Here's a structured guide covering JavaScript basics in detail:

JavaScript Basics – The Language of the Web

JavaScript is a powerful, lightweight programming language that allows us to create dynamic and interactive web applications. It is used for client-side scripting but can also be used on the server side (Node.js).

---

1. Introduction to JavaScript

- JavaScript is a scripting language primarily used for adding interactive behavior to web pages.
- It runs inside web browsers and can manipulate HTML and CSS.
- JavaScript can also be used for backend development using **Node.js**.

How to Run JavaScript?

1. Inside an HTML file using the `<script>` tag:

```
html


<script>
  console.log("Hello, JavaScript!");
</script>
```

2. Inside an external file (`.js`):

```
js

console.log("Hello, JavaScript!");
```

   Link this file inside your HTML:

```
html


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

---

2. Variables and Data Types

JavaScript has three ways to declare variables: `var`, `let`, and `const`.

Declaring Variables

```
js

var a = 10;    // Function-scoped
let b = 20;    // Block-scoped
const c = 30;  // Constant, cannot be reassigned
```

Data Types in JavaScript

JavaScript has 8 primitive data types:

1. **String** → Represents text

```
js

let name = "John";
```

2. **Number** → Represents integers and floating-point numbers

```
js

let age = 25;
let price = 99.99;
```

3. **Boolean** → Represents true or false values

```
js

let isStudent = true;
```

4. **Object** → Collection of key-value pairs

```
js

let person = { name: "Alice", age: 22 };
```

5. **Array** → List of values

```
js

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

6. **Null** → Represents an empty value

```
js

let value = null;
```

7. **Undefined** → A variable declared but not assigned a value

```
js

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

8. **Symbol** → Unique identifiers (advanced)

---

3. Understanding Scope and Hoisting

Scope

Scope determines where a variable can be accessed.

- **Global Scope** – Accessible everywhere.
- **Local Scope** – Declared inside a function, only accessible there.
- **Block Scope** – Variables declared with `let` and `const` inside `{}` cannot be accessed outside.

```
js

let globalVar = "I am global";

function myFunction() {
    let localVar = "I am local";
    console.log(globalVar); // Accessible
}

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

Hoisting

- JavaScript moves variable and function declarations to the top of their scope before execution.
- Only `var` is hoisted **with** `undefined` value.
- `let` and `const` are hoisted but **not initialized**.

```
js

console.log(x); // undefined
var x = 10;

console.log(y); // ReferenceError
let y = 20;
```

---

4. Operators in JavaScript

Arithmetic Operators

```
js

let sum = 10 + 5;   // Addition
let diff = 10 - 5;  // Subtraction
let prod = 10 * 5;  // Multiplication
let div = 10 / 5;   // Division
let mod = 10 % 3;   // Modulus (remainder)
```

Comparison Operators

```
js

console.log(5 == "5");  // true (loose comparison)
console.log(5 === "5"); // false (strict comparison)
console.log(10 > 5);    // true
console.log(10 < 5);    // false
console.log(10 !== 5);  // true
```

Logical Operators

```js
console.log(true && false); // false (AND)
console.log(true || false); // true  (OR)
console.log(!true);         // false (NOT)
```

---

5. Control Flow Statements

Conditional Statements

if, else, else if

```js
let age = 18;

if (age >= 18) {
    console.log("You can vote");
} else {
    console.log("You cannot vote");
}
```

switch Statement

- Used when multiple conditions are checked on a single variable.

```js
let fruit = "Apple";

switch (fruit) {
    case "Apple":
        console.log("It's an Apple!");
        break;
    case "Banana":
        console.log("It's a Banana!");
        break;
    default:
        console.log("Unknown fruit");
}
```

Ternary Operator

A compact way of writing an **`if-else`** condition.

```js
let result = age >= 18 ? "Adult" : "Minor";
console.log(result);
```

Summary of Key Concepts

- **JavaScript Basics**: A scripting language for web development.
- **Variables & Data Types**: **`var`**, **`let`**, **`const`**, and primitive data types.

- **Scope & Hoisting**: Variables have different scopes, and `var` is hoisted.
- **Operators**: Arithmetic, comparison, and logical operators.
- **Control Flow**: `if-else`, `switch`, and ternary operators.

This document provides a strong foundation in JavaScript. Let me know if you want to dive deeper into any specific topic! 🚀

Here's a comprehensive document covering functions in JavaScript:

Functions – Building Blocks of JavaScript

Functions are fundamental in JavaScript, allowing you to encapsulate logic, reuse code, and create more modular applications. They make programs efficient, maintainable, and easier to debug.

---

1. What is a Function and Why is it Essential?

A function is a block of reusable code that performs a specific task. Instead of writing the same code multiple times, we can define a function and call it whenever needed.

Why Use Functions?

✔️ Code Reusability
✔️ Improved Readability
✔️ Better Organization
✔️ Easy Debugging

Basic Function Structure

```
js

function functionName(parameters) {
    // Code to execute
    return result;
}
```

---

2. Function Declaration vs Function Expression

JavaScript provides two main ways to define functions:

Function Declaration (Named Function)

A function declaration defines a function with a name and can be called before its definition due to **hoisting**.

```
js

function greet(name) {
    return `Hello, ${name}!`;
}
```

```
console.log(greet("Alice")); // Output: Hello, Alice!
```

📌 **Hoisting:** Function declarations are hoisted, meaning they can be called before their actual definition.

Function Expression (Anonymous Function)

A function expression assigns a function to a variable. These are **not hoisted**.

```
js

const greet = function(name) {
    return `Hello, ${name}!`;
};

console.log(greet("Bob")); // Output: Hello, Bob!
```

📌 **Key Difference:** Unlike function declarations, function expressions **must** be defined before calling them.

---

3. Arrow Functions (ES6)

Arrow functions provide a concise syntax for writing functions.

Syntax of Arrow Functions

```
js

const functionName = (parameters) => {
    // Function body
    return result;
};
```

Example:

```
js

const square = (num) => {
    return num * num;
};

console.log(square(4)); // Output: 16
```

Implicit Return (Shorter Syntax)

If a function has only one expression, the `return` keyword and `{}` can be omitted.

```
js

const multiply = (a, b) => a * b;
console.log(multiply(3, 5)); // Output: 15
```

📌 **Arrow functions don't have their own `this` context**, which makes them useful in certain situations like event handlers and callbacks.

---

4. Understanding the `return` Statement and Function Parameters

Function Parameters and Arguments

Functions can take parameters (input values) and return a result.

```
js

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

console.log(add(5, 10)); // Output: 15
```

📌 Default Parameters

```
js

function greet(name = "Guest") {
    return `Hello, ${name}!`;
}

console.log(greet());       // Output: Hello, Guest!
console.log(greet("Alice")); // Output: Hello, Alice!
```

---

5. Function Scope and Closures

Function Scope

Scope defines where a variable is accessible.

- **Global Scope** – Accessible everywhere.
- **Local (Function) Scope** – Accessible only within the function.

```
js

let globalVar = "I am global";

function myFunction() {
    let localVar = "I am local";
    console.log(globalVar); // Accessible
}

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

📌 **Arrow functions do not have their own `this`, so they inherit it from their parent scope.**

---

6. Closures – A Key Concept in JavaScript

A **closure** is a function that remembers the variables from its outer scope even after the outer function has finished executing.

Example:

```
js

function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log(`Outer: ${outerVariable}, Inner: ${innerVariable}`);
    };
}

const newFunction = outerFunction("Hello");
newFunction("World");
// Output: Outer: Hello, Inner: World
```

📌 **Why are Closures Important?** ✔️ Allows private variables
✔️ Useful in callbacks and event handlers
✔️ Essential for JavaScript frameworks and libraries

Summary of Key Concepts

- **Function Declaration** – Hoisted, can be called before definition.
- **Function Expression** – Not hoisted, must be defined before calling.
- **Arrow Functions** – Shorter syntax, no own `this`.
- **Return Statement** – Functions return values using `return`.
- **Function Scope** – Local and global scopes.
- **Closures** – Functions remember variables from their outer scope.

With this, you now have a strong understanding of JavaScript functions. Let me know if you want to explore any topic further! 🚀

Here's a detailed document covering **Arrays and Objects** in JavaScript.

Arrays and Objects – Working with Data in JavaScript

In JavaScript, **arrays** and **objects** are the most commonly used data structures for storing and managing data efficiently. Arrays store ordered collections, while objects hold key-value pairs.

---

1. Creating and Manipulating Arrays

An **array** is an ordered list of values. You can store multiple values in a single variable.

Declaring an Array

```
js

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

Manipulating Arrays

JavaScript provides various built-in methods to work with arrays.

1️⃣ Adding and Removing Elements

Method	Description
`` `push(item)` ``	Adds an item to the **end** of an array
`` `pop()` ``	Removes the last item from an array
`` `unshift(item)` ``	Adds an item to the **beginning** of an array
`` `shift()` ``	Removes the first item from an array

```
js


let colors = ["Red", "Green", "Blue"];

// Adding elements
colors.push("Yellow");  // ["Red", "Green", "Blue", "Yellow"]
colors.unshift("Purple"); // ["Purple", "Red", "Green", "Blue", "Yellow"]

// Removing elements
colors.pop();  // ["Purple", "Red", "Green", "Blue"]
colors.shift(); // ["Red", "Green", "Blue"]

console.log(colors);
```

2️⃣ Iterating Over Arrays

You can loop through an array using different methods:

✅ Using `` `for` `` Loop

```
js


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

✅ Using `` `for...of` `` Loop

```
js


for (let fruit of fruits) {
    console.log(fruit);
}
```

✅ Using `` `forEach()` ``

```
js


fruits.forEach((fruit, index) => {
    console.log(`${index}: ${fruit}`);
});
```

3️⃣ Higher-Order Array Methods

JavaScript provides powerful methods to manipulate arrays:

Method	Description
`map()`	Returns a **new** array after applying a function to each element
`filter()`	Returns a **new** array with elements that satisfy a condition
`reduce()`	Reduces an array to a **single value**

✅ Using `map()`

```
js

let numbers = [1, 2, 3, 4];
let squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // Output: [1, 4, 9, 16]
```

✅ Using `filter()`

```
js

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

✅ Using `reduce()`

```
js

let sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // Output: 10
```

---

2. Understanding Objects

An **object** is a collection of key-value pairs, where keys are strings (or symbols) and values can be any data type.

Creating an Object

```
js

let person = {
    name: "Alice",
    age: 25,
    city: "New York"
};
console.log(person);
```

Accessing and Modifying Object Properties

✅ Dot Notation

```
js

console.log(person.name); // Output: Alice
person.age = 26;
```

✅ Bracket Notation

```
js

console.log(person["city"]); // Output: New York
person["age"] = 27;
```

✅ Adding and Deleting Properties

```
js

person.country = "USA"; // Adding new property
delete person.city; // Deleting a property

console.log(person);
```

---

3. Object Methods and Prototypes

Objects can also have **methods** (functions stored as properties).

Defining a Method

```
js

let user = {
    name: "John",
    greet: function() {
        return `Hello, my name is ${this.name}`;
    }
};

console.log(user.greet()); // Output: Hello, my name is John
```

Prototypes in JavaScript

Every JavaScript object has a **prototype**, which is an object from which it inherits properties and methods.

✅ Example of Prototypal Inheritance

```
js

function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.greet = function() {
    return `Hi, I am ${this.name}`;
};

let p1 = new Person("Mike", 30);
console.log(p1.greet()); // Output: Hi, I am Mike
```

---

4. Iterating Over Objects

You can loop through object properties using `for...in` loop.

Using `for...in` Loop

```
js

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

---

5. Summary of Key Concepts

Concept	Description
Arrays	Ordered collection of elements
Objects	Collection of key-value pairs
Array Methods	`push()`, `pop()`, `shift()`, `unshift()`, `map()`, `filter()`, `reduce()`
Object Methods	Functions stored as properties inside objects
Prototypes	Inheritance mechanism in JavaScript
Loops	`for`, `for...of`, `for...in`

Now you have a solid understanding of **arrays and objects** in JavaScript! Let me know if you want to dive deeper into any topic. 🚀

Here's a well-structured document covering **Asynchronous JavaScript** in detail.

Asynchronous JavaScript – Handling Time-Sensitive Code

JavaScript is **single-threaded**, meaning it executes one task at a time. However, some operations, like API requests, timers, and file reading, take time. Instead of blocking the entire program, JavaScript uses **asynchronous programming** to handle these tasks efficiently.

---

1. What is Asynchronous Programming and Why is it Important?

Synchronous Code: Executes **line by line** and waits for each task to complete before moving to the next one.

```
js

console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
```

💡 **Output:**

```
arduino

Task 1
Task 2
```

```
Task 3
```

Asynchronous Code: Doesn't block execution. It allows other tasks to run while waiting for an operation to complete.

```
js

console.log("Task 1");

setTimeout(() => {
    console.log("Task 2 (delayed)");
}, 2000);

console.log("Task 3");
```

💡 **Output:**

```
arduino

Task 1
Task 3
Task 2 (delayed)
```

⏳ **Even though `Task 2` takes time, `Task 3` executes immediately!** This makes JavaScript efficient for handling tasks like API calls, file operations, and database queries.

---

2. Using `setTimeout` and `setInterval`

JavaScript provides built-in functions to schedule tasks.

🕒 `setTimeout()`

Executes a function **after a delay** (in milliseconds).

```
js

console.log("Before delay");

setTimeout(() => {
    console.log("Executed after 3 seconds");
}, 3000);

console.log("After scheduling setTimeout");
```

💡 **Output:**

```
pgsql

Before delay
After scheduling setTimeout
Executed after 3 seconds
```

⏰ `setInterval()`

Repeats a function **at regular intervals**.

```
js


let count = 1;
let interval = setInterval(() => {
    console.log(`Repeating task ${count}`);
    count++;
    if (count > 5) clearInterval(interval); // Stop after 5 iterations
}, 1000);
```

💡 **Output:** (Logs every second)

```
nginx


Repeating task 1
Repeating task 2
Repeating task 3
Repeating task 4
Repeating task 5
```

⚠️ **Use `clearInterval(intervalID)` to stop execution!**

---

3. Introduction to Callbacks and Callback Hell

A **callback function** is a function **passed as an argument** to another function and executed later.

Basic Callback Example

```
js


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

function askQuestion() {
    console.log("How are you?");
}

greet("Alice", askQuestion);
```

💡 **Output:**

```
sql


Hello, Alice
How are you?
```

Callback Hell (Nested Callbacks)

When multiple callbacks are nested, it becomes difficult to manage.

```
js


setTimeout(() => {
    console.log("Task 1");
    setTimeout(() => {
        console.log("Task 2");
        setTimeout(() => {
```

```
            console.log("Task 3");
        }, 1000);
    }, 1000);
}, 1000);
```

💡 **This is known as "callback hell" or "pyramid of doom"** due to difficult-to-read nested code.

Solution? Use Promises!

---

4. Promises: Handling Asynchronous Code Better

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

Creating a Promise

```
js

const myPromise = new Promise((resolve, reject) => {
    let success = true;
    setTimeout(() => {
        if (success) {
            resolve("Task Completed ✅");
        } else {
            reject("Task Failed ❌");
        }
    }, 2000);
});

// Consuming the Promise
myPromise
    .then(result => console.log(result))  // Runs if resolved
    .catch(error => console.log(error))   // Runs if rejected
    .finally(() => console.log("Operation Finished"));
```

💡 **Output (after 2 sec):**

```
arduino

Task Completed ✅
Operation Finished
```

Chaining Promises

```
js

function step1() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Step 1 completed");
            resolve();
        }, 1000);
    });
}

function step2() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Step 2 completed");
            resolve();
        }, 1000);
```

```
    });
}

step1()
    .then(step2)
    .then(() => console.log("All steps completed!"));
```

💡 **Promises avoid callback hell and improve readability!**

---

5. `async` and `await` – Writing Cleaner Asynchronous Code

Using `async/await` makes asynchronous code look more like synchronous code.

Declaring an `async` Function

An `async` function **always returns a Promise**.

```
js

async function greet() {
    return "Hello, Async!";
}

greet().then(console.log); // Output: Hello, Async!
```

Using `await` to Handle Promises

`await` pauses execution until the Promise resolves.

```
js

function delayedMessage() {
    return new Promise(resolve => {
        setTimeout(() => resolve("Message after 2 seconds"), 2000);
    });
}

async function showMessage() {
    console.log("Waiting...");
    let message = await delayedMessage();
    console.log(message);
}

showMessage();
```

💡 **Output:**

```
mathematica

Waiting...
Message after 2 seconds
```

Error Handling with `try...catch`

```
js

async function fetchData() {
    try {
```

```
        let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log("Error fetching data:", error);
    }
}

fetchData();
```

📌 **`async/await` makes working with asynchronous operations much easier and readable!**

---

6. Summary of Key Concepts

Concept	Description
Synchronous vs Asynchronous	Sync code executes line by line, async code runs in the background
setTimeout()	Delays execution of a function
setInterval()	Executes a function at fixed intervals
Callbacks	Functions passed into another function for execution later
Callback Hell	Too many nested callbacks make code hard to read
Promises	Objects representing success or failure of async tasks
.then() / .catch()	Used to handle Promise resolution and errors
async / await	Modern syntax for handling asynchronous code cleanly

---

Now you have a **solid understanding of Asynchronous JavaScript**! 🚀
Let me know if you want to explore anything in more detail. 😊

Here's a well-structured document covering **JavaScript and the DOM** in detail.

JavaScript and the DOM – Interacting with the Browser

The **Document Object Model (DOM)** is a representation of the web page that allows JavaScript to dynamically manipulate HTML and CSS.

---

1. What is the DOM (Document Object Model)?

- The DOM is a **tree-like structure** representing an HTML document.
- Each element in the document is represented as a **node**.
- JavaScript can **access, modify, add, or remove** elements and attributes in the DOM.

Example DOM Structure

```
html


<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
</head>
<body>
    <h1 id="title">Hello, DOM!</h1>
```

```
    <p class="description">This is a paragraph.</p>
</body>
</html>
```

🔍 The DOM representation of the above HTML looks like this:

```
css


Document
└── html
    ├── head
    └── body
        ├── h1 (#title)
        ├── p (.description)
```

---

2. Accessing HTML Elements in JavaScript

JavaScript provides multiple methods to **select elements** from the DOM.

🟢 `document.getElementById()` – Selects an element by ID

```
js


let heading = document.getElementById("title");
console.log(heading);  // <h1 id="title">Hello, DOM!</h1>
```

🔵 `document.querySelector()` – Selects the first matching element

```
js


let paragraph = document.querySelector(".description");
console.log(paragraph); // <p class="description">This is a paragraph.</p>
```

🟣 `document.querySelectorAll()` – Selects all matching elements (returns a NodeList)

```
js


let allParagraphs = document.querySelectorAll("p");
console.log(allParagraphs);  // NodeList of all <p> elements
```

---

3. Modifying Content in the DOM

We can change the text, HTML, and values of elements dynamically.

📌 Changing Content

- 🔹 `innerHTML` – Changes the HTML inside an element

```js
document.getElementById("title").innerHTML = "Welcome to JavaScript!";
```

- **`textContent` – Changes only the text (ignores HTML tags)**

```js
document.querySelector(".description").textContent = "This is updated text!";
```

- **`value` – Changes input field values**

```html
<input type="text" id="username" value="John">
```

```js
document.getElementById("username").value = "Alice";
```

---

4. Manipulating Styles Dynamically

We can modify CSS properties of elements using JavaScript.

🟠 Changing an Element's Style

```js
let title = document.getElementById("title");
title.style.color = "blue";
title.style.fontSize = "24px";
title.style.backgroundColor = "yellow";
```

🔵 Adding/Removing CSS Classes

```css
.hidden {
    display: none;
}
```

```js
let para = document.querySelector(".description");

// Add a class
para.classList.add("hidden");

// Remove a class
para.classList.remove("hidden");
```

```
// Toggle a class (add if absent, remove if present)
para.classList.toggle("hidden");
```

---

5. Adding and Removing Elements in the DOM

JavaScript allows us to dynamically create, append, and remove elements.

🟢 `appendChild()` – Add a New Element

```
js

let newPara = document.createElement("p");
newPara.textContent = "This is a new paragraph!";
document.body.appendChild(newPara);
```

🔴 `removeChild()` – Remove an Element

```
js

let parent = document.body;
let child = document.getElementById("title");
parent.removeChild(child);
```

🟡 `insertBefore()` – Insert an Element Before Another

```
js

let reference = document.querySelector(".description");
document.body.insertBefore(newPara, reference);
```

---

6. Event Listeners – Responding to User Actions

JavaScript can make pages interactive by responding to user events like clicks, keypresses, and mouse movements.

🖱️ Adding a Click Event

```
js

document.getElementById("title").addEventListener("click", function() {
    alert("Title Clicked!");
});
```

⌨️ Key Press Event

```
js

document.addEventListener("keydown", function(event) {
    console.log("Key Pressed:", event.key);
});
```

7. Summary of Key Concepts

Concept	Description
DOM	The structure of an HTML document represented as a tree
getElementById()	Selects an element by its ID
querySelector()	Selects the first matching element
querySelectorAll()	Selects all matching elements (returns NodeList)
innerHTML	Modifies the HTML inside an element
textContent	Modifies only the text (ignores HTML tags)
style	Changes CSS properties dynamically
appendChild()	Adds a new child element to a parent
removeChild()	Removes a child element from a parent
insertBefore()	Inserts an element before another
addEventListener()	Attaches an event to an element

Now you have a **strong understanding of JavaScript and the DOM!** 🚀
Let me know if you need more details or practice exercises! 😊

Here's a well-structured document covering **Event Handling in JavaScript** in detail.

Event Handling – Making Web Pages Interactive

JavaScript **events** allow developers to create interactive and dynamic web pages by responding to user actions like clicks, key presses, form submissions, and more.

1. Introduction to Events in JavaScript

Events are actions or occurrences that happen in the browser. JavaScript can detect and respond to these events.

🟢 Common Events in JavaScript

Event Type	Description
click	Fired when an element is clicked
mouseover / mouseout	Triggered when the mouse enters/leaves an element
keydown / keyup	Fired when a key is pressed/released
submit	Triggered when a form is submitted
change	Fires when an input field value changes
focus / blur	Triggered when an input field gains/loses focus
scroll	Fires when a user scrolls the page

2. Event Listeners – Attaching Events Dynamically

JavaScript provides the `addEventListener()` method to attach event listeners.

🔵 Syntax of `addEventListener()`

```
js

element.addEventListener("event", function);
```

- `element`: The HTML element to attach the event to.
- `event`: The event type (e.g., `"click"`, `"keydown"`).
- `function`: The function to execute when the event occurs.

📌 Example: Handling a Click Event

```
html

<button id="myButton">Click Me</button>
<script>
    document.getElementById("myButton").addEventListener("click", function() {
        alert("Button Clicked!");
    });
</script>
```

📌 Example: Handling a Key Press

```
html

<input type="text" id="textInput" placeholder="Type something">
<script>
    document.getElementById("textInput").addEventListener("keydown", function(event) {
        console.log("Key Pressed:", event.key);
    });
</script>
```

---

3. Understanding Event Bubbling and Capturing

🔴 Event Bubbling (Default Behavior)

- Events **bubble up** from the target element to its ancestors.
- If you click a button inside a `<div>`, the event first fires on the button, then the `<div>`, and then `document`.

📌 Example of Event Bubbling

```
html

<div id="parent">
    <button id="child">Click Me</button>
</div>

<script>
    document.getElementById("parent").addEventListener("click", function() {
```

```
        console.log("Parent clicked!");
    });

    document.getElementById("child").addEventListener("click", function() {
        console.log("Child clicked!");
    });
</script>
```

Output on clicking the button:

```
nginx


Child clicked!
Parent clicked!
```

➡️ The event starts at the button and **bubbles up** to the parent.

🟠 Event Capturing (Trickling Down)

- The event is first captured at the root and then moves down to the target.

📌 Example of Event Capturing

```
js


document.getElementById("parent").addEventListener("click", function() {
    console.log("Parent clicked!");
}, true);  // Passing "true" enables capturing phase
```

📌 **Event order:** `"Parent clicked!"` first, then `"Child clicked!"`.

---

4. Preventing Default Behavior & Stopping Propagation

🔹 `event.preventDefault()` – Stops Default Actions

Some elements have default behaviors (e.g., form submission, link navigation). We can prevent these using `preventDefault()`.

📌 Example: Preventing Form Submission

```
html


<form id="myForm">
    <input type="text" placeholder="Enter text">
    <button type="submit">Submit</button>
</form>

<script>
    document.getElementById("myForm").addEventListener("submit", function(event) {
        event.preventDefault();  // Stops form from submitting
        alert("Form submission prevented!");
    });
</script>
```

🔹 `event.stopPropagation()` – Stops Event Bubbling

If you don't want an event to bubble up, use `stopPropagation()`.

📌 Example: Stopping Event Bubbling

```
html

<div id="outerDiv" style="padding: 20px; background-color: lightblue;">
    <button id="innerButton">Click Me</button>
</div>

<script>
    document.getElementById("outerDiv").addEventListener("click", function() {
        alert("Outer DIV Clicked!");
    });

    document.getElementById("innerButton").addEventListener("click", function(event) {
        event.stopPropagation(); // Stops bubbling
        alert("Button Clicked!");
    });
</script>
```

Now, clicking the button **only triggers "Button Clicked!"**, without bubbling up to `outerDiv`.

---

5. Summary of Key Concepts

Concept	Description
Event	An action occurring in the browser (click, keypress, etc.)
addEventListener()	Attaches an event listener to an element
Event Bubbling	Event starts at the target and moves up the DOM tree
Event Capturing	Event starts from the root and moves down to the target
preventDefault()	Stops the default behavior of an element
stopPropagation()	Stops an event from bubbling up

🎯 Now you have a solid understanding of event handling in JavaScript!
Let me know if you need more practice exercises. 🚀

Here's a well-structured document covering **Object-Oriented JavaScript (OOP)** in detail.

Object-Oriented JavaScript – Mastering Objects and Classes

JavaScript is a prototype-based language but supports **Object-Oriented Programming (OOP)** through **prototypes** and **ES6 classes**. OOP helps in writing modular, reusable, and scalable code.

---

1. Introduction to Object-Oriented Programming (OOP) in JavaScript

OOP is a programming paradigm based on the concept of **objects**, which contain properties (variables) and methods (functions).

🟢 Key OOP Concepts

1. **Encapsulation** – Wrapping data and methods inside objects.
2. **Abstraction** – Hiding implementation details from the user.
3. **Inheritance** – Creating new classes from existing ones.
4. **Polymorphism** – Using a method in different ways.

---

2. Defining Classes and Objects in JavaScript

🔹 Creating Objects using Object Literals

```
js

const person = {
    name: "Alice",
    age: 25,
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet(); // Output: Hello, my name is Alice
```

🔹 Creating Objects using a Constructor Function

Before ES6, objects were created using constructor functions.

```
js

function Person(name, age) {
    this.name = name;
    this.age = age;

    this.greet = function() {
        console.log(`Hi, I am ${this.name}`);
    };
}

const person1 = new Person("Bob", 30);
person1.greet(); // Output: Hi, I am Bob
```

---

3. Using Classes in JavaScript (ES6 Syntax)

🔹 Defining a Class

ES6 introduced the `class` keyword for defining classes.

```
js

class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hi, I am ${this.name}`);
```

```
    }
}

const person2 = new Person("Charlie", 28);
person2.greet(); // Output: Hi, I am Charlie
```

🔹 Key Points about Classes

- `constructor()`: Initializes object properties.
- Methods are defined inside the class, no need for `function` keyword.
- `this` refers to the instance of the class.

---

4. The `this` Keyword in JavaScript

🔹 Understanding `this` in Different Contexts

- In a **method**, `this` refers to the object.
- In a **function**, `this` refers to the global object (`window` in browsers).
- In an **arrow function**, `this` is inherited from the surrounding scope.

📌 Example: Using `this` in Methods

```
js

class Car {
    constructor(brand) {
        this.brand = brand;
    }

    showBrand() {
        console.log(`This car is a ${this.brand}`);
    }
}

const myCar = new Car("Toyota");
myCar.showBrand(); // Output: This car is a Toyota
```

---

5. Inheritance and the Prototype Chain

Inheritance allows a class to derive properties and methods from another class.

🔹 Using `extends` for Inheritance

```
js

class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        console.log("Animal sound...");
    }
}

class Dog extends Animal {
    constructor(name, breed) {
```

```
        super(name); // Calls parent class constructor
        this.breed = breed;
    }

    bark() {
        console.log(`${this.name} barks!`);
    }
}

const myDog = new Dog("Buddy", "Labrador");
myDog.makeSound(); // Output: Animal sound...
myDog.bark();      // Output: Buddy barks!
```

🔹 **Key Points about Inheritance**

- `extends` is used to inherit from another class.
- `super()` calls the parent class constructor.
- Child classes can have their own methods.

---

6. Prototypes in JavaScript

JavaScript uses **prototypes** for inheritance instead of traditional class-based inheritance.

🔹 How Prototypes Work

Every JavaScript object has a `__proto__` property, pointing to its prototype.

```
js


function Person(name) {
    this.name = name;
}

// Adding a method using prototype
Person.prototype.greet = function() {
    console.log(`Hello, I am ${this.name}`);
};

const person3 = new Person("Eve");
person3.greet(); // Output: Hello, I am Eve
```

🔹 **Prototype Chain:** If a property/method is not found in an object, JavaScript looks for it in the prototype.

---

7. Polymorphism and Encapsulation

🔹 Polymorphism – Overriding Methods

Polymorphism allows a method to be redefined in a child class.

```
js


class Shape {
    area() {
        return "Area formula not defined";
    }
}
```

```
class Circle extends Shape {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius * this.radius;
    }
}

const myCircle = new Circle(5);
console.log(myCircle.area()); // Output: 78.54
```

🔹 Encapsulation – Keeping Data Private

Encapsulation restricts direct access to object properties.

Using `#` for Private Properties (ES6+)

```
js

class BankAccount {
    #balance; // Private property

    constructor(amount) {
        this.#balance = amount;
    }

    deposit(amount) {
        this.#balance += amount;
    }

    getBalance() {
        return this.#balance;
    }
}

const myAccount = new BankAccount(1000);
myAccount.deposit(500);
console.log(myAccount.getBalance()); // Output: 1500
// console.log(myAccount.#balance);  // ❌ Error: Private field '#balance' must be declared in an e
```

---

8. Summary of Key Concepts

Concept	Description
Classes	Defines blueprints for creating objects
Objects	Instances of classes containing properties and methods
Constructor	Initializes object properties when creating an instance
this Keyword	Refers to the current instance of the object
Inheritance	Allows child classes to inherit from parent classes
Prototypes	Enables objects to inherit properties from other objects
Polymorphism	Allows overriding methods in child classes
Encapsulation	Restricts direct access to object properties

🎯 **Now you have a strong understanding of Object-Oriented JavaScript!**
Let me know if you need exercises or further explanations. 🚀

Here's a well-structured document covering **Advanced JavaScript Concepts** in detail.

Advanced JavaScript Concepts – Deep Dive

JavaScript has several advanced concepts that are crucial for writing efficient, optimized, and modular code. These concepts help in managing scope, handling asynchronous tasks, organizing code, and debugging effectively.

---

1. Closures and Lexical Scoping

🔹 Understanding Lexical Scoping

JavaScript follows **lexical scoping**, meaning a function's scope is determined by its location in the code.

```
js

function outer() {
    let outerVariable = "I am from outer";

    function inner() {
        console.log(outerVariable); // Can access outerVariable
    }

    inner();
}

outer(); // Output: I am from outer
```

- The **inner function** can access the **outer function's** variables.
- This happens because of **lexical scoping**.

🔹 What is a Closure?

A **closure** is when a function retains access to its parent function's variables even after the parent function has executed.

```
js

function counter() {
    let count = 0; // Private variable

    return function() {
        count++;
        console.log(count);
    };
}

const increment = counter();
increment(); // Output: 1
increment(); // Output: 2
```

💡 Why use closures?

- **Data encapsulation** – Variables inside closures are private.
- **State persistence** – Retains values between function calls.

---

2. Understanding `this` Keyword in JavaScript

What is `this`?

- `this` refers to the object that is executing the function.
- It behaves **differently** in different contexts.

`this` in Global Scope

```
js

console.log(this); // In browser: Window object
```

`this` Inside an Object Method

```
js

const obj = {
    name: "Alice",
    greet: function() {
        console.log(this.name);
    }
};

obj.greet(); // Output: Alice
```

- **Inside a method,** `this` refers to the object itself.

`this` in a Regular Function

```
js

function showThis() {
    console.log(this);
}

showThis(); // Output: Window (global object)
```

- **In a function,** `this` refers to the **global object (window in browsers)**.

---

3. JavaScript `this` Binding: `call()`, `apply()`, and `bind()`

Using `call()`

The `call()` method invokes a function with a specified `this` value.

```
js

const person = {
    name: "Bob"
```

```
};

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

greet.call(person); // Output: Hello, Bob
```

🔹 Using `apply()`

`apply()` works like `call()`, but it takes an array of arguments.

```
js

function introduce(age, city) {
    console.log(`I am ${this.name}, ${age} years old from ${city}`);
}

introduce.apply(person, [25, "New York"]);
// Output: I am Bob, 25 years old from New York
```

🔹 Using `bind()`

`bind()` returns a **new function** with `this` permanently set.

```
js

const greetPerson = greet.bind(person);
greetPerson(); // Output: Hello, Bob
```

📌 Key Differences

Method	When to Use
`call()`	Invoke function immediately, passing `this` explicitly.
`apply()`	Same as `call()`, but arguments are in an array.
`bind()`	Returns a new function with `this` permanently set.

---

4. JavaScript Modules – `import` and `export`

🔹 Why Use Modules?

- Keeps code **organized** and **maintainable**.
- Prevents **global namespace pollution**.
- Makes code **reusable** across multiple files.

🔹 Exporting and Importing in JavaScript Modules

🔹 **Named Export** (Can export multiple values)

📌 **File:** `math.js`

```
js

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

```
}

export function multiply(a, b) {
    return a * b;
}
```

📌 **File:** `main.js`

```
js

import { add, multiply } from './math.js';

console.log(add(2, 3)); // Output: 5
console.log(multiply(2, 3)); // Output: 6
```

🔹 **Default Export** (Only one export per file)

📌 **File:** `message.js`

```
js

export default function greet(name) {
    return `Hello, ${name}`;
}
```

📌 **File:** `main.js`

```
js

import greet from './message.js';

console.log(greet("Alice")); // Output: Hello, Alice
```

📌 **When to Use**

Export Type	Use Case
Named Export	When exporting multiple values
Default Export	When exporting a single main value

---

5. Error Handling in JavaScript

🔹 Using `try...catch` to Handle Errors

Errors can occur during runtime. Use `try...catch` to prevent crashes.

```
js

try {
    let result = 10 / 0; // No error, but infinite value
    console.log(result);

    let num = someUndefinedVariable; // ❌ This will cause an error
} catch (error) {
    console.log("An error occurred:", error.message);
}
```

📌 **Output**

```
vbnet

An error occurred: someUndefinedVariable is not defined
```

🔹 Using `throw` to Create Custom Errors

```
js

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.log("Error:", error.message);
}
```

🔹 **Output**: `Error: Cannot divide by zero`

🔹 `finally` Block

The `finally` block runs **always**, regardless of whether an error occurred or not.

```
js

try {
    console.log("Trying...");
    throw new Error("Oops!");
} catch (error) {
    console.log("Caught error:", error.message);
} finally {
    console.log("This always runs");
}
```

Output

```
go

Trying...
Caught error: Oops!
This always runs
```

---

6. Summary of Key Concepts

Concept	Description
Closures	Functions retain access to variables in their lexical scope.
Lexical Scope	Functions can access variables from their parent scope.
this Keyword	Refers to the object calling the function.
call, apply, bind	Methods to set `this` explicitly.
Modules (`import/export`)	Helps in organizing and reusing code across files.

Concept	Description
Error Handling (`try...catch`)	Prevents crashes and improves debugging.

🎯 **Now you have a strong grasp of Advanced JavaScript Concepts!**
Let me know if you need exercises or explanations for any specific topic. 🚀

Here's a comprehensive document covering **Modern JavaScript (ES6+) Features** in detail.

JavaScript ES6+ Features – Modern JavaScript Syntax

ES6 (ECMAScript 2015) and later versions introduced powerful features that make JavaScript more readable, efficient, and developer-friendly. Understanding these concepts is essential for writing clean and modern JavaScript code.

---

1. Destructuring Assignment for Objects and Arrays

🔹 Destructuring Objects

Destructuring allows extracting values from objects easily.

```
js

const person = {
    name: "Alice",
    age: 25,
    city: "New York"
};

// Without destructuring
console.log(person.name, person.age);

// With destructuring
const { name, age } = person;
console.log(name, age); // Output: Alice 25

// Renaming variables
const { city: location } = person;
console.log(location); // Output: New York
```

🔹 Destructuring Arrays

Extract elements from an array using destructuring.

```
js

const numbers = [10, 20, 30];

// Without destructuring
console.log(numbers[0], numbers[1]);

// With destructuring
const [first, second] = numbers;
console.log(first, second); // Output: 10 20

// Skipping elements
const [, , third] = numbers;
console.log(third); // Output: 30
```

🔹 Use Cases

- Extract values from API responses.
- Assign multiple variables efficiently.
- Improve code readability.

---

2. Template Literals and String Interpolation

🔹 What Are Template Literals?

Template literals allow embedding expressions inside strings using **backticks** (` `).

```
js

const name = "Alice";
const age = 25;

// Traditional way
console.log("My name is " + name + " and I am " + age + " years old.");

// Using template literals
console.log(`My name is ${name} and I am ${age} years old.`);
```

🔹 Advantages

- **Multi-line strings** without needing `\n`.
- **Expression interpolation** using `${}`.
- **Easier to read and maintain**.

---

3. Default Parameters, Rest Parameters, and Spread Syntax

🔹 Default Parameters

Assign default values to function parameters.

```
js

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

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

🔹 Rest Parameters (`...args`)

Collects multiple arguments into an array.

```
js

function sum(...numbers) {
    return numbers.reduce((acc, num) => acc + num, 0);
}
```

```
console.log(sum(1, 2, 3, 4)); // Output: 10
```

🔹 Spread Syntax (`...`)

Used to **expand** arrays or objects.

```
js

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

console.log(newNumbers); // Output: [1, 2, 3, 4, 5]
```

🔹 Use Cases

- Cloning objects/arrays.
- Passing function arguments dynamically.
- Merging arrays easily.

```
js

const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3 };

const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // Output: { a: 1, b: 2, c: 3 }
```

---

4. Introduction to `Map`, `Set`, `WeakMap`, and `WeakSet`

🔹 `Map` – Key-Value Pairs

- Stores key-value pairs, **preserves insertion order**.
- Keys can be **any type** (not just strings).

```
js

const userMap = new Map();
userMap.set("name", "Alice");
userMap.set("age", 25);

console.log(userMap.get("name")); // Output: Alice
console.log(userMap.has("age")); // Output: true

userMap.delete("age");
console.log(userMap.size); // Output: 1
```

🔹 `Set` – Unique Values

- Stores **only unique** values.

```
js

const uniqueNumbers = new Set([1, 2, 2, 3, 4, 4]);
console.log(uniqueNumbers); // Output: Set { 1, 2, 3, 4 }
```

```
uniqueNumbers.add(5);
console.log(uniqueNumbers.has(3)); // Output: true
uniqueNumbers.delete(2);
console.log(uniqueNumbers); // Output: Set { 1, 3, 4, 5 }
```

🔹 `WeakMap` and `WeakSet`

- Stores **weak references** (used for memory optimization).
- Keys must be **objects** (not primitives).

```
js

let obj = { id: 1 };
const weakMap = new WeakMap();
weakMap.set(obj, "Some data");

console.log(weakMap.get(obj)); // Output: Some data

// If `obj` is garbage collected, WeakMap removes it automatically.
```

---

5. Summary of Key Concepts

Feature	Description
Destructuring	Extracts values from objects and arrays easily.
Template Literals	Uses backticks (`) for multi-line strings and embedding expressions.
Default Parameters	Assigns default values to function parameters.
Rest Parameters	Gathers multiple arguments into an array (`...args`).
Spread Syntax	Expands arrays/objects (`...`).
Map	Stores key-value pairs with any key type.
Set	Stores only unique values.
WeakMap	Stores weak references to objects, preventing memory leaks.

🎯 You now have a strong grasp of Modern JavaScript (ES6+) Features!
Let me know if you need practical exercises or further explanations. 🚀