccount("Rahim", 5000);
account.deposit(2000);
account.withdraw(1000);
account.withdraw(7000);
```

Example 2: Library Book Management

```
class Book {
   constructor(title, author) {
      this.title = title;
      this.author = author;
   }

   showDetails() {
      console.log(`Title: ${this.title}, Author: ${this.author}`);
   }
}

const book1 = new Book("Learn JavaScript", "Alim");
const book2 = new Book("Mastering PHP", "Hasan");

book1.showDetails();
book2.showDetails();
```

---

11. Important Points about JavaScript Classes ⚡

✅ Class আসলে একটা Special Type of Function।

✅ Class এর নাম Capital Letter দিয়ে শুরু করা উচিত।

✅ Constructor method প্রতি Class-এ শুধুমাত্র একবারই define করা যায়।

✅ Inheritance ব্যবহার করে Code Reusability বাড়ানো যায়।

✅ Class ব্যবহার করে Clean, Manageable এবং Scalable Code লেখা যায়।

JavaScript JSON

📋 Table of Contents

---

1. What is JSON? 🎯

JSON এর ফুল ফর্ম হলো **JavaScript Object Notation**।

এটা হলো একটা **Data format**, যেটা ব্যবহার করা হয় **Data Store** এবং **Data Transfer** করার জন্য।

Short form এ বললে:

JSON = Simple text format, যা Human এবং Machine দুইজনই সহজে পড়তে পারে।

🔵 Example of JSON:

```
{
   "name": "Alim",
   "age": 25,
   "city": "Dhaka"
}
```

এটা দেখতে অনেকটা JavaScript Object এর মতো হলেও, এটা হচ্ছে **pure text** format!

---

2. Why use JSON? ❓

- **Data transfer** করার জন্য (specially Server ↔ Client communication)
- **Configuration file** হিসেবে (যেমনঃ package.json)
- **Database** এর ভিতর Data store করার জন্য (যেমনঃ MongoDB JSON-like format ইউজ করে)

- **APIs** response বা request format হিসেবে ব্যবহার হয় (e.g., REST APIs)

---

3. JSON Syntax Rules 📋

JSON লেখার সময় কিছু Rule মেনে চলতে হয়:

✅ Data থাকে **Key/Value** pair আকারে

✅ Key অবশ্যই **double quotes ("")** দিয়ে লিখতে হবে

✅ Data types হতে পারে: String, Number, Object, Array, Boolean, Null

✅ JSON file বা string এর শুরু এবং শেষ হয় **{ }** দিয়ে (object), অথবা **[ ]** দিয়ে (array)

🔵 Example:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "isStudent": true,
  "age": 22,
  "courses": ["Math", "Science", "History"]
}
```

---

4. JSON vs JavaScript Object 🔥

Aspect	**JSON**	**JavaScript Object**
Syntax	Always keys in double quotes	Keys without quotes allowed
Data Type	Only String, Number, Array, Boolean, Null, Object	Functions, Undefined allowed
Format	Pure Text	JavaScript Code

🔵 Example:

👉 **JavaScript Object:**

```
const student = {
    name: "Alim",
    age: 25,
    isStudent: true
};
```

👉 **JSON:**

```
{
    "name": "Alim",
    "age": 25,
    "isStudent": true
}
```

---

5. How to Convert JavaScript Object to JSON (Stringify) 🔄

JavaScript Object কে JSON String এ রূপান্তর করতে আমরা ব্যবহার করি:

👉 JSON.stringify()

🔵 Example:

```
const student = { name: "Alim", age: 25, city: "Dhaka" };
const jsonString = JSON.stringify(student);

console.log(jsonString);
// Output: {"name":"Alim","age":25,"city":"Dhaka"}
```

এখানে পুরো Object টাকে একটা String হিসেবে রূপান্তর করা হয়েছে।

---

6. How to Convert JSON to JavaScript Object (Parse) 🔄

JSON String কে আবার JavaScript Object এ রূপান্তর করতে ব্যবহার করি:

👉 JSON.parse()

🔵 Example:

```
const jsonString = '{"name":"Alim","age":25,"city":"Dhaka"}';
const student = JSON.parse(jsonString);
```

```
console.log(student);
// Output: { name: 'Alim', age: 25, city: 'Dhaka' }
```

এখানে JSON থেকে Real JavaScript Object পাওয়া গেছে।

---

7. Real Life Examples 🌍

Example 1: Fetching Data from an API

```
fetch('https://api.example.com/users')
  .then(response => response.json())
  .then(data => console.log(data));
```

এখানে Server থেকে JSON format এ Data আসে, যেটাকে আমরা .json() method দিয়ে JavaScript Object বানিয়ে ব্যবহার করি।

Example 2: Storing Data in LocalStorage

```
const user = { name: "Alim", age: 25 };
localStorage.setItem("user", JSON.stringify(user));

const storedUser = JSON.parse(localStorage.getItem("user"));
console.log(storedUser);
// Output: { name: 'Alim', age: 25 }
```

LocalStorage এ Data রাখতে হলে প্রথমে JSON.stringify() করে রাখতে হয়, এবং পড়ার সময় JSON.parse() করে নিতে হয়।

---

8. Important Points about JSON ⚡

✅ JSON শুধুমাত্র **Data store** এবং **Data transfer** করার জন্য ব্যবহৃত হয়।

✅ JSON এর মধ্যে Function বা Comments লেখা যায় না।

✅ JSON ফাইলের Extension সাধারণত .json হয়।

✅ সব Major Programming Language (Python, PHP, Java, C# etc.) JSON ব্যবহার করতে পারে।

JavaScript Debugging

📋 Table of Contents

---

1. What is Debugging? 🐞

Debugging মানে হলো — কোডের মধ্যে থাকা **Bug (ভুল/সমস্যা)** খুঁজে বের করা এবং সেটা **ঠিক করা।**

Shortly:
Debugging = Find problems + Fix problems

👉 যখন আমাদের Code ঠিকমতো কাজ করে না, তখন Debugging করে দেখা হয় কোথায় ভুল হয়েছে।

---

2. Why Debugging is Important? ❓

- **Error-free Application** তৈরি করার জন্য।
- কোডের Performance এবং Reliability বাড়ানোর জন্য।
- Development Process কে আরো Smooth করার জন্য।
- Bigger Projects এ যদি ছোট ছোট bug ঠিক করা না হয়, পরে বড় সমস্যা তৈরি হয়।

---

3. Common Debugging Techniques 🛠️

✅ **console.log()** দিয়ে Variable এর মান দেখা
✅ **Developer Tools** ব্যবহার করে Error Details বোঝা
✅ **Breakpoints** দিয়ে Specific Line এ কোড Pause করে Analysis করা
✅ **Try-Catch** ব্যবহার করে Error Handling করা
✅ **Debugger Statement** দিয়ে Manually Debugging চালু করা

---

4. Using console.log() for Debugging 🖥️

সবচেয়ে Simple এবং Powerful টেকনিক হচ্ছে **console.log()** ব্যবহার করে কোডের ভিতরের Data বা Flow দেখতে পাওয়া।

🔵 Example:

```
function calculateSum(a, b) {
   console.log("Value of a:", a);
   console.log("Value of b:", b);
   return a + b;
}

calculateSum(5, 10);
```

Output:

```
Value of a: 5
Value of b: 10
```

👉 এইভাবে বুঝা যাবে কোন Variable এ ঠিকমতো Value আসছে কিনা।

---

5. Using Browser Developer Tools 🔥

Almost সব Modern Browser (যেমনঃ Chrome, Firefox) এ থাকে **Developer Tools**।
Developer Tools এর মাধ্যমে:

✅ Console Log দেখতে পারি

✅ Errors ও Warnings Check করতে পারি

✅ Source Code Inspect করতে পারি

✅ Breakpoints Set করে Step-by-Step Code Run করতে পারি

🔵 Chrome এ Developer Tools Open করার Shortcut:

- Press **F12** অথবা **Ctrl + Shift + I**

---

6. Breakpoints in Debugging 🎯

Breakpoints ব্যবহার করে আমরা Code Execution pause করাতে পারি নির্দিষ্ট জায়গায়।
তারপর Line-by-Line চেক করতে পারি কোড ঠিকমতো কাজ করছে কিনা।

🔵 Example Flow:

1. Developer Tools খুলুন।
2. Sources Tab এ যান।
3. যেই Line এ Pause করতে চান, সেখানে Click করে Breakpoint বসান।
4. এখন Page Reload করলে Code ঐ Line এ এসে Pause করবে।
5. এরপর Step-by-Step Code Analyze করতে পারবেন।

---

7. The Debugger Keyword 🧩

debugger keyword দিলে যেখানে এই লাইন থাকবে, Browser অটোমেটিক Debugging Mode চালু করবে।

🔵 Example:

```
function testDebug(x) {
   debugger;
   console.log("Value of x:", x);
}

testDebug(100);
```

👉 যখন এই Function চালাবেন, Browser Stop করে Debugging Start করবে যেখানে debugger আছে।

---

8. Real Life Debugging Example 🌍

Suppose আপনি একটি Function লিখেছেন, যেটা দুইটি সংখ্যার গুণফল বের করার কথা ছিল, কিন্তু ভুল করে যোগফল বের করছে:

🔵 Problematic Code:

```
function multiply(a, b) {
   console.log("a:", a);
   console.log("b:", b);
```

```
    return a + b; // Mistake here
}

console.log(multiply(5, 3));
// Output: 8, but expected 15
```

👉 Debugging করলে আমরা বুঝতে পারবো এখানে + থাকা উচিত ছিল * |

🔵 Fixed Code:

```
function multiply(a, b) {
    console.log("a:", a);
    console.log("b:", b);
    return a * b;
}

console.log(multiply(5, 3));
// Output: 15
```

---

9. Best Practices for Debugging ⚡

✅ Debugging করার সময় Problem টাকে ছোট ছোট অংশে ভাগ করুন|

✅ Clear এবং Simple console.logs ব্যবহার করুন|

✅ Browser DevTools এর Sources Tab এ Breakpoints ব্যবহার করুন|

✅ Real Device অথবা Real Browser Environment এ Test করুন|

✅ কখনো কখনো Fresh Mind নিয়ে নতুন করে Code পড়লে ভুল ধরা পড়ে!

↥ Go to Top

Chapter-13: JavaScript Object in Detail

JavaScript Object Definition

Table of Contents

What is a JavaScript Object?

A JavaScript object is a standalone entity that holds properties and values. Properties are the values associated with an object, and these properties can be of any data type, including other objects, functions, or arrays. In essence, an object is a collection of key-value pairs where the keys are strings (referred to as properties) and the values can be any type of data.

JavaScript objects are fundamental building blocks used to store and manage data in a structured way. They are similar to real-world objects, where properties define the characteristics of the object.

Creating JavaScript Objects

Object Literals

The most straightforward way to create an object in JavaScript is by using object literals. You can define an object by specifying its properties and values within curly braces {}.

let car = {

```
  brand: "Toyota",
  model: "Corolla",
  year: 2021,
};
```

Using the new Object() Syntax

Another approach to creating an object is using the new Object() constructor. This method is less common but still valid.

```
let car = new Object();
car.brand = "Toyota";
car.model = "Corolla";
car.year = 2021;
```

Using Object Constructors

You can define a custom constructor function to create objects with a specific structure. This method allows you to create multiple objects with similar properties.

```
function Car(brand, model, year) {
  this.brand = brand;
  this.model = model;
  this.year = year;
}

let myCar = new Car("Toyota", "Corolla", 2021);
```

Accessing Object Properties

You can access properties of a JavaScript object using two main methods: dot notation and bracket notation.

Dot Notation

Dot notation is the most common way to access properties. It involves using a period . followed by the property name.

```
let car = {
  brand: "Toyota",
  model: "Corolla",
  year: 2021,
};

console.log(car.brand); // Outputs: Toyota
```

Bracket Notation

Bracket notation is useful when the property name is dynamic or not a valid JavaScript identifier (e.g., contains spaces or starts with a number).

```
let car = {
  "car brand": "Toyota",
  model: "Corolla",
  year: 2021,
};

console.log(car["car brand"]); // Outputs: Toyota
```

Adding, Modifying, and Deleting Properties

You can add new properties, modify existing ones, or delete properties from an object.

```
let car = {
  brand: "Toyota",
  model: "Corolla",
};

// Adding a new property
car.year = 2021;

// Modifying an existing property
car.model = "Camry";

// Deleting a property
delete car.year;
```

```
console.log(car); // Outputs: { brand: "Toyota", model: "Camry" }
```

JavaScript Object Methods

JavaScript objects can have methods, which are functions stored as properties within the object. These methods can perform actions using the object's data.

```
let car = {
  brand: "Toyota",
  model: "Corolla",
  year: 2021,
  displayInfo: function () {
    console.log(this.brand + " " + this.model + " (" + this.year + ")");
  },
};

car.displayInfo(); // Outputs: Toyota Corolla (2021)
```

JavaScript Object Method Table

Method	Description	Example Code	Output/Result
Object.assign()	Copies properties from one or more source objects to a target object.	Object.assign(target, source);	Combines properties of source into target.
Object.keys()	Returns an array of a given object's own property names.	Object.keys(car);	["brand", "model", "year"]
Object.values()	Returns an array of a given	Object.values(car);	["Toyota", "Corolla", 2021]

Method	Description	Example Code	Output/Result
	object's own property values.		
Object.entries()	Returns an array of a given object's own key-value pairs.	Object.entries(car);	[["brand", "Toyota"], ["model", "Corolla"], ["year", 2021]]
Object.freeze()	Freezes an object, preventing new properties from being added or existing properties from being modified or deleted.	Object.freeze(car);	The car object is now immutable.
Object.seal()	Seals an object, preventing new properties from being added, but allowing modification of existing properties.	Object.seal(car);	New properties can't be added, but existing ones can be modified.

Examples of Object Usage

Example 1: Defining a Person Object

```
let person = {
  firstName: "John",
  lastName: "Doe",
  age: 25,
  greet: function () {
    console.log("Hello, my name is " + this.firstName + " " + this.lastName);
  },
};

person.greet(); // Outputs: Hello, my name is John Doe
```

Example 2: Creating a Method to Calculate Age

```
let person = {
  firstName: "John",
  lastName: "Doe",
  birthYear: 1995,
  calculateAge: function () {
    let currentYear = new Date().getFullYear();
    return currentYear - this.birthYear;
  },
};

console.log(person.calculateAge()); // Outputs the current age based on birthYear
```

JavaScript objects are versatile and powerful, enabling developers to structure and manipulate data efficiently. Understanding the fundamentals of objects, including creation, property management, and method utilization, is crucial for mastering JavaScript.

JavaScript Object Prototypes

Table of Contents

What is a JavaScript Prototype?

জাভাস্ক্রিপ্টে, প্রতিটি অবজেক্টের একটি প্রোটোটাইপ রয়েছে। একটি প্রোটোটাইপও একটি অবজেক্ট, এবং এটি একটি টেমপ্লেট হিসাবে কাজ করে যা থেকে অন্যান্য অবজেক্টগুলোর Properties এবং Methods এর উত্তরাধিকারী হতে পারে। আপনি যখন একটি অবজেক্ট তৈরি করেন, জাভাস্ক্রিপ্ট স্বয়ংক্রিয়ভাবে এটি একটি প্রোটোটাইপ অবজেক্টের সাথে লিঙ্ক করে। এটি প্রতিটি অবজেক্টের মধ্যে সরাসরি Define না করেই Objects গুলোর মধ্যে Share করা Properties এবং Methods ব্যবহারের জন্য অনুমতি দেয়।

JavaScript uses prototypes to implement inheritance, which is a way for an object to access properties and methods from another object. This mechanism is central to how JavaScript handles object-oriented programming.

Prototype Chain

The prototype chain is a series of links between objects where one object’s prototype is linked to another object’s prototype. This chain continues until it reaches null, which signifies the end of the chain.

When you attempt to access a property or method on an object, JavaScript first checks if the property or method exists on that object. If it doesn't, JavaScript checks the object's prototype, and if it's not found there, it continues up the prototype chain until it either finds the property or method or reaches the end of the chain (null).

```
let car = {
  brand: "Toyota",
};


console.log(car.toString()); // Outputs: [object Object]
```

In the example above, the toString method isn't directly defined in the car object, but it is found in the Object.prototype, which is part of the prototype chain.

Understanding __proto__ and prototype

- **__proto__:** Every object in JavaScript has a __proto__ property that points to the prototype object it was inherited from. This property is often used for accessing the prototype directly.

- **prototype:** The prototype property is associated with functions, specifically constructor functions. When you create an object using a constructor function, that object's __proto__ is set to the constructor function's prototype.

```
function Car(brand, model) {
  this.brand = brand;
  this.model = model;
}

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

console.log(myCar.__proto__ === Car.prototype); // Outputs: true
```

Adding Properties and Methods to Prototypes

You can add properties and methods to an object's prototype even after the object has been created. This allows all instances of that object to share these properties and methods.

```
function Car(brand, model) {
  this.brand = brand;
  this.model = model;
}

// Adding a method to the prototype
Car.prototype.displayInfo = function () {
  console.log(this.brand + " " + this.model);
};

let myCar = new Car("Toyota", "Corolla");
myCar.displayInfo(); // Outputs: Toyota Corolla
```

Prototype Inheritance

JavaScript objects can inherit properties and methods from other objects through prototypes. This is known as prototype inheritance. It allows one object to acquire the properties and methods of another object.

```
function Vehicle(type) {
  this.type = type;
}

Vehicle.prototype.start = function () {
  console.log("Starting the " + this.type);
};

function Car(brand, model) {
  Vehicle.call(this, "car");
  this.brand = brand;
  this.model = model;
}

Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.constructor = Car;

let myCar = new Car("Toyota", "Corolla");
myCar.start(); // Outputs: Starting the car
```

In this example, the Car object inherits the start method from the Vehicle object.

JavaScript Prototype Method Table

Method	**Description**	**Example Code**	**Output/Result**
Object.getPrototypeOf()	Returns the prototype of the	Object.getPrototypeOf(myCar);	Returns the prototype object of myCar.

Method	Description	Example Code	Output/Result
	specified object.		
Object.setPrototypeOf()	Sets the prototype (i.e., __proto__) of a specified object to another object or null.	Object.setPrototypeOf(obj, prototypeObj);	Changes the prototype of obj.
Object.create()	Creates a new object with the specified prototype object and properties.	Object.create(proto, props);	Creates a new object that inherits from proto.
hasOwnProperty()	Returns a boolean indicating whether the object has the specified property as its own property (not inherited).	obj.hasOwnProperty('prop');	true if obj has prop as its own property.

Method	Description	Example Code	Output/Result
isPrototypeOf()	Checks if an object exists in another object's prototype chain.	prototypeObj.isPrototypeOf(obj);	true if prototypeObj is in obj's prototype chain.

Examples of Prototype Usage

Example 1: Inheriting Methods from a Prototype

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function () {
  console.log("Hello, my name is " + this.name);
};

let john = new Person("John", 30);
john.greet(); // Outputs: Hello, my name is John
```

Example 2: Modifying Prototypes

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

Animal.prototype.speak = function () {
  console.log(this.species + " makes a sound");
};

let dog = new Animal("Dog");
dog.speak(); // Outputs: Dog makes a sound

// Adding a new method to the prototype
Animal.prototype.eat = function () {
  console.log(this.species + " is eating");
};

dog.eat(); // Outputs: Dog is eating
```

In this example, the Animal object is extended with a new method eat, which is then available to all instances of Animal.

JavaScript prototypes are a powerful feature that allows for inheritance and sharing of methods and properties across objects. By understanding how to work with prototypes, you can create more efficient and organized code, leveraging JavaScript's object-oriented capabilities to their fullest.

JavaScript Object Methods

Table of Contents

What are JavaScript Object Methods?

JavaScript object methods are functions that are properties of an object. These methods are used to perform actions on the data within the object. JavaScript provides built-in object methods to manipulate, interact with, and access the properties and values of objects. By using these methods, developers can effectively manage and manipulate objects in JavaScript.

Common JavaScript Object Methods

Here are some of the most commonly used JavaScript object methods:

- **Object.keys()**: Returns an array containing the names of all the object's own enumerable properties.
- **Object.values()**: Returns an array containing the values of all the object's own enumerable properties.
- **Object.entries()**: Returns an array of the object's own enumerable properties in key-value pair format.
- **Object.assign()**: Copies all enumerable properties from one or more source objects to a target object.
- **Object.freeze()**: Freezes an object, preventing new properties from being added, existing properties from being removed, or any changes to property values.
- **Object.seal()**: Seals an object, preventing new properties from being added and making existing properties non-configurable.
- **Object.create()**: Creates a new object with a specified prototype object and properties.
- **Object.hasOwnProperty()**: Returns a boolean indicating whether the object has the specified property as its own property.
- **Object.is()**: Compares two values to determine if they are the same.
- **Object.defineProperty()**: Adds a new property directly to an object or modifies an existing property on an object.
- **Object.defineProperties()**: Adds multiple properties directly to an object or modifies existing properties on an object.

JavaScript Object Method Table

Method	**Description**	**Example Code**	**Output/Result**
Object.keys()	Returns an array of the object's own enumerable	javascript let obj = {a: 1, b: 2}; Object.keys(obj);	["a", "b"]

Method	Descriptio n	Example Code	Output/Res ult
	property names.		
Object.values()	Returns an array of the object’s own enumerable property values.	javascript let obj = {a: 1, b: 2}; Object.values(obj);	[1, 2]
Object.entries()	Returns an array of the object’s own enumerable key-value pairs.	javascript let obj = {a: 1, b: 2}; Object.entries(obj);	[['a', 1], ['b', 2]]
Object.assign()	Copies the values of all enumerable own properties from one or more source objects to a target object.	javascript let obj1 = {a: 1}; let obj2 = {b: 2}; Object.assign(obj1, obj2);	{a: 1, b: 2}
Object.freeze()	Freezes the object, preventing any	javascript let obj = {a: 1}; Object.freeze(obj); obj.a = 2;	{a: 1}

Method	Description	Example Code	Output/Result
	changes to it.		
Object.seal()	Seals the object, preventing new properties from being added but allowing modification of existing properties.	javascript let obj = {a: 1}; Object.seal(obj); obj.b = 2;	{a: 1}
Object.create()	Creates a new object with the specified prototype object and properties.	javascript let proto = {a: 1}; let obj = Object.create(proto);	An object linked to proto.
Object.hasOwnProperty()	Returns true if the object has the specified property as its own property.	javascript let obj = {a: 1}; obj.hasOwnProperty('a');	true
Object.is()	Compares two values to determine if	javascript Object.is(0, 0); Object.is(NaN, NaN);	true for both

Method	Description	Example Code	Output/Result
	they are the same.		
Object.defineProperty()	Adds or modifies a property directly on an object.	javascript let obj = {}; Object.defineProperty(obj, 'a', {value: 42});	{a: 42}
Object.defineProperties()	Adds or modifies multiple properties directly on an object.	javascript let obj = {}; Object.defineProperties (obj, {a: {value: 42}, b: {value: 36}});	{a: 42, b: 36}

Examples of Using Object Methods

Example 1: Using Object.keys() and Object.values()

```
let car = {
  brand: "Toyota",
  model: "Corolla",
  year: 2021,
};

let keys = Object.keys(car);
console.log(keys); // Outputs: ["brand", "model", "year"]

let values = Object.values(car);
console.log(values); // Outputs: ["Toyota", "Corolla", 2021]
```

Example 2: Using Object.entries()

```
let user = {
  name: "John",
  age: 30,
};

let entries = Object.entries(user);
console.log(entries); // Outputs: [["name", "John"], ["age", 30]]
```

Example 3: Using Object.assign() to Merge Objects

```
let obj1 = { a: 1, b: 2 };
let obj2 = { b: 3, c: 4 };

let mergedObj = Object.assign({}, obj1, obj2);
console.log(mergedObj); // Outputs: { a: 1, b: 3, c: 4 }
```

Example 4: Using Object.freeze() and Object.seal()

```
let car = {
  brand: "Toyota",
  model: "Corolla",
  year: 2021,
};

// Freezing the object
Object.freeze(car);
car.year = 2022; // This will have no effect
console.log(car.year); // Outputs: 2021

// Sealing the object
let bike = {
  brand: "Yamaha",
  model: "FZ",
};
```

```
Object.seal(bike);
bike.model = "FZS"; // This is allowed
bike.year = 2021; // This will have no effect as adding new properties is not allowed
console.log(bike); // Outputs: { brand: "Yamaha", model: "FZS" }
```

Example 5: Using Object.create()

```
let vehicle = {
  type: "Vehicle",
  displayType: function () {
    console.log(this.type);
  },
};

let car = Object.create(vehicle);
car.type = "Car";
car.displayType(); // Outputs: Car
```

Example 6: Using Object.hasOwnProperty()

```
let person = {
  name: "Alice",
  age: 25,
};

console.log(person.hasOwnProperty("name")); // Outputs: true
console.log(person.hasOwnProperty("gender")); // Outputs: false
```

Example 7: Using Object.is()

```
console.log(Object.is(25, 25)); // Outputs: true
console.log(Object.is(NaN, NaN)); // Outputs: true
console.log(Object.is(0, -0)); // Outputs: false
```

Example 8: Using Object.defineProperty()

```
let obj = {};
Object.defineProperty(obj, "a", {
  value: 42,
  writable: false,
});

console.log(obj.a); // Outputs: 42
obj.a = 25; // This will not change the value
console.log(obj.a); // Still outputs: 42
```

Example 9: Using Object.defineProperties()

```
let person = {};

Object.defineProperties(person, {
  name: {
    value: "John",
    writable: true,
  },
  age: {
    value: 30,
    writable: false,
  },
});

console.log(person.name); // Outputs: John
console.log(person.age); // Outputs: 30

person.age = 35; // This will not change the value
console.log(person.age); // Still outputs: 30
```

By using these object methods effectively, you can manage and manipulate objects in JavaScript more efficiently, allowing for cleaner and more maintainable code.

JavaScript Object Properties

Table of Contents

What are JavaScript Object Properties?

JavaScript object properties are key-value pairs where the key is a string (property name) and the value can be any data type. These properties define the characteristics and behavior of an object. They allow you to store and access data within an object.

Types of JavaScript Object Properties

JavaScript properties can be categorized into two main types:

1. **Data Properties**: These are the most common type of properties, which store data values. Each data property has the following attributes:
   - **value**: The value of the property.
   - **writable**: A boolean indicating if the property can be changed.
   - **enumerable**: A boolean indicating if the property can be iterated over.
   - **configurable**: A boolean indicating if the property can be deleted or changed.
2. **Accessor Properties**: These properties are defined by getter and setter methods. They don't store a value directly but compute it based on other values.
   - **get**: A function that is called when the property is accessed.
   - **set**: A function that is called when the property is set.

JavaScript Object Property Table

Property Type	**Description**	**Example Code**	**Output/Result**
Data Property	Stores a value. Has attributes: value, writable, enumerable, and configurable.	javascript let obj = {name: "Alice"}; obj.name = "Bob";	"Bob"

Property Type	Description	Example Code	Output/Result
Accessor Property	Defined by getter and setter methods. It does not store a value directly.	javascript let obj = { get fullName() { return "John Doe"; } }; console.log(obj.fullName);	"John Doe"

Examples of Using Object Properties

Example 1: Defining and Accessing Data Properties

```
let person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
};

console.log(person.firstName); // Outputs: John
console.log(person.age); // Outputs: 30

// Modifying a data property
person.age = 31;
console.log(person.age); // Outputs: 31
```

Example 2: Defining and Accessing Accessor Properties

```
let user = {
  firstName: "Alice",
  lastName: "Johnson",
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  set fullName(name) {
    let parts = name.split(" ");
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
};

// Accessing the full name using getter
console.log(user.fullName); // Outputs: Alice Johnson

// Setting the full name using setter
user.fullName = "Bob Smith";
console.log(user.firstName); // Outputs: Bob
console.log(user.lastName); // Outputs: Smith
```

Example 3: Using Object.defineProperty() to Control Property Attributes

```
let car = {};

Object.defineProperty(car, "brand", {
  value: "Toyota",
  writable: false, // The brand cannot be changed
  enumerable: true, // The brand will appear in a for-in loop
  configurable: true, // The brand property can be deleted or changed
});

console.log(car.brand); // Outputs: Toyota
car.brand = "Honda"; // This will not change the value
console.log(car.brand); // Still outputs: Toyota
```

Example 4: Using Object.defineProperties() to Define Multiple Properties

```
let book = {};
```

```
Object.defineProperties(book, {
  title: {
    value: "JavaScript: The Good Parts",
    writable: true,
  },
  author: {
    value: "Douglas Crockford",
    writable: false,
  },
});

console.log(book.title); // Outputs: JavaScript: The Good Parts
console.log(book.author); // Outputs: Douglas Crockford

book.author = "John Doe"; // This will not change the author
console.log(book.author); // Still outputs: Douglas Crockford
```

Using object properties in JavaScript allows you to define and manipulate the characteristics and behaviors of objects, providing more control over how data is managed within your applications.

JavaScript Object Accessors

Table of Contents

What are JavaScript Object Accessors?

JavaScript Object Accessors are special methods that allow you to access and update object properties indirectly. Accessors are defined using get and set keywords.
A **getter** is a method that retrieves the value of a property, while a **setter** is a method that sets or updates the value of a property.

Key Points:

- **Getter**: Used to access (get) the value of an object’s property.
- **Setter**: Used to change (set) the value of an object’s property.

JavaScript Object Accessor Table

Accessor Type	Description	Example Code	Output/Result
Getter	Retrieves the value of a property.	javascript let obj = { get fullName() { return "John Doe"; } }; console.log(obj.fullName);	"John Doe"
Setter	Updates the value of a property.	javascript let obj = { set fullName(name) { [this.first, this.last] = name.split(' '); } }; obj.fullName = "Jane Doe";	Updates obj.first to "Jane "

Examples of Using Object Accessors

Example 1: Basic Getter

```
let person = {
  firstName: "John",
  lastName: "Doe",
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

console.log(person.fullName); // Outputs: John Doe
```

In this example, the fullName getter concatenates firstName and lastName properties and returns the full name of the person.

Example 2: Basic Setter

```
let person = {
  firstName: "John",
  lastName: "Doe",
  set fullName(name) {
    let parts = name.split(" ");
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
};

person.fullName = "Jane Smith";
console.log(person.firstName); // Outputs: Jane
console.log(person.lastName); // Outputs: Smith
```

Here, the fullName setter splits the given name into firstName and lastName and updates the respective properties.

Example 3: Combining Getter and Setter

```
let user = {
  firstName: "Alice",
  lastName: "Johnson",
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  set fullName(name) {
    let parts = name.split(" ");
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
};
```

```
console.log(user.fullName); // Outputs: Alice Johnson
user.fullName = "Bob Brown";
console.log(user.firstName); // Outputs: Bob
console.log(user.lastName); // Outputs: Brown
```

In this example, both a getter and setter for fullName are defined. The getter returns the full name, and the setter updates the firstName and lastName properties based on the provided input.

Example 4: Defining Accessors Using Object.defineProperty()

```
let car = {
  brand: "Toyota",
  model: "Camry",
};

Object.defineProperty(car, "description", {
  get: function () {
    return `${this.brand} ${this.model}`;
  },
  set: function (value) {
    let parts = value.split(" ");
    this.brand = parts[0];
    this.model = parts[1];
  },
});

console.log(car.description); // Outputs: Toyota Camry
car.description = "Honda Accord";
console.log(car.brand); // Outputs: Honda
console.log(car.model); // Outputs: Accord
```

Using Object.defineProperty(), you can define both getters and setters for object properties, providing more control over how properties are accessed and updated.

Summary

JavaScript accessors offer a powerful way to control how properties of an object are accessed and updated, making it easier to maintain and manipulate data in a structured manner.

JavaScript Object Protection

Table of Contents

What is Object Protection in JavaScript?

Object Protection in JavaScript refers to techniques and methods used to control and restrict the modification of objects. By protecting objects, developers can ensure data integrity, prevent unintended mutations, and enforce encapsulation within their code.

JavaScript provides several built-in methods and patterns to achieve different levels and types of object protection, such as:

- Preventing addition of new properties.
- Preventing deletion or modification of existing properties.
- Creating truly private properties that cannot be accessed or modified from outside the object.

Understanding and utilizing these object protection mechanisms is essential for writing robust, secure, and maintainable JavaScript code.

Methods for Protecting Objects

JavaScript offers various methods and techniques to protect objects. Each method provides different levels of restriction and serves different purposes. Below are the most commonly used methods:

2.1. Object.freeze()

Description:

- The Object.freeze() method freezes an object, making it **immutable**.
- Once an object is frozen:
    - **No new properties** can be added.
    - **Existing properties** cannot be **removed or modified**.
    - The **prototype** cannot be changed.
- All properties become **non-configurable and non-writable**.

Syntax:

```
Object.freeze(object);
```

Example:

```
const obj = { name: "Alice" };
Object.freeze(obj);

obj.age = 30; // Fails silently in non-strict mode
obj.name = "Bob"; // Fails silently in non-strict mode
delete obj.name; // Fails silently in non-strict mode

console.log(obj); // Outputs: { name: "Alice" }
```

Use Cases:

- Ensuring that an object remains constant throughout the program.
- Preventing accidental mutations in objects that should not change.

2.2. Object.seal()

Description:

- The Object.seal() method seals an object.
- Once an object is sealed:
    - **No new properties** can be added.
    - **Existing properties** cannot be **deleted**.
    - **Property values** can still be **modified** if they are writable.
- All properties become **non-configurable**, but existing writable properties remain **writable**.

Syntax:

```
Object.seal(object);
```

Example:

```
const obj = { name: "Alice" };
Object.seal(obj);

obj.age = 30; // Fails silently in non-strict mode
obj.name = "Bob"; // Successfully updates the value
delete obj.name; // Fails silently in non-strict mode

console.log(obj); // Outputs: { name: "Bob" }
```

Use Cases:

- Preventing addition or removal of properties while allowing updates to existing properties.
- Maintaining the structure of an object while permitting value changes.

2.3. Object.preventExtensions()

Description:

- The Object.preventExtensions() method prevents an object from having new properties added to it.
- Once an object is made non-extensible:
    - **No new properties** can be added.
    - **Existing properties** can be **deleted**.
    - **Existing properties** can be **modified**.
- The object’s **prototype** cannot be changed.

Syntax:

```
Object.preventExtensions(object);
```

Example:

```
const obj = { name: "Alice" };
Object.preventExtensions(obj);

obj.age = 30; // Fails silently in non-strict mode
obj.name = "Bob"; // Successfully updates the value
delete obj.name; // Successfully deletes the property

console.log(obj); // Outputs: {}
```

Use Cases:

- Locking down the structure of an object to prevent accidental addition of new properties.
- Allowing flexibility to modify or remove existing properties as needed.

2.4. Defining Non-Writable and Non-Configurable Properties

Description:

- Using Object.defineProperty() or Object.defineProperties(), you can define properties with specific attributes:
    - **writable**: Determines if the property value can be changed.
    - **configurable**: Determines if the property can be deleted or its attributes can be modified.

    - **enumerable**: Determines if the property shows up during enumeration (e.g., in for...in loops).
- By setting properties as **non-writable** or **non-configurable**, you can control how and whether properties can be altered.

Syntax:

```
Object.defineProperty(object, propertyName, descriptor);
```

Example:

```
const obj = {};

Object.defineProperty(obj, "name", {
  value: "Alice",
  writable: false,
  configurable: false,
  enumerable: true,
});

obj.name = "Bob"; // Fails silently in non-strict mode
delete obj.name; // Fails silently

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

Use Cases:

- Creating constants within objects.
- Protecting critical properties from being changed or deleted.

2.5. Using Closures for Private Data

Description:

- Closures can be used to create **truly private variables**.
- Variables defined within a function scope are not accessible from outside.
- Public methods (privileged methods) can access these private variables through closure.

Syntax:

```
function createObject() {
  let privateVar = "secret";

  return {
    getPrivateVar: function () {
      return privateVar;
    },
    setPrivateVar: function (value) {
      privateVar = value;
    },
  };
}
```

Example:

```
const obj = (function () {
  let privateCounter = 0;

  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function () {
      changeBy(1);
    },
    decrement: function () {
      changeBy(-1);
    },
    value: function () {
      return privateCounter;
    },
  };
})();

console.log(obj.value()); // Outputs: 0
obj.increment();
obj.increment();
console.log(obj.value()); // Outputs: 2
console.log(obj.privateCounter); // Outputs: undefined
```

Use Cases:

- Encapsulating data and providing controlled access.
- Implementing module patterns and maintaining clean namespaces.

2.6. Using Symbols for Private Properties

Description:

- **Symbols** are a new primitive type introduced in ES6, and they are unique and immutable.
- Using symbols as property keys makes them **non-enumerable** and **less accessible**.
- Although not truly private, symbol-keyed properties are harder to access unintentionally.

Syntax:

```
const privateProp = Symbol("privateProp");
const obj = {
  [privateProp]: "secret",
};
```

Example:

```
const password = Symbol("password");

const user = {
  username: "alice",
  [password]: "12345",
};

console.log(user.username); // Outputs: "alice"
console.log(user.password); // Outputs: undefined
console.log(user[password]); // Outputs: "12345"

// Symbols are not listed in for...in loops
for (let key in user) {
  console.log(key); // Outputs: "username"
}

console.log(Object.keys(user)); // Outputs: ["username"]
```

Use Cases:

- Hiding properties from enumeration and casual access.
- Preventing property name clashes in large codebases or libraries.

2.7. Using WeakMaps for Private Data

Description:

- **WeakMaps** allow associating private data with objects.
- Keys in WeakMaps must be objects, and the associated values can be accessed only through these keys.
- Provides true privacy as the WeakMap is not accessible outside the scope.

Syntax:

```
const privateData = new WeakMap();

function MyClass() {
  privateData.set(this, { secret: "hidden" });
}
```

```
MyClass.prototype.getSecret = function () {
  return privateData.get(this).secret;
};
```

Example:

```
const privateData = new WeakMap();

class Person {
  constructor(name) {
    this.name = name;
    privateData.set(this, { age: 30 });
  }

  getAge() {
    return privateData.get(this).age;
  }

  setAge(age) {
    privateData.get(this).age = age;
  }
}

const person = new Person("Alice");
console.log(person.name); // Outputs: "Alice"
console.log(person.getAge()); // Outputs: 30
person.setAge(31);
console.log(person.getAge()); // Outputs: 31
console.log(person.age); // Outputs: undefined
```

Use Cases:

- Storing private data associated with object instances.

- Preventing external access and modification to sensitive data.

Comparison of Object Protection Methods

The table below summarizes the differences between various object protection methods:

Feature	Object.freeze()	Object.seal()	Object.preventExtensions()	Non-Writable Properties
Add new properties	❌	❌	❌	✅
Delete existing properties	❌	❌	✅	✅
Modify existing property values	❌	✅	✅	❌
Modify property descriptors	❌	❌	✅	❌
Change prototype	❌	❌	❌	✅
Use Cases	Immutable objects	Fixed structure with mutable values	Fixed structure	Immutable properties

Legend:

- ✅ = Allowed
- ❌ = Not Allowed

Notes:

- **Object.freeze()** provides the highest level of immutability by making all properties non-writable and non-configurable.
- **Object.seal()** allows modification of existing properties but prevents adding or deleting properties.
- **Object.preventExtensions()** only prevents adding new properties; existing properties can be modified or deleted.
- **Non-Writable Properties** can be achieved using Object.defineProperty() by setting writable: false.

Examples of Using Object Protection

4.1. Example with Object.freeze()

```
const config = {
  apiEndpoint: "https://api.example.com",
  timeout: 5000,
};

Object.freeze(config);

config.timeout = 10000; // Fails silently
config.newProp = true; // Fails silently
delete config.apiEndpoint; // Fails silently

console.log(config);
// Outputs: { apiEndpoint: "https://api.example.com", timeout: 5000 }
```

Explanation:

- After freezing, attempts to modify, add, or delete properties fail silently (or throw errors in strict mode).
- Useful for configuration objects that should remain constant.

4.2. Example with Object.seal()

```
const user = {
  name: "Bob",
  role: "Admin",
};

Object.seal(user);

user.role = "User"; // Successful
user.age = 30; // Fails silently
delete user.name; // Fails silently

console.log(user);
// Outputs: { name: "Bob", role: "User" }
```

Explanation:

- Modification of existing properties is allowed.
- Addition or deletion of properties is not allowed.
- Suitable when the object's structure should remain fixed but values may change.

4.3. Example with Object.preventExtensions()

```
const settings = {
  theme: "dark",
  notifications: true,
};

Object.preventExtensions(settings);

settings.theme = "light"; // Successful
delete settings.notifications; // Successful
```

```
settings.language = "en"; // Fails silently

console.log(settings);
// Outputs: { theme: "light" }
```

Explanation:

- Allows modification and deletion of existing properties.
- Prevents addition of new properties.
- Ideal when you want to prevent expansion of the object but allow modifications.

4.4. Example with Closures for Private Data

```
function createBankAccount(initialBalance) {
  let balance = initialBalance;

  return {
    deposit(amount) {
      if (amount > 0) balance += amount;
    },
    withdraw(amount) {
      if (amount > 0 && amount <= balance) balance -= amount;
    },
    getBalance() {
      return balance;
    },
  };
}

const account = createBankAccount(1000);
account.deposit(500);
account.withdraw(200);
console.log(account.getBalance()); // Outputs: 1300
console.log(account.balance); // Outputs: undefined
```

Explanation:

- The balance variable is private and cannot be accessed directly.
- Access is controlled through the returned methods.
- Ensures encapsulation and protects sensitive data.

4.5. Example with Symbols for Private Properties

```
const _salary = Symbol("salary");

class Employee {
  constructor(name, salary) {
    this.name = name;
    this[_salary] = salary;
  }

  getSalary() {
    return this[_salary];
  }
}

const emp = new Employee("Alice", 50000);
console.log(emp.name); // Outputs: "Alice"
console.log(emp.getSalary()); // Outputs: 50000
console.log(emp.salary); // Outputs: undefined
console.log(Object.keys(emp)); // Outputs: ["name"]
```

Explanation:

- The _salary property is not accessible through normal means.
- It does not appear during enumeration.
- Provides a level of privacy suitable for many applications.

4.6. Example with WeakMaps for Private Data

```
const privateProps = new WeakMap();

class Car {
  constructor(brand, model) {
    this.brand = brand;
    this.model = model;
    privateProps.set(this, { mileage: 0 });
  }

  drive(distance) {
    const props = privateProps.get(this);
    props.mileage += distance;
  }

  getMileage() {
    return privateProps.get(this).mileage;
  }
}

const myCar = new Car("Toyota", "Corolla");
myCar.drive(100);
myCar.drive(200);
console.log(myCar.getMileage()); // Outputs: 300
console.log(myCar.mileage); // Outputs: undefined
```

Explanation:

- The mileage property is truly private and cannot be accessed externally.
- WeakMap keys are garbage collected when the object is no longer in use.
- Ensures strong encapsulation and memory efficiency.

Summary

JavaScript provides multiple methods and patterns to protect and control access to object properties. Choosing the appropriate method depends on the specific requirements of your application:

- Use **Object.freeze()** when you need complete immutability.
- Use **Object.seal()** to prevent structural changes while allowing value updates.
- Use **Object.preventExtensions()** to stop addition of new properties but allow modification and deletion.
- Define **non-writable and non-configurable properties** for fine-grained control over individual properties.
- Utilize **closures**, **symbols**, and **WeakMaps** to implement various levels of property privacy and encapsulation.

Understanding these techniques enhances code reliability, security, and maintainability by preventing unintended side effects and enforcing proper access controls.

↥ Go to Top

Chapter-14: JavaScript Functions in Detail

JavaScript Function Definitions

Table of Contents

What is a Function in JavaScript?

- Function একটি নির্দিষ্ট কাজ সম্পন্ন করে। যখন Programming এ আমাদের একই কাজ বার বার দরকার হয়, তখন আমরা Function তৈরি করি এবং প্রয়োজন অনুসারে বার বার ব্যবহার করি।
- প্রতিটা Function ইনপুট নেয় (যাদেরকে Parameters বলা হয়) এবং একটি Output Return করে। এটিই Function এর General Concept. তবে ক্ষেত্র বিশেষে Input/Output নাও থাকতে পারে।

Types of Function Definitions

- JavaScript এ বিভিন্ন ধরনের Function আছে। যেমনঃ

2.1. Function Declaration

Description:

- A **function declaration** defines a named function using the function keyword.

- Function declarations are hoisted, meaning they can be called before they are defined in the code.

Syntax:

```
function functionName(parameters) {
  // Function body
}
```

Example:

```
function greet(name) {
  return `Hello, ${name}!`;
}

console.log(greet("Alice")); // Outputs: "Hello, Alice!"
```

Use Cases:

- When you need a function that is available throughout your code, even before the function is defined.

2.2. Function Expression

Description:

- A **function expression** defines a function as part of a variable assignment.
- Unlike function declarations, function expressions are not hoisted.

Syntax:

```
const functionName = function (parameters) {
  // Function body
};
```

Example:

```
const greet = function (name) {
  return `Hello, ${name}!`;
};

console.log(greet("Bob")); // Outputs: "Hello, Bob!"
```

Use Cases:

- When you want to define a function that is not available before its definition in the code.
- Useful for inline functions or assigning functions to variables, objects, or arrays.

2.3. Arrow Function

Description:

- An **arrow function** is a shorthand syntax for writing function expressions, introduced in ES6.
- Arrow functions do not have their own this, arguments, super, or new.target bindings.
- They are often used in situations where you need concise syntax, such as in callbacks or functional programming.

Syntax:

```
const functionName = (parameters) => {
  // Function body
};
```

Example:

```
const greet = (name) => `Hello, ${name}!`;


console.log(greet("Charlie")); // Outputs: "Hello, Charlie!"
```

Use Cases:

- When you need a shorter syntax for writing functions, particularly in callbacks, or when you want to maintain the this context from the surrounding scope.

2.4. Anonymous Function

Description:

- An **anonymous function** is a function that is defined without a name.
- These functions are often used as arguments to other functions or as immediately invoked function expressions (IIFE).

Syntax:

```
const functionName = function (parameters) {
  // Function body
};
```

Example:

```
setTimeout(function () {
  console.log("This is an anonymous function.");
}, 1000);
```

Use Cases:

- When you need a function temporarily, such as for event handlers, callbacks, or immediately invoked function expressions.

2.5. Immediately Invoked Function Expression (IIFE)

Description:

- An **immediately invoked function expression (IIFE)** is a function that is defined and executed immediately after it is created.
- IIFEs are used to create a private scope, avoiding the pollution of the global scope.

Syntax:

```
(function () {
  // Function body
})();
```

Example:

```
(function () {
  console.log("This IIFE runs immediately after its creation.");
})();
```

Use Cases:

- When you need to execute a function immediately while keeping the variables inside it private and isolated from the global scope.

2.6. Constructor Function

Description:

- A **constructor function** is a special type of function used to create and initialize objects in JavaScript.
- When a function is used as a constructor, it is called with the new keyword, creating a new object instance with the properties and methods defined in the constructor.

Syntax:

```
function ConstructorFunction(parameters) {
  // Initialize properties
  this.propertyName = value;
}
```

Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

const alice = new Person("Alice", 25);
console.log(alice.name); // Outputs: "Alice"
console.log(alice.age); // Outputs: 25
```

Use Cases:

- When you need to create multiple objects with the same properties and methods, using a function as a blueprint.

Parameters and Arguments

Parameters are the names listed in the function definition, while **arguments** are the actual values passed to the function when it is invoked.

- **Default Parameters:** Functions can have default values for parameters, allowing them to be optional.
- **Rest Parameters:** The rest parameter syntax (...) allows a function to accept an indefinite number of arguments as an array.

Example with Default and Rest Parameters:

```
function sum(a = 0, b = 0, ...rest) {
  return a + b + rest.reduce((acc, val) => acc + val, 0);
}

console.log(sum(1, 2)); // Outputs: 3
console.log(sum(1, 2, 3, 4)); // Outputs: 10
console.log(sum()); // Outputs: 0
```

Function Scope and Closures

Scope refers to the accessibility of variables within different parts of the code. Functions create their own scope, meaning variables declared inside a function are not accessible outside of it.

Closures are functions that "remember" the environment in which they were created, allowing them to access variables from their outer scope even after the outer function has finished executing.

Example of Closure:

```
function outer() {
  let counter = 0;

  return function () {
    counter++;
    return counter;
  };
}

const increment = outer();
console.log(increment()); // Outputs: 1
console.log(increment()); // Outputs: 2
```

Higher-Order Functions

A **higher-order function** is a function that takes one or more functions as arguments or returns a function as its result. Higher-order functions are a key concept in functional programming.

Example of Higher-Order Function:

```
function greet(name) {
  return function (message) {
    return `${message}, ${name}!`;
  };
}

const greetAlice = greet("Alice");
console.log(greetAlice("Good morning")); // Outputs: "Good morning, Alice!"
```

Use Cases:

- Higher-order functions are commonly used for function composition, callbacks, and other functional programming techniques.

Function Methods and Properties

In JavaScript, functions are objects, and as such, they have methods and properties. Some important methods include:

- **call()**: Calls a function with a given this value and arguments provided individually.
- **apply()**: Similar to call(), but arguments are provided as an array.
- **bind()**: Creates a new function that, when called, has its this value set to the provided value, with a given sequence of arguments.

Example with call(), apply(), and bind():

```
function introduce(greeting, punctuation) {
  console.log(`${greeting}, I'm ${this.name}${punctuation}`);
}

const person = { name: "Bob" };
```

```
introduce.call(person, "Hello", "!"); // Outputs: "Hello, I'm Bob!"
introduce.apply(
  person,

  ["Hi", "."]
); // Outputs: "Hi, I'm Bob."
const boundFunction = introduce.bind(person);
boundFunction("Hey", "..."); // Outputs: "Hey, I'm Bob..."
```

Examples of Function Definitions

7.1. Example with Function Declaration

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

console.log(add(3, 4)); // Outputs: 7
```

7.2. Example with Function Expression

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

console.log(subtract(9, 5)); // Outputs: 4
```

7.3. Example with Arrow Function

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

console.log(multiply(4, 6)); // Outputs: 24
```

7.4. Example with Anonymous Function

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

```
console.log(divide(10, 2)); // Outputs: 5
```

7.5. Example with IIFE

```
(function () {
  console.log("This IIFE runs immediately.");
})();
```

7.6. Example with Constructor Function

```
function Car(make, model) {
  this.make = make;
  this.model = model;
}

const myCar = new Car("Toyota", "Corolla");
console.log(myCar.make); // Outputs: "Toyota"
console.log(myCar.model); // Outputs: "Corolla"
```

Summary

JavaScript offers various ways to define functions, each with its strengths and best use cases. Understanding function declarations, expressions, arrow functions, anonymous functions, IIFEs, and constructor functions is essential for writing efficient and maintainable code. By mastering these function types, you can better structure your code, improve readability, and leverage the full power of JavaScript.

JavaScript Function Parameters

Table of Contents

What are Function Parameters?

Function parameters are the variables listed as part of the function definition. They are placeholders for the values (called arguments) that will be passed to the function when it is called. Parameters allow functions to be more flexible and reusable by accepting different inputs.

Syntax:

```
function functionName(parameter1, parameter2, ...) {
   // Function body
}
```

Types of Function Parameters

2.1. Default Parameters

Description:

- Default parameters allow you to initialize function parameters with default values if no arguments or undefined are passed.

Syntax:

```
function functionName(param1 = defaultValue1, param2 = defaultValue2) {
  // Function body
}
```

Example:

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}
```

```
console.log(greet("Alice")); // Outputs: "Hello, Alice!"
console.log(greet()); // Outputs: "Hello, Guest!"
```

Use Cases:

- When you want to ensure that a parameter always has a value, even if the caller doesn't provide one.

2.2. Rest Parameters

Description:

- Rest parameters allow a function to accept an indefinite number of arguments as an array. The rest parameter syntax uses the ... notation.

Syntax:

```
function functionName(...rest) {
  // Function body
}
```

Example:

```
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3)); // Outputs: 6
console.log(sum(4, 5)); // Outputs: 9
```

Use Cases:

- When you need a function that can handle an unknown number of arguments, such as in mathematical operations or variadic functions.

2.3. Named Parameters

Description:

- Named parameters are an object-oriented way to pass parameters, allowing the function to receive an object where the properties match the parameter names.

Syntax:

```
function functionName({ param1, param2 }) {
  // Function body
}
```

Example:

```
function createUser({ name, age, email }) {
  return `Name: ${name}, Age: ${age}, Email: ${email}`;
}

const user = { name: "Bob", age: 30, email: "bob@example.com" };
console.log(createUser(user)); // Outputs: "Name: Bob, Age: 30, Email: bob@example.com"
```

Use Cases:

- When you want to improve readability and avoid confusion with the order of parameters, especially when there are many optional parameters.

Passing Parameters by Value vs. Reference

JavaScript function parameters can be passed by value or by reference, depending on the type of the argument.

- **Primitive Types (e.g., numbers, strings, booleans):** Passed by value, meaning the function works with a copy of the original value.
- **Reference Types (e.g., objects, arrays):** Passed by reference, meaning the function works with a reference to the original object.

Example of Passing by Value:

```
function changeValue(value) {
  value = 10;
}

let num = 5;
changeValue(num);
console.log(num); // Outputs: 5
```

Example of Passing by Reference:

```
function changeObject(obj) {
  obj.name = "Alice";
}

let person = { name: "Bob" };
changeObject(person);
console.log(person.name); // Outputs: "Alice"
```

Parameter Destructuring

Description:

- Parameter destructuring is a feature that allows you to extract values from objects or arrays directly within the function parameters, making the code more concise and readable.

Syntax:

```
function functionName({ param1, param2 }) {
  // Function body
}
```

Example with Object Destructuring:

```
function displayUser({ name, age }) {
  return `Name: ${name}, Age: ${age}`;
}

const user = { name: "Alice", age: 25 };
console.log(displayUser(user)); // Outputs: "Name: Alice, Age: 25"
```

Example with Array Destructuring:

```
function sum([a, b, c]) {
  return a + b + c;
}

console.log(sum([1, 2, 3])); // Outputs: 6
```

Use Cases:

- When you want to work directly with specific properties or elements from an object or array passed as a parameter, reducing the need for additional variable assignments.

Examples of Function Parameters

5.1. Example with Default Parameters

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}

console.log(greet("Alice")); // Outputs: "Hello, Alice!"
console.log(greet()); // Outputs: "Hello, Guest!"
```

5.2. Example with Rest Parameters

```
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3)); // Outputs: 6
console.log(sum(4, 5)); // Outputs: 9
```

5.3. Example with Named Parameters

```
function createUser({ name, age, email }) {
  return `Name: ${name}, Age: ${age}, Email: ${email}`;
}

const user = { name: "Bob", age: 30, email: "bob@example.com" };
console.log(createUser(user)); // Outputs: "Name: Bob, Age: 30, Email: bob@example.com"
```

5.4. Example with Destructuring Parameters

```
function displayUser({ name, age }) {
  return `Name: ${name}, Age: ${age}`;
}

const user = { name: "Alice", age: 25 };
console.log(displayUser(user)); // Outputs: "Name: Alice, Age: 25"
```

Summary

Understanding how to use function parameters effectively allows you to write more flexible and reusable functions in JavaScript. By mastering default parameters, rest parameters, named parameters, and destructuring, you can enhance the readability and maintainability of your code.

JavaScript Function Invocation

Table of Contents

What is Function Invocation?

Function invocation is the process of calling or executing a function in JavaScript. When a function is invoked, the code inside the function body is executed. The way a function is invoked determines the context in which the function is executed and affects the value of the this keyword.

Syntax:

```
functionName(arguments);
```

Types of Function Invocation

2.1. Function Invocation as a Function

Description:

- This is the simplest form of invocation. The function is invoked directly using its name followed by parentheses.

Syntax:

```
functionName();
```

Example:

```
function greet() {
  console.log("Hello, world!");
}

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

Use Cases:

- Standard way of executing a function.

2.2. Function Invocation as a Method

Description:

- When a function is invoked as a method of an object, it is executed in the context of that object. The this keyword refers to the object that owns the method.

Syntax:

```
object.methodName();
```

Example:

```
const person = {
  name: "Alice",
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};

person.greet(); // Outputs: "Hello, my name is Alice"
```

Use Cases:

- Useful when working with object-oriented programming in JavaScript.

2.3. Function Invocation with call and apply

Description:

- The call and apply methods allow a function to be invoked with a specified this value and arguments.

Syntax:

```
functionName.call(thisArg, arg1, arg2, ...);
functionName.apply(thisArg, [arg1, arg2, ...]);
```

Example:

```
function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}

const person = { name: "Bob" };

introduce.call(person, "Hi"); // Outputs: "Hi, I'm Bob"
introduce.apply(person, ["Hey"]); // Outputs: "Hey, I'm Bob"
```

Use Cases:

- When you need to explicitly set the this value for a function, such as when borrowing methods from another object.

2.4. Constructor Invocation

Description:

- When a function is invoked with the new keyword, it acts as a constructor, creating a new object and setting the this keyword to that new object.

Syntax:

```
new FunctionName(arguments);
```

Example:

```
function Car(make, model) {
  this.make = make;
  this.model = model;
}

const myCar = new Car("Toyota", "Corolla");
console.log(myCar.make); // Outputs: "Toyota"
console.log(myCar.model); // Outputs: "Corolla"
```

Use Cases:

- Used for creating instances of objects using constructor functions.

2.5. Indirect Invocation with bind

Description:

- The bind method creates a new function that, when invoked, has its this keyword set to the provided value.

Syntax:

```
const boundFunction = functionName.bind(thisArg);
```

Example:

```
function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}

const person = { name: "Bob" };
const boundIntroduce = introduce.bind(person);
boundIntroduce("Hello"); // Outputs: "Hello, I'm Bob"
```

Use Cases:

- When you want to create a function with a fixed this context, useful in event handling or callback functions.

The this Keyword in Different Invocations

The value of this depends on how the function is invoked:

- **Function Invocation as a Function:** this refers to the global object (in non-strict mode) or undefined (in strict mode).
- **Function Invocation as a Method:** this refers to the object the method belongs to.
- **Function Invocation with call and apply:** this is explicitly set to the specified object.
- **Constructor Invocation:** this refers to the newly created object.
- **Indirect Invocation with bind:** this is set to the value provided to bind.

Examples of Function Invocation

4.1. Example with Function Invocation as a Function

```
function greet() {
  console.log("Hello, world!");
}


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

4.2. Example with Function Invocation as a Method

```
const person = {
  name: "Alice",
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};
```

```
person.greet(); // Outputs: "Hello, my name is Alice"
```

4.3. Example with call and apply Invocation

```
function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}

const person = { name: "Bob" };

introduce.call(person, "Hi"); // Outputs: "Hi, I'm Bob"
introduce.apply(person, ["Hey"]); // Outputs: "Hey, I'm Bob"
```

4.4. Example with Constructor Invocation

```
function Car(make, model) {
  this.make = make;
  this.model = model;
}

const myCar = new Car("Toyota", "Corolla");
console.log(myCar.make); // Outputs: "Toyota"
console.log(myCar.model); // Outputs: "Corolla"
```

4.5. Example with bind Invocation

```
function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}

const person = { name: "Bob" };

const boundIntroduce = introduce.bind(person);
boundIntroduce("Hello"); // Outputs: "Hello, I'm Bob"
```

Summary

Function invocation is a fundamental concept in JavaScript that determines how and in what context a function is executed. Understanding the different types of invocation—whether as a regular function, method, constructor, or through call, apply, or bind—is crucial for controlling the execution context (this) and writing robust, reusable code.

JavaScript call() Method

Table of Contents

Introduction to call()

- Scenario যদি এরকম হয় যে ধরুন, আমারা একটা A Object এর একটা Method কে Call করতে চাচ্ছি এবং ঐ A Object এর ঐ Method এ **this** আছে। এখন এই **this** অবশ্যই কোন একটা Object কে Refer করবে। এই **this** কোন Object কে নির্দেশ করবে সেটা যদি আমারা বাইরে থেকে বলে দিতে চাই, তাহলে **call()** Method এর ১ম Argument এ সেই Object এর নাম বলে দিতে পারি।

Syntax of call()

Syntax:

function.call(objectName, arg1, arg2, ...);

- **objectName**: The value to be passed as the this context. If null or undefined, the global object will be used.
- **arg1, arg2, ...**: Arguments to be passed to the function.

Examples of Using call()

4.1. Example with Method Borrowing

Example:

```
const person = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
};

const person1 = {
  firstName: "Abdul",
  lastName: "Alim",
};
const person2 = {
  firstName: "Abdur",
  lastName: "Rahim",
};

console.log(person.fullName.call(person1)); // Outputs: "Abdul Alim"
console.log(person.fullName.call(person2)); // Outputs: "Abdur Rahim"
```

4.2. Example with Inheriting Methods from Other Objects

Example:

```
const person = {
  firstName: "Abdul",
  lastName: "Alim",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
};
```

```
const anotherPerson = {
  firstName: "Abdur",
  lastName: "Rahim",
};

console.log(person.fullName.call(anotherPerson)); // Outputs: "Abdur Rahim"
```

4.3. Example with Function Invocation

Example:

```
function greet(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}

const person = { name: "Alice" };

greet.call(person, "Hello"); // Outputs: "Hello, I'm Alice"
```

Explanation: The greet function is invoked with call(), and the this keyword inside greet is set to person, allowing the function to access person's name property.

4.4. Example with call() for this Binding

Example:

```
const car = {
  brand: "Toyota",
  showBrand: function () {
    console.log(`This car is a ${this.brand}`);
  },
};

const bike = {
  brand: "Honda",
```

};

car.showBrand.call(bike); // Outputs: "This bike is a Honda"

Explanation: Here, the call() method is used to set the this value inside showBrand to bike, making it display "Honda" instead of "Toyota".

Advantages of call()

1. **Method Borrowing:** Easily borrow methods from other objects without duplicating code.
2. **Explicit this Binding:** Explicitly set the this value for any function, providing more control over function execution.
3. **Function Reusability:** Reuse functions across different objects, making your code DRY (Don't Repeat Yourself).

JavaScript apply() Method

Table of Contents

Introduction to apply()

The apply() method in JavaScript is similar to call(), but with a key difference: it allows you to call a function with a specified this value and arguments provided as an array (or an array-like object). This is particularly useful when you have a variable number of arguments or an array of arguments that you want to pass to a function.

Syntax of apply()

Syntax:

```
function.apply(thisArg, [argsArray]);
```

- **thisArg**: The value to be passed as the this context. If null or undefined, the global object will be used.
- **argsArray**: An array or array-like object containing the arguments to be passed to the function.

Examples of Using apply()

4.1. Example with Array as Arguments

Example:

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

const numbers = [1, 2, 3];
console.log(sum.apply(null, numbers)); // Outputs: 6
```

Explanation: In this example, the sum function is invoked using apply(), with the arguments provided as an array. The function adds the numbers in the array and returns the result.

4.2. Example with Method Borrowing

Example:

```
const person1 = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
};

const person2 = {
  firstName: "Jane",
  lastName: "Doe",
};

console.log(person1.fullName.apply(person2)); // Outputs: "Jane Doe"
```

Explanation: Here, person2 borrows the fullName method from person1 using apply(), allowing person2 to use the method as if it were its own.

4.3. Example with Math Functions

Example:

```
const numbers = [5, 6, 2, 3, 7];

const max = Math.max.apply(null, numbers);
const min = Math.min.apply(null, numbers);

console.log(max); // Outputs: 7
console.log(min); // Outputs: 2
```

Explanation: The apply() method is used here to pass an array of numbers to the Math.max and Math.min functions, which return the largest and smallest numbers in the array, respectively.

4.4. Example with apply() for this Binding

Example:

```
const car = {
  brand: "Tesla",
  showBrand: function (speed, time) {
    console.log(
      `This car is a ${this.brand} and it travels ${
        speed * time
      } miles in ${time} hours.`
    );
  },
};

const bike = {
  brand: "Yamaha",
};

car.showBrand.apply(bike, [60, 2]); // Outputs: "This car is a Yamaha and it travels 120 miles in 2 hours."
```

Explanation: In this example, the apply() method is used to invoke the showBrand method with bike as the this context, passing the speed and time as an array.

JavaScript bind() Method

Table of Contents

Introduction to bind()

The bind() method in JavaScript is used to create a new function that, when called, has its this keyword set to the provided value. It also allows you to pass in a sequence of arguments that will be prepended to any arguments provided when the new function is invoked. This is particularly useful for preserving the this context across different scopes and for partial function application.

Syntax of bind()

Syntax:

```
function.bind(thisArg, arg1, arg2, ...);
```

- **thisArg**: The value to be passed as the this context. If null or undefined, the global object will be used.
- **arg1, arg2, ...**: Arguments to be passed to the function.

How bind() Works

The bind() method returns a new function with the this value set to thisArg and any arguments provided in the initial bind() call. When the new function is invoked, it will use the bound this value and arguments, along with any arguments passed during the invocation.

Examples of Using bind()

4.1. Example with Preserving this Context

Example:

```
const person = {
  name: "Alice",
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};


const greet = person.greet.bind(person);
greet(); // Outputs: "Hello, my name is Alice"
```

Explanation: In this example, the greet function is bound to the person object, preserving the this context so that when greet is called later, it correctly references person.

4.2. Example with Partial Application

Example:

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

const double = multiply.bind(null, 2);
console.log(double(5)); // Outputs: 10
```

Explanation: Here, the multiply function is partially applied using bind(), creating a new function double that always multiplies its argument by 2.

4.3. Example with Event Handlers

Example:

```
function Button() {
  this.clicked = false;
  this.click = function () {
    this.clicked = true;
    console.log("Button clicked:", this.clicked);
  };
}

const button = new Button();
const boundClick = button.click.bind(button);
document.getElementById("myButton").addEventListener("click", boundClick);
```

Explanation: In this example, bind() is used to ensure that the this context inside the click method of the Button class remains correctly set, even when the method is used as an event handler.

4.4. Example with Method Borrowing

Example:

```
const person1 = {
  name: "John",
  getName: function () {
    return this.name;
  },
};

const person2 = {
  name: "Doe",
};

const getName = person1.getName.bind(person2);
console.log(getName()); // Outputs: "Doe"
```

Explanation: In this example, bind() is used to borrow the getName method from person1 and bind it to person2, so that when the method is invoked, it returns person2's name.

Advantages of bind()

1. **Preserve this Context:** Ensure that the this context is maintained across different scopes, especially in asynchronous operations or event handlers.
2. **Partial Application:** Create new functions with pre-filled arguments, making your code more modular and reusable.
3. **Function Borrowing:** Borrow methods from other objects and bind them to a different this context.

Summary

The bind() method is a powerful tool in JavaScript that allows you to control the this context and create partially applied functions. Whether you're dealing with event handlers, asynchronous operations, or method borrowing, bind() can help you write more maintainable and flexible code. Understanding how to use bind() effectively is essential for working with JavaScript functions.

JavaScript Closures

Table of Contents

Why Do You Need Closures?

Imagine you're creating a counter in JavaScript. You want the counter to start at zero and increase by one each time you call it. However, you also want to ensure that the count value is not accessible or modifiable from outside the counter function—essentially keeping it private.

Without closures, it can be challenging to maintain this private state across multiple calls to the counter function. Closures allow you to encapsulate the counter logic within a function, ensuring that the count variable remains private and is only accessible through the function itself.

Real Example:

```
function createCounter() {
  let count = 0; // Private variable

  return function () {
    count += 1;
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // Outputs: 1
console.log(counter()); // Outputs: 2
console.log(counter()); // Outputs: 3
```

In this example, the count variable is protected within the createCounter function. Each time the inner function is called, it accesses and modifies the count variable, demonstrating the power of closures to maintain a private state across function calls.

Introduction to Closures

A **closure** is a feature in JavaScript where an inner function has access to the outer (enclosing) function’s variables— a scope chain. Closures are created every time a function is created, at function creation time. Closures allow a function to access variables from an outer function even after the outer function has completed its execution.

How Closures Work

In JavaScript, when a function is declared inside another function, it forms a closure. This closure allows the inner function to access variables from its outer function even after the outer function has finished executing. This happens because the inner function maintains a reference to its outer function's scope.

Examples of Closures

4.1. Example with Nested Functions

Example:

```
function outerFunction() {
  let outerVariable = "I am outside!";

  function innerFunction() {
    console.log(outerVariable);
  }
  return innerFunction;
}

const myClosure = outerFunction();
myClosure(); // Outputs: "I am outside!"
```

Explanation: In this example, innerFunction forms a closure with outerFunction, allowing it to access outerVariable even after outerFunction has returned.

4.2. Example with Private Variables

Example:

```
function createCounter() {
  let count = 0;

  return function () {
    count += 1;
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // Outputs: 1
console.log(counter()); // Outputs: 2
console.log(counter()); // Outputs: 3
```

Explanation: Here, the count variable is private to the createCounter function, and it can only be accessed and modified through the inner function returned by createCounter. Each call to counter() increments and returns the updated count, demonstrating how closures can be used to create private variables.

4.3. Example with Callbacks

Example:

```
function fetchData(url) {
  const secretKey = "my-secret-key";

  return function () {
    console.log(`Fetching data from ${url} using key ${secretKey}`);
  };
}

const fetchDataFromAPI = fetchData("https://api.example.com");
fetchDataFromAPI(); // Outputs: "Fetching data from https://api.example.com using key my-secret-key"
```

Explanation: In this example, the fetchData function returns a closure that remembers the url and secretKey variables, allowing the inner function to access them later, even after fetchData has completed.

4.4. Example with Looping Constructs

Example:

```
function createTimers() {
  for (let i = 0; i < 5; i++) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }
}

createTimers();
// Outputs: 0, 1, 2, 3, 4 (each after 1 second interval)
```

Explanation: In this example, the loop creates a closure for each iteration of the loop. Since let is block-scoped, each closure maintains a reference to its specific i value, allowing it to print the correct number at the correct time.

Advantages of Closures

1. **Data Encapsulation:** Closures allow you to encapsulate data within a function, keeping it private and protected from the global scope.
2. **Persistent State:** Closures help maintain a persistent state across function calls, which is useful in scenarios like counters or settings.
3. **Modular Code:** By using closures, you can write more modular and reusable code, avoiding global variables and minimizing side effects.

Common Use Cases for Closures

1. **Event Handlers:** Closures are often used in event handlers to maintain access to variables from the outer scope.
2. **Callback Functions:** Closures are commonly used in asynchronous programming and callbacks to preserve the state between function calls.
3. **Module Patterns:** Closures form the basis for creating modules in JavaScript, where functions and variables are encapsulated within a single scope.

Summary

Closures are a powerful feature in JavaScript that allow functions to access variables from their outer scope, even after the outer function has completed execution. They provide a way to create private variables, maintain state, and write more modular code. Understanding closures is crucial for mastering JavaScript, as they are fundamental to many advanced programming patterns and techniques.

Higher-Order Functions

Table of Contents

Why Do You Need Higher-Order Functions?

Higher-order functions are a powerful concept in JavaScript that allow you to write more flexible, reusable, and expressive code. Imagine you need to apply a series of transformations to an array of numbers, such as doubling each number and then filtering out those greater than 10. Instead of writing repetitive loops, you can use higher-order functions like map() and filter() to achieve this in a clean and efficient way.

Real Example:

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


const doubled = numbers.map((num) => num * 2); // [2, 4, 6, 8, 10]

const filtered = doubled.filter((num) => num <= 10); // [2, 4, 6, 8, 10]


console.log(filtered); // Outputs: [2, 4, 6, 8, 10]
```

In this example, the map() and filter() functions are higher-order functions that help you transform and filter the array with minimal code. Without higher-order functions, achieving the same result would require more verbose and less readable code.

Introduction to Higher-Order Functions

A **higher-order function** is a function that can take another function as an argument, return a function as its result, or do both. This capability allows you to create more abstract and reusable code by encapsulating behavior that can be easily passed around and applied in different contexts.

How Higher-Order Functions Work

Higher-order functions work by accepting a function as an argument or returning a function as a result. This allows for a more functional programming style, where functions can be composed, reused, and passed around as first-class citizens in your code.

Examples of Higher-Order Functions

4.1. Array Methods as Higher-Order Functions

Example:

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


const squared = numbers.map((num) => num * num);

console.log(squared); // Outputs: [1, 4, 9, 16, 25]
```

Explanation: In this example, map() is a higher-order function that takes a function (num => num * num) as its argument and applies it to each element in the array, returning a new array with the results.

4.2. Creating a Custom Higher-Order Function

Example:

```
function repeat(operation, num) {
  return function () {
    for (let i = 0; i < num; i++) {
      operation();
    }
  };
}


const sayHello = repeat(() => console.log("Hello!"), 3);

sayHello();

// Outputs: "Hello!" three times
```

Explanation: Here, repeat is a higher-order function that takes an operation function and a num value as arguments. It returns a new function that, when called, repeats the operation a specified number of times.

4.3. Using Higher-Order Functions for Event Handling

Example:

```
function addEventListenerWithLog(element, event, handler) {
  element.addEventListener(event, function (event) {
    console.log(`Event triggered: ${event.type}`);
    handler(event);
  });
}

addEventListenerWithLog(
  document.getElementById("myButton"),
  "click",
  function () {
    console.log("Button clicked!");
  }
);
```

Explanation: In this example, addEventListenerWithLog is a higher-order function that wraps the original event handler with additional logging functionality. This allows you to add custom behavior to existing event handlers without modifying them directly.

4.4. Functional Composition with Higher-Order Functions

Example:

```
function compose(f, g) {
  return function (x) {
    return f(g(x));
  };
}

const addOne = (x) => x + 1;
const square = (x) => x * x;

const addOneThenSquare = compose(square, addOne);
```

```
console.log(addOneThenSquare(2)); // Outputs: 9
```

Explanation: Here, compose is a higher-order function that takes two functions, f and g, as arguments and returns a new function that applies g to its input and then applies f to the result. This demonstrates how higher-order functions can be used to create function compositions.

Advantages of Higher-Order Functions

1. **Code Reusability:** Higher-order functions promote code reuse by allowing you to create generic functions that can be customized with different behaviors.
2. **Functional Programming:** They enable a more functional programming style, where functions can be composed and reused in a declarative manner.
3. **Cleaner Code:** By abstracting repetitive behavior into higher-order functions, your code becomes more concise and easier to read.

Common Use Cases for Higher-Order Functions

1. **Array Operations:** Functions like map, filter, and reduce are commonly used higher-order functions for transforming and processing arrays.
2. **Event Handling:** Higher-order functions can be used to enhance or modify event handlers with additional functionality.
3. **Function Composition:** They are often used to compose multiple functions into a single function, enabling more modular and reusable code.

Summary

Higher-order functions are a fundamental concept in JavaScript that allow you to write more modular, reusable, and expressive code. They enable a functional programming style, where functions can be passed around, composed, and applied in various contexts. Understanding and using higher-order functions is key to mastering JavaScript and writing clean, efficient code.

Chapter-15: Asynchronous JavaScript

JavaScript Callbacks

Table of Contents

1. Introduction to Callbacks

What is a Callback?

Callback function হলো JavaScript এর একটি powerful concept। এটি হলো একটি function যা অন্য একটি function এর argument হিসেবে pass করা হয় এবং প্রয়োজনমতো সেই function এর মধ্যে call করা হয়।

Understanding with Simple Terms:

ধরুন আপনি আপনার বন্ধুকে একটি কাজ করার জন্য বললেন, এবং বললেন কাজটি শেষ হলে আপনাকে ফোন করতে। এখানেই callback এর মূল ধারণা। আপনি আপনার বন্ধুকে বললেন কাজটি শেষ হলে আপনাকে ফোন করতে (যেটা হলো callback function), এবং আপনার বন্ধু কাজ শেষে আপনাকে ফোন করবে (যেটা হলো function call)।

Example:

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

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("John", sayGoodbye);
```

Console Output:

```
Hello John
Goodbye!
```

Explanation: এখানে greet নামক একটি function আছে যা name এবং callback নামে দুইটি parameter নেয়। greet function প্রথমে console এ "Hello" এবং name print করবে এবং তারপর callback() function কে call করবে। এখানে, sayGoodbye function callback হিসেবে pass করা হয়েছে এবং পরে call করা হচ্ছে।

2. Why Use Callbacks?

Callbacks ব্যবহার করার প্রয়োজনীয়তা

JavaScript একটি asynchronous programming language। অর্থাৎ, এটি একাধিক কাজ একসঙ্গে handle করতে পারে। একাধিক কাজ handle করতে গেলে callback functions ব্যবহার করা হয় যাতে একটি কাজ শেষ হওয়ার পর অন্য কাজ শুরু হয়।

Example:

```
function downloadFile(url, callback) {
  console.log("Downloading file from " + url);
  setTimeout(function () {
    console.log("File downloaded");
    callback();
  }, 3000);
}

function processFile() {
  console.log("Processing the downloaded file");
}

downloadFile("http://example.com/file", processFile);
```

Console Output:

```
Downloading file from http://example.com/file
File downloaded
Processing the downloaded file
```

Explanation: এখানে downloadFile function url এবং callback নামে দুইটি parameter নেয়। এটি একটি file download করতে ৩ সেকেন্ড সময় নেয় (এটি simulate করার জন্য setTimeout function ব্যবহার করা হয়েছে)। file download শেষে callback() function call হয়, যেটি এখানে processFile function. এইভাবে আমরা কাজগুলিকে নির্দিষ্ট ক্রমে চালাতে পারি।

3. Creating a Callback Function

How to Create a Callback Function

একটি callback function তৈরি করা খুবই সহজ। এটি মূলত একটি function যা অন্য একটি function এর argument হিসেবে pass করা হয় এবং সেই function এর মধ্যে নির্দিষ্ট সময়ে call করা হয়।

Step-by-Step Example:

1. **Define a Main Function:** প্রথমে একটি main function তৈরি করতে হবে, যেটি callback function কে parameter হিসেবে নেবে।
2. **Define a Callback Function:** তারপর একটি callback function তৈরি করতে হবে।
3. **Pass the Callback to the Main Function:** main function call করার সময় callback function কে argument হিসেবে pass করতে হবে।

Example:

```
function performTask(task, callback) {
  console.log("Performing task: " + task);
  callback();
}


performTask("Cleaning", function () {
  console.log("Task completed!");
});
```

Console Output:

```
Performing task: Cleaning
Task completed!
```

Explanation: performTask function একটি কাজ করে (এখানে console এ একটি message print করা হচ্ছে) এবং তারপর callback function (function() { console.log('Task completed!'); }) call করে। এটি একটি simple উদাহরণ, কিন্তু বাস্তবে callback function অনেক বড় এবং জটিল হতে পারে।

4. Asynchronous JavaScript and Callbacks

Understanding Asynchronous Behavior

JavaScript একটি single-threaded language, অর্থাৎ এটি এক সময়ে একটি কাজ করতে পারে। কিন্তু, asynchronous কাজের জন্য এটি callbacks ব্যবহার করে যাতে একাধিক কাজ করতে পারে। এর অর্থ, JavaScript কোন কাজ শেষ হওয়ার জন্য অপেক্ষা না করে অন্য কাজ শুরু করতে পারে, এবং কাজ শেষ হলে callback function এর মাধ্যমে জানানো হয়।

Example with setTimeout:

```
function fetchData(callback) {
  setTimeout(function () {
    console.log("Data fetched from server");
    callback();
  }, 2000);
}

function processData() {
  console.log("Processing data...");
}

fetchData(processData);
```

Console Output:

```
Data fetched from server
Processing data...
```

Explanation: fetchData function asynchronous কাজ করে এবং ২ সেকেন্ড পর processData function কে call করে। এখানে setTimeout asynchronous behavior simulate করতে ব্যবহার করা হয়েছে।

5. Common Use Cases for Callbacks

5.1 Event Handling

ইভেন্ট handling এ callback functions ব্যবহার করা হয়। যখন user কোন button এ click করে বা কোন input field এ কিছু লেখে, তখন ইভেন্ট ঘটে। সেই ইভেন্টগুলো handle করতে callback functions ব্যবহার করা হয়।

```
document.getElementById("myButton").addEventListener("click", function () {
  console.log("Button clicked!");
});
```

Explanation: এখানে, যখন button এ click করা হয়, তখন callback function call হয় যা console এ একটি message print করে। **Console Output** এর জন্য বাস্তব সময়ে এই code টি browser এ run করতে হবে।

5.2 Server Requests

AJAX বা অন্য কোন server requests এ callbacks অত্যন্ত গুরুত্বপূর্ণ।

```
function makeRequest(url, callback) {
  console.log("Making request to " + url);
  // Hypothetical request simulation
  setTimeout(function () {
    console.log("Request completed!");
    callback();
  }, 2000);
}

makeRequest("http://example.com", function () {
  console.log("Data received and processed!");
});
```

Console Output:

```
Making request to http://example.com
Request completed!
Data received and processed!
```

Explanation: এই উদাহরণে, makeRequest function asynchronous request simulate করে এবং request শেষ হওয়ার পর callback function call করে।

5.3 Array Methods

JavaScript এর built-in array methods, যেমন: forEach, map, filter, এও callback functions ব্যবহার করা হয়।

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

Console Output:

```
2
4
6
8
10
```

Explanation: এখানে, forEach method এর মধ্যে একটি callback function pass করা হয়েছে যা array এর প্রতিটি element কে ২ দিয়ে গুন করে।

6. Real-life Example of Callbacks

Scenario 1: Photo Upload and Resizing

ধরুন, আপনি একটি photo editing application তৈরি করছেন যেখানে একজন user তার profile picture upload করবে, তারপর সেই picture টি automatically resize এবং database এ save করা হবে। এখানে, আমরা দুটি প্রধান কাজ করতে চাই:

1. **Image Upload:** প্রথমে image upload করতে হবে।
2. **Image Resize:** এরপর image টি resize করে database এ save করতে হবে।

এই কাজগুলো sequentially করতে হবে, অর্থাৎ upload হওয়ার পরই resize করতে হবে। এই ধরনের কাজ callback function এর মাধ্যমে সহজেই করা যায়।

```
function uploadImage(image, callback) {
  console.log("Uploading image...");
  setTimeout(function () {
    console.log("Image uploaded.");
    callback();
  }, 2000);
}

function resizeImage() {
  console.log("Resizing image...");
  setTimeout(function () {
    console.log("Image resized and saved to database.");
  }, 1000);
}

uploadImage("profile.jpg", resizeImage);
```

Console Output:

```
Uploading image...
Image uploaded.
Resizing image...
Image resized and saved to database.
```

Detailed Explanation:

1. **Upload Image:** uploadImage function টি image upload করার জন্য simulate করা হয়েছে। এখানে, setTimeout function ব্যবহার করে asynchronous কাজ করা হয়েছে যা ২ সেকেন্ড সময় নিয়ে image upload complete করেছে।
2. **Resize Image:** Image upload হওয়ার পর resizeImage function call হয়, যা image resize এবং database এ save করার কাজটি সম্পন্ন করে। এটি আরও ১ সেকেন্ড সময় নেয়।

এই কাজগুলোতে callback function এর ব্যবহার অত্যন্ত গুরুত্বপূর্ণ, কারণ আমরা চাই image টি upload হওয়ার পরেই resize করতে। Callback function এই ধরনের কাজগুলিকে নির্দিষ্ট ক্রমে (sequence) সম্পন্ন করতে সাহায্য করে।

Scenario 2: User Registration Process

ধরুন, আপনি একটি user registration system তৈরি করছেন যেখানে user তার নাম এবং email address দিয়ে register করবে। Registration প্রক্রিয়া সম্পন্ন করার পর user কে একটি welcome email পাঠানো হবে। এই ধরনের কাজ sequential ভাবে করতে হবে।

```
function registerUser(name, email, callback) {
  console.log("Registering user: " + name);
  setTimeout(function () {
    console.log("User registered successfully.");
    callback();
  }, 1500);
}

function sendWelcomeEmail() {
  console.log("Sending welcome email...");
  setTimeout(function () {
    console.log("Welcome email sent!");
  }, 1000);
}

registerUser("John Doe", "john@example.com", sendWelcomeEmail);
```

Console Output:

```
Registering user: John Doe
User registered successfully.
Sending welcome email...
Welcome email sent!
```

Detailed Explanation:

1. **Register User:** প্রথমে registerUser function call করে user registration সম্পন্ন করা হচ্ছে। এটি ১.৫ সেকেন্ড সময় নিচ্ছে।
2. **Send Welcome Email:** Registration সম্পন্ন হলে sendWelcomeEmail function call করা হচ্ছে যা user কে একটি welcome email পাঠাচ্ছে।

এখানে callback function ব্যবহার করা হয়েছে যাতে registration সম্পন্ন হওয়ার পর welcome email পাঠানো হয়।

7. Handling Errors in Callbacks

Error Handling in Callbacks

Callback functions error handling এর জন্যও ব্যবহার করা হয়। error handle করার জন্য সাধারণত প্রথম argument হিসেবে error message pass করা হয়।

Example:

```
function doTask(task, callback) {
  if (!task) {
    callback("No task provided", null); // Passing error message
  } else {
    console.log("Task is being done");
    callback(null, "Task completed"); // No error, passing result
  }
}

doTask(null, function (error, result) {
  if (error) {
    console.log("Error: " + error);
  } else {
    console.log(result);
  }
});
```

Console Output:

```
Error: No task provided
```

Explanation: এই উদাহরণে, যদি কোনো task provide না করা হয়, তাহলে callback function এ error message pass করা হয় এবং error handle করা হয়।

8. Conclusion

JavaScript callbacks powerful এবং flexible functions যা asynchronous এবং synchronous উভয় কাজেই ব্যবহৃত হয়। callbacks ব্যবহার করার ফলে JavaScript non-blocking, event-driven programming করতে পারে। Proper callback usage করার মাধ্যমে complex asynchronous tasks সহজে manage করা যায় এবং error handling সহজ হয়।

Asynchronous JavaScript

Table of Contents

1. Introduction to Asynchronous JavaScript

What is Asynchronous JavaScript?

Asynchronous JavaScript হলো এমন একটি programming paradigm যেখানে code execution একসঙ্গে multiple tasks handle করতে পারে without blocking the main thread। অর্থাৎ, JavaScript অন্য tasks complete হওয়ার অপেক্ষা না করে পরবর্তী tasks execute করতে পারে।

Understanding with Simple Terms:

ধরুন, আপনি রেস্টুরেন্টে খাবার অর্ডার করেছেন। ওয়েটার আপনার অর্ডার নিয়ে কিচেনে পাঠাল এবং তখনই অন্য কাস্টমারের অর্ডার নিতে শুরু করল। আপনার খাবার প্রস্তুত হয়ে গেলে ওয়েটার আপনার কাছে নিয়ে আসবে। এখানে, ওয়েটার একসঙ্গে অনেক কাজ handle করছে without waiting for one task to complete। ঠিক একইভাবে, Asynchronous JavaScript multiple tasks manage করতে পারে একসঙ্গে।

Importance of Asynchronous JavaScript:

- **Improved Performance:** Long-running tasks যেমন data fetching বা file processing main thread কে block না করে background এ চলতে পারে।
- **Better User Experience:** UI responsive থাকে এবং user interactions smoothly handle করা যায়।

- **Efficient Resource Utilization:** System resources efficiently ব্যবহার হয়, কারণ tasks parallelly execute হয়।

2. Synchronous vs Asynchronous Programming

Synchronous Programming

Synchronous Programming এ code sequentially execute হয়। একটি task complete না হওয়া পর্যন্ত পরবর্তী task শুরু হয় না।

Example:

```
console.log("Task 1: Starting");
console.log("Task 2: In Progress");
console.log("Task 3: Completed");
```

Console Output:

```
Task 1: Starting
Task 2: In Progress
Task 3: Completed
```

Explanation: এখানে, প্রতিটি console.log statement sequentially execute হচ্ছে। প্রথমটি complete হওয়ার পর দ্বিতীয়টি শুরু হচ্ছে, এভাবে ক্রমান্বয়ে চলছে।

Asynchronous Programming

Asynchronous Programming এ tasks concurrently execute হয়। একটি task complete হওয়ার জন্য অপেক্ষা না করে পরবর্তী task শুরু হয়ে যায়।

Example:

```
console.log("Task 1: Starting");

setTimeout(function () {
  console.log("Task 2: In Progress");
}, 2000);

console.log("Task 3: Completed");
```

Console Output:

```
Task 1: Starting
Task 3: Completed
Task 2: In Progress
```

Explanation:

- Task 1 immediate execute হয়।
- setTimeout function asynchronous ভাবে execute হয় এবং 2 সেকেন্ড delay করে Task 2 print করে।
- meantime, Task 3 execute হয়ে যায় without waiting for Task 2 to complete।

Visualization:

```
Time 0s: Task 1 executed
Time 0s: Task 3 executed
Time 2s: Task 2 executed
```

Benefits of Asynchronous Programming:

- Long-running operations UI কে freeze করে না।
- Multiple operations parallelly handle করা যায়।
- Network requests এবং file operations efficiently manage করা যায়।

3. Callbacks in Asynchronous JavaScript

What is a Callback?

Callback হলো একটি function যেটি অন্য একটি function এর argument হিসেবে pass হয় এবং নির্দিষ্ট কাজ সম্পন্ন হওয়ার পর call হয়।

How Callbacks Enable Asynchronous Behavior:

JavaScript এ callbacks asynchronous operations handle করতে ব্যবহৃত হয়। যখন একটি function asynchronous কাজ complete করে, তখন callback function execute হয়।

Example:

```
function fetchData(callback) {
  console.log("Fetching data...");

  setTimeout(function () {
    const data = { name: "John", age: 30 };
    console.log("Data fetched");
    callback(data);
  }, 3000);
}

function processData(data) {
  console.log("Processing data...");
  console.log(`Name: ${data.name}, Age: ${data.age}`);
}

fetchData(processData);
```

Console Output:

```
Fetching data...
Data fetched
Processing data...
Name: John, Age: 30
```

Explanation:

1. fetchData function asynchronous ভাবে data fetch করে (simulate করা হয়েছে setTimeout দিয়ে)।
2. Data fetch complete হওয়ার পর, callback হিসেবে pass করা processData function execute হয় এবং fetched data process করে।

Real-life Analogy:

- **Fetching Data:** আপনি কোনো website থেকে information download করছেন।
- **Callback Function:** ডাউনলোড complete হওয়ার পর সেই information process করছেন।

Nested Callbacks and Callback Hell:

Multiple asynchronous operations sequentially execute করতে গেলে nested callbacks ব্যবহৃত হয়, যা code কে complex এবং unreadable করে তোলে। একে বলা হয় **Callback Hell**।

Example of Callback Hell:

```
function firstTask(callback) {
  setTimeout(function () {
    console.log("First task completed");
    callback();
  }, 1000);
}

function secondTask(callback) {
  setTimeout(function () {
    console.log("Second task completed");
    callback();
  }, 1000);
}

function thirdTask() {
  setTimeout(function () {
    console.log("Third task completed");
  }, 1000);
}

firstTask(function () {
  secondTask(function () {
    thirdTask();
  });
});
```

Console Output:

```
First task completed
Second task completed
Third task completed
```

Problems with Callback Hell:

- Code readability কমে যায়।
- Error handling কঠিন হয়ে যায়।
- Maintenance এবং debugging challenging হয়।

Solution:

- **Promises** এবং **Async/Await** ব্যবহার করে callback hell avoid করা যায়।

4. Promises

What is a Promise?

Promise হলো JavaScript এ asynchronous operations handle করার modern approach। এটি asynchronous operation এর eventual completion বা failure কে represent করে এবং সেই অনুযায়ী value return করে।

States of a Promise:

1. **Pending:** Initial state, neither fulfilled nor rejected।
2. **Fulfilled:** Operation সফলভাবে complete হয়েছে।
3. **Rejected:** Operation ব্যর্থ হয়েছে।

Advantages of Promises over Callbacks:

- **Better Readability:** Code sequentially পড়া যায়।
- **Error Handling:** Errors সহজে catch করা যায়।
- **Avoids Callback Hell:** Nested callbacks avoid করা যায়।

Creating a Promise

Basic Structure:

```
const promise = new Promise(function (resolve, reject) {
  // asynchronous operation
});
```

Explanation:

- resolve function call হয় যখন operation successful হয়।
- reject function call হয় যখন operation failure হয়।

Example:

```
function fetchData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      const success = true; // Try changing this to false to see rejection
      if (success) {
        const data = { name: "Alice", age: 25 };
        console.log("Data fetched successfully");
        resolve(data);
      } else {
        reject("Error fetching data");
      }
    }, 2000);
  });
}

function processData(data) {
  console.log("Processing data...");
  console.log(`Name: ${data.name}, Age: ${data.age}`);
}

fetchData()
  .then(processData)
  .catch(function (error) {
    console.error(error);
  });
```

Console Output (when success is true):

```
Data fetched successfully
Processing data...
Name: Alice, Age: 25
```

Console Output (when success is false):

```
Error fetching data
```

Explanation:

- fetchData function একটি promise return করে।
- then method use করে successful result handle করা হয়।
- catch method use করে errors handle করা হয়।

Chaining Promises

Multiple asynchronous operations sequentially execute করতে promises chaining করা হয়।

Example:

```
function stepOne() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Step One completed");
      resolve("Data from Step One");
    }, 1000);
  });
}

function stepTwo(data) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log(`Step Two received: ${data}`);
      resolve("Data from Step Two");
    }, 1000);
  });
}

function stepThree(data) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log(`Step Three received: ${data}`);
      resolve("All steps completed");
    }, 1000);
  });
}

stepOne()
  .then(stepTwo)
  .then(stepThree)
  .then(function (result) {
    console.log(result);
  })
  .catch(function (error) {
    console.error(error);
  });
```

Console Output:

```
Step One completed
Step Two received: Data from Step One
Step Three received: Data from Step Two
All steps completed
```

Explanation:

- Each function returns a promise and passes data to the next function in the chain.
- This approach maintains code readability and manages asynchronous tasks efficiently.

Handling Errors with Promises

Promises provide robust error handling mechanisms.

Example:

```
function fetchUserData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      const error = false;
      if (!error) {
        resolve({ username: "Bob", email: "bob@example.com" });
      } else {
        reject("Failed to fetch user data");
      }
    }, 1000);
  });
}

fetchUserData()
  .then(function (user) {
    console.log(`User fetched: ${user.username}`);
  })
  .catch(function (error) {
    console.error(`Error: ${error}`);
  })
  .finally(function () {
    console.log("Operation completed");
  });
```

Console Output (when error is false):

```
User fetched: Bob
Operation completed
```

Console Output (when error is true):

```
Error: Failed to fetch user data
Operation completed
```

Explanation:

- catch method catches any errors during the promise execution.

- finally method executes regardless of success or failure, useful for cleanup operations.

5. Async/Await

What is Async/Await?

Async/Await হলো promises এর উপর ভিত্তি করে তৈরি syntax যা asynchronous code কে synchronous মত দেখায় এবং সহজে manage করতে সাহায্য করে।

Using Async Functions

Async functions declare করা হয় async keyword দিয়ে এবং asynchronous operations এর আগে await keyword use করা হয়।

Example:

```
function fetchData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      const data = { product: "Laptop", price: 1500 };
      resolve(data);
    }, 2000);
  });
}

async function getData() {
  console.log("Fetching data...");
  try {
    const result = await fetchData();
    console.log("Data received:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

getData();
```

Console Output:

```
Fetching data...
Data received: { product: 'Laptop', price: 1500 }
```

Explanation:

- getData function asynchronous হলেও code sequentially পড়া যায়।
- await fetchData এর result পাওয়ার জন্য wait করে।
- try...catch block use করে errors handle করা হয়।

Error Handling in Async/Await

Errors সহজে handle করা যায় try...catch blocks এর মাধ্যমে।

Example:

```
function fetchUser() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      const error = true;
      if (!error) {
        resolve({ user: "Charlie", age: 28 });
      } else {
        reject("User not found");
      }
    }, 1000);
  });
}

async function displayUser() {
  try {
    const user = await fetchUser();
    console.log(`User: ${user.user}, Age: ${user.age}`);
  } catch (error) {
    console.error("Error:", error);
  } finally {
    console.log("Execution completed");
  }
}


displayUser();
```

Console Output:

```
Error: User not found
Execution completed
```

Explanation:

- Error occurs in fetchUser promise and is caught in the catch block.
- finally block executes regardless of success or failure.

Benefits of Async/Await:

- Cleaner and more readable code.
- Easier error handling.
- Sequential execution makes logic easier to follow.

6. Real-life Examples of Asynchronous JavaScript

Fetching Data from an API

Suppose you want to fetch user data from an external API and display it on your webpage.

Example:

```
async function getUser() {
  try {
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/users/1"
    );
    const user = await response.json();
    console.log("User Name:", user.name);
    console.log("User Email:", user.email);
  } catch (error) {
    console.error("Error fetching user:", error);
  }
}

getUser();
```

Console Output:

```
User Name: Leanne Graham
User Email: Sincere@april.biz
```

Explanation:

- fetch API asynchronously data retrieve করে।
- await keywords response এবং json conversion এর জন্য wait করে।
- Errors network issues বা invalid responses handle করতে try...catch block ব্যবহৃত হয়।

Reading Files

Suppose you want to read a file from the file system using Node.js.

Example (Node.js):

```
const fs = require("fs").promises;

async function readFile() {
  try {
    const data = await fs.readFile("example.txt", "utf8");
    console.log("File Content:", data);
  } catch (error) {
    console.error("Error reading file:", error);
  }
}

readFile();
```

Console Output:

```
File Content: This is an example file content.
```

Explanation:

- fs.readFile asynchronously file read করে এবং promise return করে।
- await keyword file content পাওয়ার জন্য wait করে।
- Errors যেমন file not found সহজে catch করা যায়।

Note: Ensure example.txt file exists in your project directory for successful execution.

7. Conclusion

Asynchronous JavaScript modern web development এর একটি essential part। এটি applications কে responsive এবং efficient রাখে। Callbacks, Promises, এবং Async/Await বিভিন্ন পরিস্থিতিতে asynchronous operations handle করতে সাহায্য করে। Proper understanding এবং implementation এর মাধ্যমে complex tasks সহজে manage করা যায় এবং user experience উন্নত হয়।

JavaScript Promises

Table of Contents

1. Introduction to Promises

Promises হলো JavaScript এর একটি powerful feature যা asynchronous কাজগুলোকে handle করতে সাহায্য করে। Promise মূলত একটি object, যা একটি asynchronous কাজের future result represent করে। একটি Promise asynchronous কাজের completion বা failure এর result ধরে রাখে এবং এই result এর উপর নির্ভর করে actions perform করা হয়।

2. Why Use Promises?

JavaScript এ asynchronous কাজ করতে গেলে callback functions ব্যবহার করা হয়। কিন্তু অনেক সময় nested callbacks বা "callback hell" এর সৃষ্টি হয়, যা code পড়তে এবং maintain করতে অসুবিধা সৃষ্টি করে। Promises এই সমস্যা সমাধান করতে সাহায্য করে এবং asynchronous কাজগুলোকে sequentially ও effectively handle করতে সহজ করে তোলে।

3. States of a Promise

একটি Promise এর তিনটি প্রধান state রয়েছে:

1. **Pending:** Promise যখন initiate করা হয়, তখন এটি pending state এ থাকে, অর্থাৎ কাজটি এখনো সম্পন্ন হয়নি।
2. **Fulfilled (Resolved):** যদি asynchronous কাজটি সফলভাবে সম্পন্ন হয়, তবে Promise fulfilled বা resolved state এ চলে যায়। এই সময়ে resolve function call করা হয় এবং Promise এর result পাওয়া যায়।
3. **Rejected:** যদি asynchronous কাজটি সফলভাবে সম্পন্ন না হয়, তবে Promise rejected state এ চলে যায়। এই সময়ে reject function call করা হয় এবং error handle করা হয়।

4. Creating a Promise

একটি Promise তৈরি করতে হলে, নতুন Promise constructor (new Promise) ব্যবহার করতে হয়। এই constructor দুটি function নেয়: resolve এবং reject.

Example:

```
let promise = new Promise(function (resolve, reject) {
  let success = true;
  if (success) {
    resolve("The operation was successful!");
  } else {
    reject("The operation failed.");
  }
});
```

Explanation: এখানে একটি Promise তৈরি করা হয়েছে যা success বা failure এর উপর ভিত্তি করে resolve বা reject করবে।

5. Consuming Promises

একটি Promise create করার পর, সেই Promise এর result handle করার জন্য .then(), .catch(), এবং .finally() method গুলো ব্যবহার করা হয়।

5.1 Using .then()

.then() method ব্যবহার করে fulfilled (resolved) Promise এর result handle করা হয়।

Example:

```
let promise = new Promise(function (resolve, reject) {
  resolve("Data fetched successfully!");
});

promise.then(function (message) {
  console.log(message);
});
```

Console Output:

```
Data fetched successfully!
```

Explanation: এখানে Promise resolve হওয়ার পর .then() method এর মাধ্যমে result handle করা হয়েছে এবং console এ message print হয়েছে।

5.2 Handling Errors with .catch()

.catch() method ব্যবহার করে rejected Promise এর error handle করা হয়।

Example:

```
let promise = new Promise(function (resolve, reject) {
  reject("Failed to fetch data.");
});

promise
  .then(function (message) {
    console.log(message);
  })
  .catch(function (error) {
    console.log("Error: " + error);
  });
```

Console Output:

```
Error: Failed to fetch data.
```

Explanation: এখানে Promise reject হলে .catch() method এর মাধ্যমে error handle করা হয়েছে।

5.3 Using .finally()

.finally() method ব্যবহার করা হয় যখন Promise fulfilled বা rejected যাই হোক না কেন, শেষে কিছু কাজ করতে হয়।

Example:

```
let promise = new Promise(function (resolve, reject) {
  resolve("Data processed successfully!");
});

promise
  .then(function (message) {
    console.log(message);
  })
  .catch(function (error) {
    console.log("Error: " + error);
  })
  .finally(function () {
    console.log("This will run regardless of the result.");
  });
```

Console Output:

```
Data processed successfully!
This will run regardless of the result.
```

Explanation: এখানে Promise resolve হওয়ার পরে .finally() block execute হয়েছে, যা result এর উপর নির্ভর করে না।

6. Chaining Promises

Promises কে chain করা যায় যাতে একের পর এক asynchronous কাজ সম্পন্ন করা যায়। এটি sequential asynchronous operations handle করতে খুবই কার্যকর।

Example:

```
let promise = new Promise(function (resolve, reject) {
  resolve("Step 1 completed");
});

promise
  .then(function (message) {
    console.log(message);
    return "Step 2 completed";
  })
  .then(function (message) {
    console.log(message);
    return "Step 3 completed";
  })
  .then(function (message) {
    console.log(message);
  })
  .catch(function (error) {
    console.log("Error: " + error);
  });
```

Console Output:

```
Step 1 completed
Step 2 completed
Step 3 completed
```

Explanation: এখানে sequential ভাবে একের পর এক Promise resolve হয়ে কাজগুলো সম্পন্ন হয়েছে। যদি কোনো এক জায়গায় error ঘটে, তাহলে .catch() block এ সেটা handle করা হয়।

7. Real-life Example of Promises

Scenario: Fetching Data from a Server and Processing It

ধরুন, আপনি একটি application তৈরি করছেন যেখানে server থেকে data fetch করতে হবে এবং সেই data process করতে হবে। এখানে Promises ব্যবহার করে sequentially কাজগুলো সম্পন্ন করা হবে।

```
function fetchData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("Data fetched from server");
    }, 2000);
  });
}

function processData(data) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(`${data} and processed successfully`);
    }, 1500);
  });
}

fetchData()
  .then(function (result) {
    console.log(result);
    return processData(result);
  })
  .then(function (result) {
    console.log(result);
  })
  .catch(function (error) {
    console.log("Error: " + error);
});
```

Console Output:

```
Data fetched from server
Data fetched from server and processed successfully
```

Detailed Explanation:

1. **fetchData:** প্রথমে data fetch করার জন্য একটি Promise return করা হয়েছে যা ২ সেকেন্ড পর resolve হয়।
2. **processData:** Data fetch হওয়ার পরে, সেই data process করার জন্য আরেকটি Promise return করা হয়েছে যা ১.৫ সেকেন্ড পরে resolve হয়।
3. **Promise Chaining:** এখানে Promises chain করে sequentially data fetch এবং data process করা হয়েছে।

8. Conclusion

JavaScript Promises asynchronous কাজগুলোকে handle করতে অত্যন্ত কার্যকর এবং powerful একটি পদ্ধতি। Promises এর মাধ্যমে asynchronous operations কে sequentially এবং efficiently manage করা যায়। Promises এর মাধ্যমে asynchronous কাজ সহজে পড়া যায় এবং maintain করা যায়, যা callback functions এর ক্ষেত্রে অনেক কঠিন হতে পারে।

JavaScript Async/Await

Table of Contents

1. Introduction to Async/Await

Async/Await হলো JavaScript এর modern syntax যা Promises এর ওপর ভিত্তি করে তৈরি করা হয়েছে। এটি asynchronous code লেখার একটি নতুন উপায় যা asynchronous operations কে synchronous code এর মতো দেখতে এবং behave করতে সাহায্য করে। Async/Await এর মাধ্যমে asynchronous code আরও পড়তে সহজ হয় এবং callback বা promise chaining এর জটিলতা এড়ানো যায়।

2. Why Use Async/Await?

Async/Await ব্যবহার করার কারণগুলোর মধ্যে অন্যতম হলো:

1. **Readability:** Async/Await এর মাধ্যমে asynchronous code কে synchronous code এর মতো দেখায়, যা পড়তে এবং বোঝা সহজ।
2. **Error Handling:** Async/Await এর মাধ্যমে error handling করা অনেক সহজ। এটি try/catch block ব্যবহার করে errors handle করা যায়।
3. **Cleaner Code:** Async/Await এর মাধ্যমে nested callbacks বা promise chaining এর জটিলতা এড়ানো যায় এবং code আরও পরিষ্কার ও সুন্দর হয়।

3. Understanding Async Functions

async keyword ব্যবহার করে একটি function কে asynchronous function হিসেবে declare করা হয়। একটি async function স্বাভাবিকভাবে একটি Promise return করে।

Example:

```
async function fetchData() {
  return "Data fetched successfully!";
}


fetchData().then(function (result) {
  console.log(result);
});
```

Console Output:

```
Data fetched successfully!
```

Explanation: এখানে fetchData একটি async function যা একটি Promise return করে এবং .then() method এর মাধ্যমে সেই result handle করা হয়।

4. Using the await Keyword

await keyword শুধুমাত্র async function এর মধ্যে ব্যবহার করা যায়। এটি একটি Promise resolve বা reject হওয়া পর্যন্ত অপেক্ষা করে এবং তারপর result return করে। await এর মাধ্যমে asynchronous operations কে synchronous code এর মতো করে লেখা যায়।

Example:

```
function fetchData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("Data fetched from server");
    }, 2000);
  });
}

async function processData() {
  let data = await fetchData();
  console.log(data);
  console.log("Processing data...");
}

processData();
```

Console Output:

```
Data fetched from server
Processing data...
```

Explanation: এখানে await keyword ব্যবহার করে fetchData function এর Promise resolve হওয়া পর্যন্ত অপেক্ষা করা হয়েছে এবং তারপর result handle করা হয়েছে।

5. Error Handling with Async/Await

Async/Await এর মাধ্যমে error handling করা খুবই সহজ। try/catch block ব্যবহার করে asynchronous code এ errors handle করা যায়।

Example:

```
function fetchData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      reject("Failed to fetch data");
    }, 2000);
  });
}

async function processData() {
  try {
    let data = await fetchData();
    console.log(data);
  } catch (error) {
    console.log("Error: " + error);
  }
}

processData();
```

Console Output:

```
Error: Failed to fetch data
```

Explanation: এখানে try/catch block ব্যবহার করে fetchData function এর মধ্যে কোনো error থাকলে তা handle করা হয়েছে।

6. Real-life Example of Async/Await

Scenario: Fetching and Processing Multiple API Data

ধরুন, আপনি একটি application তৈরি করছেন যেখানে multiple API থেকে data fetch করতে হবে এবং সেই data process করতে হবে। Async/Await ব্যবহার করে এই কাজগুলো sequentially এবং সহজে handle করা যাবে।

```
function fetchUserData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("User data fetched");
    }, 2000);
  });
}

function fetchOrdersData() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("Orders data fetched");
    }, 1500);
  });
}

async function processData() {
  try {
    let userData = await fetchUserData();
    console.log(userData);

    let ordersData = await fetchOrdersData();
    console.log(ordersData);

    console.log("Data processing complete.");
  } catch (error) {
    console.log("Error: " + error);
  }
}

processData();
```

Console Output:

```
User data fetched
Orders data fetched
```

Data processing complete.

Detailed Explanation:

1. **fetchUserData:** প্রথমে user data fetch করার জন্য একটি Promise return করা হয়েছে যা ২ সেকেন্ড পরে resolve হয়।
2. **fetchOrdersData:** তারপর orders data fetch করার জন্য আরেকটি Promise return করা হয়েছে যা ১.৫ সেকেন্ড পরে resolve হয়।
3. **Async/Await and Try/Catch:** Async/Await এবং try/catch block ব্যবহার করে sequentially data fetch এবং process করা হয়েছে। যদি কোনো error ঘটে, তাহলে তা catch block এ handle করা হয়।

7. Conclusion

Async/Await হলো JavaScript এর একটি modern এবং powerful feature যা asynchronous code কে synchronous code এর মতো দেখতে এবং behave করতে সাহায্য করে। এটি asynchronous কাজগুলোকে সহজ এবং পরিষ্কারভাবে handle করতে সাহায্য করে এবং callback hell বা promise chaining এর জটিলতা থেকে মুক্তি দেয়। Async/Await এর মাধ্যমে JavaScript এর asynchronous প্রোগ্রামিং আরও readable এবং maintainable হয়।

↥ Go to Top

Chapter-16: DOM, DOM Methods, DOM Documents, DOM Elements, DOM Events, DOM Event Listener, DOM Nodes, DOM Collections, DOM NodeList Object

Document Object Model

Table of Contents

1. Introduction to the DOM

- **Document Object Model (DOM)** হলো একটি প্রোগ্রামিং interface যা HTML এবং XML document কে একটি tree structure হিসেবে represent করে। DOM এর মাধ্যমে JavaScript ব্যবহার করে একটি webpage এর elements dynamically access এবং manipulate করা সম্ভব হয়।
- প্রোগ্রামিং ইন্টারফেস (Programming Interface) বলতে একটি সফটওয়্যার বা সিস্টেমের সাথে অন্য সফটওয়্যার বা প্রোগ্রাম কিভাবে ইন্টার‍্যাক্ট করবে তা নির্দেশ করে। এটি সাধারণত দুটি প্রোগ্রামের মধ্যে ডেটা বিনিময়ের একটি মাধ্যম হিসেবে কাজ করে।

দুটি সাধারণ ধরনের প্রোগ্রামিং ইন্টারফেস রয়েছে:

1. **API (Application Programming Interface):** এটি সফটওয়্যার অ্যাপ্লিকেশনগুলোর মধ্যে যোগাযোগের একটি মাধ্যম। API এর মাধ্যমে একটি প্রোগ্রাম অন্য একটি প্রোগ্রাম থেকে ডেটা বা ফাংশনালিটি অ্যাক্সেস করতে পারে। উদাহরণস্বরূপ, ওয়েব API এর মাধ্যমে অন্য ওয়েবসাইট থেকে ডেটা সংগ্রহ করা সম্ভব।

2. **GUI (Graphical User Interface):** এটি ব্যবহারকারী এবং সফটওয়্যার বা সিস্টেমের মধ্যে ইন্টারঅ্যাক্ট করার ভিজুয়াল মাধ্যম। প্রোগ্রামিংয়ের ক্ষেত্রে GUI বলতে এমন ইন্টারফেস বোঝায় যেখানে ব্যবহারকারী বাটন, আইকন, মেনু ইত্যাদি ব্যবহার করে সফটওয়্যার নিয়ন্ত্রণ করতে পারে।

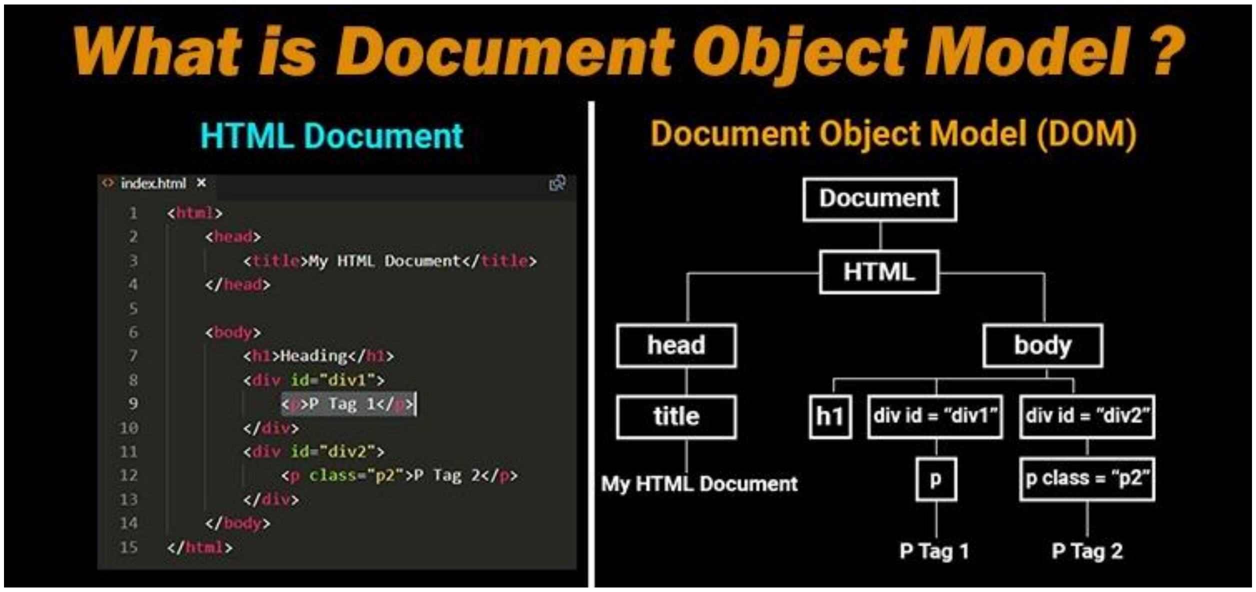
- DOM মূলত browser এবং webpage এর মধ্যে একটি bridge হিসাবে কাজ করে।
- When a web page is loaded, the browser creates a DOM of the page.

2. Why Use the DOM?

DOM এর মাধ্যমে JavaScript ব্যবহার করে একটি webpage এর content এবং structure পরিবর্তন করা যায়। User interaction এর ভিত্তিতে dynamically HTML elements update, remove, অথবা add করা যায়। DOM manipulation user experience এবং webpage এর interactivity উন্নত করতে সহায়ক।

3. DOM Structure

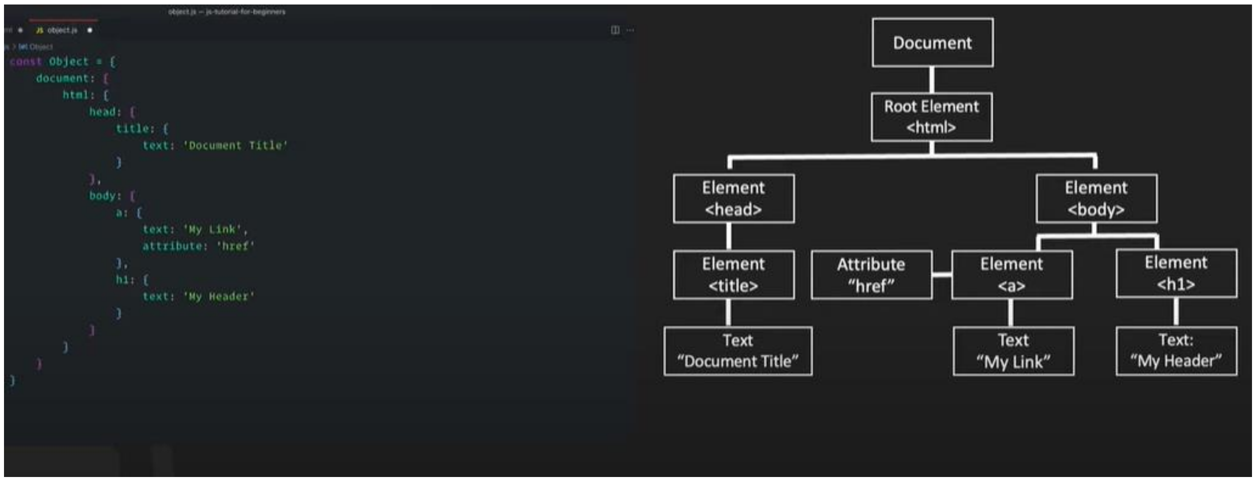
DOM একটি tree structure হিসেবে কাজ করে, যেখানে প্রতিটি HTML document এর elements, attributes, এবং text গুলোকে node হিসেবে consider করা হয়। DOM tree এর প্রতিটি node একটি object হিসেবে কাজ করে, এবং JavaScript এর মাধ্যমে সেই nodes access এবং manipulate করা যায়।



```
<html>
    <head>
        <title>My HTML Document</title>
    </head>

    <body>
        <h1>Heading</h1>
        <div id="div1">
            <p>P Tag 1</p>
        </div>
        <div id="div2">
            <p class="p2">P Tag 2</p>
        </div>
    </body>
</html>
```

আমরা একটি HTML Document কে এভাবে Object আকারে চিন্তা করতে পারিঃ



```
const Object = {
    document: {
        html: {
            head: {
                title: {
                    text: 'Document Title'
                }
            },
            body: {
                a: {
                    text: 'My Link',
                    attribute: 'href'
                },
                h1: {
                    text: 'My Header'
                }
            }
        }
    }
}
```

3.1 Node Types

DOM এ বিভিন্ন ধরনের nodes রয়েছে:

1. **Element Node:** HTML tags গুলো DOM এ element node হিসেবে কাজ করে। যেমন: <p>, <div>, <img>.
2. **Text Node:** Element এর মধ্যে থাকা text গুলো text node হিসেবে থাকে।
3. **Attribute Node:** HTML element এর attributes গুলো attribute node হিসেবে থাকে। যেমন: id, class, src ইত্যাদি।

3.2 Parent-Child Relationships

DOM tree তে প্রতিটি node এর মধ্যে parent-child সম্পর্ক থাকে। কোন element এর মধ্যে যদি অন্য element থাকে, তাহলে parent-child relationship তৈরি হয়।

```
<div>
  <p>This is a paragraph</p>
</div>
```

Explanation: এখানে <div> হলো parent এবং <p> হলো child।

4. Accessing DOM Elements

JavaScript এর মাধ্যমে DOM এর elements access করার জন্য বিভিন্ন method রয়েছে।

4.1 Using getElementById()

getElementById() method একটি element এর id attribute এর মাধ্যমে সেই element কে access করতে সাহায্য করে।

Example:

```
<p id="myParagraph">This is a paragraph</p>

<script>
  let element = document.getElementById("myParagraph");
  console.log(element.innerText);
</script>
```

Console Output:

```
This is a paragraph
```

Explanation: এখানে id="myParagraph" সহ একটি paragraph element কে JavaScript এর মাধ্যমে access করা হয়েছে।

4.2 Using querySelector()

querySelector() CSS selectors এর মতো করে প্রথম matching element return করে।

Example:

```
<div class="container">
  <p>This is inside the container</p>
</div>

<script>
  let element = document.querySelector(".container p");
  console.log(element.innerText);
</script>
```

Console Output:

```
This is inside the container
```

Explanation: এখানে CSS selector ব্যবহার করে class="container" এর মধ্যে থাকা p element কে access করা হয়েছে।

5. Manipulating DOM Elements

DOM এর elements access করার পরে JavaScript ব্যবহার করে তাদের modify করা যায়।

5.1 Changing Text Content

DOM এর element এর text content পরিবর্তন করতে innerText অথবা textContent property ব্যবহার করা যায়।

Example:

```
<p id="myText">Old Text</p>

<script>
  document.getElementById("myText").innerText = "New Text";
</script>
```

Output on Webpage:

```
New Text
```

Explanation: এখানে innerText ব্যবহার করে paragraph element এর content পরিবর্তন করা হয়েছে।

5.2 Changing Attributes

Element এর attributes, যেমন: src, href, alt, dynamically JavaScript এর মাধ্যমে পরিবর্তন করা যায়।

Example:

```
<img id="myImage" src="old_image.jpg" alt="Old Image" />

<script>
  document.getElementById("myImage").src = "new_image.jpg";
</script>
```

Explanation: এখানে image এর src attribute পরিবর্তন করে নতুন image set করা হয়েছে।

5.3 Changing Styles

CSS styles dynamically JavaScript এর মাধ্যমে পরিবর্তন করা যায়।

Example:

```
<p id="myText">Change my style!</p>

<script>
```

```
  document.getElementById("myText").style.color = "blue";
  document.getElementById("myText").style.fontSize = "20px";
</script>
```

Output on Webpage:

(This text will appear in blue and 20px size)

Explanation: এখানে style object ব্যবহার করে text এর color এবং font size পরিবর্তন করা হয়েছে।

6. Traversing the DOM

DOM এর elements এর মধ্যে traversal করা যায়, অর্থাৎ একটি element থেকে এর parent, child, sibling elements access করা যায়।

Example:

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>


<script>
  let list = document.getElementById("myList");
  let firstItem = list.firstElementChild; // Access first child
  console.log(firstItem.innerText);
</script>
```

Console Output:

```
Item 1
```

Explanation: এখানে firstElementChild property ব্যবহার করে list এর প্রথম element কে access করা হয়েছে।

7. Real-life Example of DOM Manipulation

Scenario-01: Adding Items to a To-do List

ধরুন, আপনি একটি To-do list তৈরি করছেন যেখানে user নতুন কাজ যোগ করতে পারে। DOM এর মাধ্যমে dynamically নতুন items list এ add করা হবে।

Example:

```
<h2>To-do List</h2>
<ul id="todoList">
  <li>Learn JavaScript</li>
</ul>

<input type="text" id="newTask" placeholder="Add a new task" />
<button onclick="addTask()">Add Task</button>

<script>
  function addTask() {
    let task = document.getElementById("newTask").value;
    let list = document.getElementById("todoList");

    let newItem = document.createElement("li");
    newItem.innerText = task;

    list.appendChild(newItem);
  }
</script>
```

Output on Webpage:

```
To-do List
- Learn JavaScript
- (Newly added task by user)
```

Detailed Explanation:

1. **Accessing Elements:** Input field এবং unordered list কে JavaScript এর মাধ্যমে access করা হয়েছে।
2. **Creating New Element:** createElement() method ব্যবহার করে নতুন li element তৈরি করা হয়েছে।

3. **Appending New Item:** নতুন task add করার জন্য dynamically নতুন element list এ append করা হয়েছে।

Scenario-02: Updating User Profile Information

ধরুন, আপনি একটি profile update form তৈরি করছেন যেখানে user এর profile information update হওয়ার পর user কে একটি confirmation message দেখাতে হবে।

Example:

```
<h2>User Profile</h2>
<p id="username">Rahim</p>
<p id="email">rahim@gmail.com</p>

<!-- Input fields for username and email -->
<label for="newUsername">New Username:</label>
<input type="text" id="newUsername" placeholder="Enter new username"><br><br>

<label for="newEmail">New Email:</label>
<input type="email" id="newEmail" placeholder="Enter new email"><br><br>

<button onclick="updateProfile()">Update Profile</button>

<script>
function updateProfile() {
   // Get the values from the input fields
   const newUsername = document.getElementById('newUsername').value;
   const newEmail = document.getElementById('newEmail').value;

   // Update the profile information
   document.getElementById('username').innerText = newUsername;
   document.getElementById('email').innerText = newEmail;
```

```
    // Display a confirmation message
    let confirmation = document.createElement('p');
    confirmation.innerText = 'Profile updated successfully!';
    document.body.appendChild(confirmation);
}
</script>
```

8. Conclusion

Document Object Model (DOM) হলো JavaScript এর মাধ্যমে HTML document কে dynamically access এবং modify করার একটি powerful tool। DOM এর মাধ্যমে elements কে access করে তাদের content, attributes, এবং styles পরিবর্তন করা যায়, যা modern dynamic websites তৈরি করতে অপরিহার্য। DOM traversal এর মাধ্যমে webpage এর বিভিন্ন অংশের মধ্যে navigation করা সম্ভব, এবং user interaction এর ভিত্তিতে page এর structure dynamically পরিবর্তন করা যায়।

DOM Methods

Table of Contents

1. Introduction to DOM Methods

DOM methods হলো JavaScript এর functions, যা HTML document এর elements গুলো dynamically access, manipulate, এবং modify করতে সাহায্য করে। DOM methods এর মাধ্যমে webpage এর elements এর structure, attributes, এবং content পরিবর্তন করা সম্ভব হয়।

2. Commonly Used DOM Methods

DOM এর বিভিন্ন methods রয়েছে, যা বিভিন্ন কাজ করতে ব্যবহার করা হয়। নিচে কিছু commonly used DOM methods এর উদাহরণ দেওয়া হলো:

2.1 getElementById()

getElementById() method একটি HTML element এর id attribute এর মাধ্যমে সেই element কে access করতে সাহায্য করে। এটি একটি element return করে।

Example:

```
<p id="myParagraph">This is a paragraph</p>


<script>
  let element = document.getElementById("myParagraph");
  console.log(element.innerText);
</script>
```

Console Output:

```
This is a paragraph
```

Explanation: এখানে id="myParagraph" সহ একটি paragraph element কে JavaScript এর মাধ্যমে access করা হয়েছে।

2.2 getElementsByClassName()

getElementsByClassName() method ব্যবহার করে HTML এর elements এর class attribute এর মাধ্যমে একাধিক element কে access করা যায়। এটি একটি collection (HTMLCollection) return করে।

Example:

```
<p class="myClass">Paragraph 1</p>
<p class="myClass">Paragraph 2</p>


<script>
  let elements = document.getElementsByClassName("myClass");
  console.log(elements[0].innerText); // First element
  console.log(elements[1].innerText); // Second element
</script>
```

Console Output:

```
Paragraph 1
Paragraph 2
```

Explanation: এখানে class="myClass" সহ দুটি paragraph element কে access করা হয়েছে এবং তাদের text content console এ print করা হয়েছে।

2.3 querySelector() and querySelectorAll()

querySelector() method CSS selector এর মতো করে প্রথম matching element return করে। আর querySelectorAll() method সব matching elements return করে, যা NodeList হিসেবে থাকে।

Example:

```
<div class="container">
  <p>This is a paragraph inside a container.</p>
</div>


<script>
  let element = document.querySelector(".container p");
  console.log(element.innerText);
</script>
```

Console Output:

```
This is a paragraph inside a container.
```

Explanation: এখানে querySelector() method ব্যবহার করে CSS selector এর মতো করে class="container" এর মধ্যে থাকা p element কে access করা হয়েছে।

2.4 createElement()

createElement() method ব্যবহার করে JavaScript এর মাধ্যমে নতুন HTML element তৈরি করা যায়।

Example:

```
<ul id="myList">
  <li>Item 1</li>
```

```
</ul>

<script>
  let newItem = document.createElement("li");
  newItem.innerText = "Item 2";
  document.getElementById("myList").appendChild(newItem);
</script>
```

Output on Webpage:

```
Item 1
Item 2
```

Explanation: এখানে createElement() method ব্যবহার করে নতুন li element তৈরি করা হয়েছে এবং list এ append করা হয়েছে।

2.5 appendChild()

appendChild() method একটি নতুন element কে কোন parent element এর মধ্যে append করতে ব্যবহার করা হয়।

Example:

```
<div id="myDiv">
  <p>First paragraph</p>
</div>

<script>
  let newParagraph = document.createElement("p");
  newParagraph.innerText = "Second paragraph";
  document.getElementById("myDiv").appendChild(newParagraph);
</script>
```

Output on Webpage:

```
First paragraph
Second paragraph
```

Explanation: এখানে appendChild() method ব্যবহার করে নতুন paragraph element কে existing div element এর মধ্যে append করা হয়েছে।

2.6 remove()

remove() method ব্যবহার করে কোনো HTML element কে DOM থেকে সরিয়ে ফেলা যায়।

Example:

```
<p id="myParagraph">This will be removed.</p>

<script>
  let element = document.getElementById("myParagraph");
  element.remove();
</script>
```

Explanation: এখানে remove() method ব্যবহার করে paragraph element টি DOM থেকে সরিয়ে ফেলা হয়েছে।

2.7 setAttribute()

setAttribute() method ব্যবহার করে একটি HTML element এর attribute dynamically set করা যায়।

Example:

```
<img id="myImage" src="old_image.jpg" />

<script>
  document.getElementById("myImage").setAttribute("src", "new_image.jpg");
</script>
```

Explanation: এখানে image element এর src attribute পরিবর্তন করে নতুন image set করা হয়েছে।

2.8 getAttribute()

getAttribute() method ব্যবহার করে একটি HTML element এর attribute এর value পাওয়া যায়।

Example:

```
<img id="myImage" src="image.jpg" alt="My Image" />

<script>
```

```
  let altText = document.getElementById("myImage").getAttribute("alt");
  console.log(altText);
</script>
```

Console Output:

```
My Image
```

Explanation: এখানে image element এর alt attribute এর value পাওয়া হয়েছে।

3. Real-life Example of DOM Methods

Scenario: Dynamically Adding Items to a To-do List

ধরুন, আপনি একটি To-do list তৈরি করছেন যেখানে user নতুন task add করতে পারে এবং সেই task dynamically list এ যোগ হয়।

Example:

```
<h2>User Profile</h2>
<p id="username">Rahim</p>
<p id="email">rahim@gmail.com</p>

<!-- Input fields for username and email -->
<label for="newUsername">New Username:</label>
<input
  type="text"
  id="newUsername"
  placeholder="Enter new username"
/><br /><br />

<label for="newEmail">New Email:</label>
<input type="email" id="newEmail" placeholder="Enter new email" /><br /><br />

<button onclick="updateProfile()">Update Profile</button>

<script>
```

```
  function updateProfile() {
    // Get the values from the input fields
    const newUsername = document.getElementById("newUsername").value;
    const newEmail = document.getElementById("newEmail").value;

    // Update the profile information
    document.getElementById("username").innerText = newUsername;
    document.getElementById("email").innerText = newEmail;

    // Display a confirmation message
    let confirmation = document.createElement("p");
    confirmation.innerText = "Profile updated successfully!";
    document.body.appendChild(confirmation);
  }
</script>
```

4. Conclusion

DOM methods হলো JavaScript এর powerful tools, যা HTML document এর elements dynamically access, modify, এবং manipulate করতে সাহায্য করে। Commonly used DOM methods এর
মধ্যে getElementById(), querySelector(), createElement(), appendChild() এবং remove() ইত্যাদি methods উল্লেখযোগ্য। DOM methods এর মাধ্যমে dynamic এবং interactive webpages তৈরি করা যায়, যা user experience উন্নত করতে সহায়ক।

HTML DOM Document

Table of Contents