**JAVASCRIPT NOTES**

**1. Overview of JavaScript**

JavaScript is a **high-level**, **interpreted** scripting language that runs primarily in web browsers to create **interactive effects** on webpages. As mentioned, it was created by **Brendan Eich** in 1995 at **Netscape Communications**, and initially named **LiveScript** before being renamed to **JavaScript** due to its association with Java, which was becoming popular at the time.

**2. JavaScript Features**

* **Event-driven**: JavaScript responds to user interactions such as clicks, keystrokes, and mouse movements, making it essential for creating dynamic web pages.
* **Functional programming**: JavaScript supports functional programming features like higher-order functions, closures, and anonymous functions.
* **Imperative programming**: JavaScript also supports imperative styles, where you can control the flow of the program with statements and conditions.
* **Object-oriented**: It supports object-oriented programming features, where you can create objects and classes to organize code and manage complexity.
* **Asynchronous**: JavaScript allows asynchronous execution, which is crucial for tasks like API calls and handling large data without freezing the interface. This is achieved with callbacks, promises, and async/await.

**3. JavaScript Use Cases**

* **Frontend Web Development**: JavaScript is a cornerstone of web development, used to create dynamic and interactive websites. Along with HTML and CSS, it enables functionalities like form validation, interactive maps, and animations.
* **Backend Development**: With the rise of **Node.js**, JavaScript can also be used on the server side to build scalable backend applications.
* **Mobile Apps**: Frameworks like **React Native** allow developers to build mobile applications using JavaScript.
* **Game Development**: JavaScript, with libraries like **Phaser.js**, is used to create games directly in the browser.

**4. JavaScript vs Java: Execution & Use Cases**

Though **JavaScript** and **Java** share a similar name, they are distinct languages with different use cases and execution environments.

* **JavaScript**: Interpreted language that runs in web browsers. JavaScript code is typically executed by a browser's JavaScript engine (like V8 in Chrome). It is mainly used for frontend development but can also be used for backend development with Node.js.
* **Java**: A compiled, object-oriented programming language that runs on the **JVM (Java Virtual Machine)**. Java is widely used for building large-scale, enterprise-level applications, backend services, and Android applications. Java code must be compiled into bytecode and then executed on the JVM.

**JavaScript Execution Flow:**

JavaScript is **interpreted** by the browser (or by Node.js for server-side applications). When a webpage is loaded, the browser loads the HTML, CSS, and JavaScript files. The browser then executes the JavaScript code, interacting with the DOM (Document Object Model) to modify the page dynamically.

**5. How JavaScript Runs**

* **Browser**: JavaScript is executed in the browser using an engine (like **V8** in Chrome, **SpiderMonkey** in Firefox, or **JavaScriptCore** in Safari).
  + The browser's JavaScript engine reads and interprets the JavaScript code line by line.
  + JavaScript can interact with HTML and CSS through the DOM to modify webpage content dynamically.
* **Node.js**: JavaScript can also run on the server with **Node.js**, a runtime that uses the **V8 engine** from Google Chrome. Node.js enables developers to use JavaScript for both frontend and backend development, creating full-stack applications with a single language.

**Ways to Run JavaScript:**

1. **In the browser**:
   * Directly in the browser's **Developer Tools Console**.
   * Inside HTML <script> tags.
   * Through external JavaScript files linked with <script src="file.js"></script>.
2. **In Node.js**:
   * By executing JavaScript files in the command line using the node command: node file.js.
3. **Online Editors**:
   * Websites like **CodePen**, **JSFiddle**, and **JSBin** allow you to write and execute JavaScript directly in a browser without any setup.

**6. JavaScript Syntax Overview**

* **Variables**: Declared using var, let, or const.
* **Functions**: Can be declared using function declarations, expressions, or arrow functions.
* **Objects & Arrays**: JavaScript uses objects (key-value pairs) and arrays (ordered lists) for data storage and manipulation.

**7. JavaScript Ownership**

JavaScript is **open-source**, and the specification is maintained by **Ecma International**, under the ECMAScript standard. Every modern browser has its own engine to execute JavaScript code, but they all follow the ECMAScript standard, ensuring a level of consistency across platforms.

In summary, JavaScript is a **highly versatile** and **dynamic** language, integral to web development, enabling developers to create responsive, interactive web applications. While it shares similarities in name with Java, it operates in a completely different manner and serves different purposes, from frontend development to server-side programming.

Here’s a list of commonly used **Node.js commands** in JavaScript, along with their use and explanations:

**1. node**

* **Use**: This is the most fundamental command used to run JavaScript files in Node.js.
* **Example**:

node app.js

* **Explanation**: It executes a JavaScript file (in this case, app.js) using the Node.js runtime.

**2. npm init**

* **Use**: Initializes a new Node.js project by creating a package.json file.
* **Example**:

npm init

or with default values:

npm init -y

* **Explanation**: This command sets up your project and manages its dependencies. It prompts you to enter details about your project like name, version, description, entry point, etc.

**3. npm install or npm i**

* **Use**: Installs dependencies listed in the package.json file.
* **Example**:

npm install

or for a specific package:

npm install express

* **Explanation**: This command fetches the required libraries and installs them in the node\_modules folder. You can also use npm i as a shortcut.

**4. npm install --save**

* **Use**: Installs a package and adds it to the dependencies section of package.json.
* **Example**:

npm install express --save

* **Explanation**: The --save flag tells npm to add the installed package to the dependencies section of package.json. (Note: In modern versions of npm, --save is the default.)

**5. npm install --save-dev**

* **Use**: Installs a package as a development dependency (not for production).
* **Example**:

npm install mocha --save-dev

* **Explanation**: This installs the package as a development-only dependency (typically used for testing tools).

**6. npm uninstall**

* **Use**: Removes a package from your project.
* **Example**:

npm uninstall express

* **Explanation**: Removes the specified package from node\_modules and updates the package.json file.

**7. npm update**

* **Use**: Updates all the packages to their latest versions according to the version ranges in package.json.
* **Example**:

npm update

* **Explanation**: This will update your installed packages to the latest versions as specified by the version constraints in package.json.

**8. npm list**

* **Use**: Displays installed packages in the current project.
* **Example**:

npm list

* **Explanation**: Lists all the installed packages, showing the hierarchy of dependencies.

**9. npm run**

* **Use**: Executes a script defined in the scripts section of package.json.
* **Example**:

npm run start

* **Explanation**: If start is defined in package.json, this command will run it. You can define custom scripts in the scripts section to automate tasks.

**10. npm start**

* **Use**: Runs the start script from package.json.
* **Example**:

npm start

* **Explanation**: It executes the start script defined in the package.json. By default, it is used to start the application (like starting a server).

**11. npm test**

* **Use**: Runs the test script defined in package.json.
* **Example**:

npm test

* **Explanation**: This command runs your test suite. You can define custom testing scripts in package.json.

**12. npm outdated**

* **Use**: Checks if any of the installed packages are outdated.
* **Example**:

npm outdated

* **Explanation**: Lists all the packages that are outdated, showing the current version, wanted version, and the latest version.

**13. npm cache clean**

* **Use**: Clears the npm cache.
* **Example**:

npm cache clean --force

* **Explanation**: This command clears the npm cache, which can sometimes become corrupted. The --force flag is required in newer versions of npm.

**14. npm audit**

* **Use**: Scans your dependencies for vulnerabilities.
* **Example**:

npm audit

* **Explanation**: This command checks your installed packages for known security vulnerabilities and provides recommendations for updates.

**15. npm audit fix**

* **Use**: Automatically fixes vulnerabilities found by npm audit.
* **Example**:

npm audit fix

* **Explanation**: It attempts to update dependencies to versions that do not have known security vulnerabilities.

**16. node -v or node --version**

* **Use**: Displays the installed version of Node.js.
* **Example**:

node -v

* **Explanation**: This command shows the version of Node.js currently installed on your system.

**17. npm -v or npm --version**

* **Use**: Displays the installed version of npm (Node.js package manager).
* **Example**:

npm -v

* **Explanation**: This command shows the version of npm currently installed on your system.

**18. npx**

* **Use**: Executes binaries from node\_modules/.bin or remote packages without installing them globally.
* **Example**:

npx create-react-app my-app

* **Explanation**: This command runs a package directly from the command line without needing to install it globally.

**19. node --inspect**

* **Use**: Launches a Node.js process in debugging mode.
* **Example**:

node --inspect app.js

* **Explanation**: Starts Node.js with debugging enabled, allowing you to debug your code using Chrome DevTools.

**20. node --harmony**

* **Use**: Enables experimental ECMAScript features in Node.js.
* **Example**:

node --harmony app.js

* **Explanation**: This flag allows you to use experimental ECMAScript features that aren't fully supported yet in the current version of Node.js.
* **What is JavaScript?**

JavaScript is a high-level, interpreted programming language that is primarily used for building interactive and dynamic websites. It was initially developed to add interactivity to web pages but has since evolved into a versatile language used for both client-side (in the browser) and server-side (on the backend) development. It is a core technology of the web, alongside HTML and CSS, and is supported by all modern web browsers.

Key points about JavaScript:

* **Interpreted Language**: JavaScript code is executed line by line by the browser or a JavaScript runtime like Node.js.
* **Event-driven**: JavaScript can react to user events, such as clicks or form submissions, making web pages interactive.
* **Versatile**: Used in both frontend and backend development, thanks to the introduction of Node.js.
* **Asynchronous**: JavaScript can perform non-blocking operations like API requests, using promises or async/await syntax.

**Setting Up JavaScript**

**In the Browser:**

1. **Include JavaScript in HTML**:
   * JavaScript can be included directly in an HTML file using the <script> tag.
   * You can place JavaScript code between the opening and closing <script> tags within the HTML file, or you can link to an external .js file using the src attribute.

Example of inline JavaScript:

<script>

console.log("Hello, World!");

</script>

Example of linking an external JavaScript file:

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

1. **Browser Developer Tools**:
   * You can open the browser’s developer tools (usually by pressing F12) to access the console for testing and debugging JavaScript code.

**In Node.js:**

1. **Install Node.js**:
   * Download and install Node.js from [nodejs.org](https://nodejs.org).
   * This will also install the Node.js package manager (npm) to manage dependencies.
2. **Running JavaScript in Node.js**:
   * Once Node.js is installed, you can execute JavaScript files via the command line using the node command:

node app.js

1. **Using npm**:
   * You can use npm to install various JavaScript packages and libraries, allowing you to build full-fledged applications.

Example:

npm init -y

npm install express

**JavaScript in the Browser vs Node.js**

**1. JavaScript in the Browser:**

* **Environment**: Runs within a web browser (e.g., Chrome, Firefox, Safari).
* **APIs**: Has access to the **Document Object Model (DOM)** and **Browser APIs**, enabling interaction with web pages, handling events, manipulating HTML, and responding to user inputs.
* **Usage**: Primarily used for **front-end development** to enhance the user experience with dynamic content, animations, and interactive features.
* **Global Object**: The global object is window, which gives you access to browser-specific functionality like window.alert(), window.location, etc.
* **Limitations**: Limited to client-side activities, such as DOM manipulation, handling events, or making AJAX calls.

Example of Browser-Specific Code:

javascript

document.getElementById("myButton").addEventListener("click", function() {

alert("Button clicked!");

});

**2. JavaScript in Node.js:**

* **Environment**: Runs on the server, using a runtime environment called Node.js. It is built on the V8 JavaScript engine (same engine used by Chrome).
* **APIs**: Does not have access to the DOM. Instead, it provides APIs for **file system operations**, **networking**, and **creating HTTP servers**. It is ideal for building server-side applications.
* **Usage**: Used for **backend development**, such as building web servers, APIs, handling database operations, and managing server-side logic.
* **Global Object**: The global object is global, and you have access to built-in Node.js modules such as fs (for file system operations), http (for creating HTTP servers), and path (for file path utilities).
* **Asynchronous Nature**: Node.js is known for being asynchronous and non-blocking, using an event-driven architecture and callback mechanisms for handling I/O operations.

Example of Node.js Code:

const http = require('http');

const server = http.createServer((req, res) => {

res.write('Hello, World!');

res.end();

});

server.listen(3000, () => {

console.log('Server is running on port 3000');

});

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Feature | JavaScript in the Browser | JavaScript in Node.js |
| Environment | Runs inside the browser | Runs on the server (Node.js runtime) |
| APIs Available | DOM manipulation, Browser APIs (e.g., localStorage) | File system, HTTP server, networking APIs |
| Global Object | window | global |
| Use Case | Frontend development (interactive websites) | Backend development (servers, APIs, databases) |
| Concurrency | Single-threaded (with some async features) | Non-blocking, asynchronous I/O operations |
| Libraries/Modules | Limited (primarily for frontend work) | Extensive (for building full-stack applications) |
| Example Use | Manipulating DOM elements, handling user events | Building APIs, handling file system tasks |

Both environments allow JavaScript to be a versatile language, but each is tailored to different aspects of development—browsers for the client-side and Node.js for server-side operations.

* **Detailed Explanation of JavaScript Console Methods**

This document provides a detailed explanation of various JavaScript console methods. These methods are essential for debugging, logging, and tracking performance during development.

# console.log()

Usage: console.log('Log message');  
Purpose: Outputs general information or messages to the console. Commonly used for debugging and providing feedback.  
Example: console.log('This is a log message');

# console.info()

Usage: console.info('Info message');  
Purpose: Logs informational messages, often displayed in blue font or different formatting in some browsers.  
Example: console.info('Info message displayed');

# console.warn()

Usage: console.warn('Warning message');  
Purpose: Logs a warning message, usually displayed with a yellow warning sign, alerting about potential issues.  
Example: console.warn('This is a warning message');

# console.error()

Usage: console.error('Error message');  
Purpose: Logs an error message, typically in red, with stack traces for debugging.  
Example: console.error('This is an error message');

# console.debug()

Usage: console.debug('Debugging message');  
Purpose: Logs debugging messages. Behaves like console.log() but may be filtered out by default in some browsers.  
Example: console.debug('Debugging info here');

# console.table()

Usage: console.table(['row1', 'row2', 'row3']);  
Purpose: Displays tabular data in a structured format, useful for visualizing arrays or objects.  
Example: const students = [{ name: 'John', age: 25 }, { name: 'Jane', age: 22 }];  
console.table(students);

# console.group()

Usage: console.group('Group name');  
Purpose: Starts a new group, indenting subsequent log messages until console.groupEnd() is called. Useful for organizing logs.  
Example: console.group('Group 1');  
console.log('Message inside group');  
console.groupEnd();

# console.groupCollapsed()

Usage: console.groupCollapsed('Collapsed group name');  
Purpose: Starts a new group in a collapsed state, making it hidden initially.  
Example: console.groupCollapsed('Collapsed Group');  
console.log('This is inside the collapsed group');  
console.groupEnd();

# console.groupEnd()

Usage: console.groupEnd();  
Purpose: Ends the most recent group started by console.group() or console.groupCollapsed().  
Example: console.group('Nested Group');  
console.log('Message inside nested group');  
console.groupEnd();

# console.time()

Usage: console.time('Timer label');  
Purpose: Starts a timer with a specific label to measure the performance of an operation.  
Example: console.time('Timer1');  
// Some time-consuming operation  
console.timeEnd('Timer1');

# console.timeEnd()

Usage: console.timeEnd('Timer label');  
Purpose: Stops the timer and logs the elapsed time in milliseconds.  
Example: console.time('Test');  
for (let i = 0; i < 1000; i++) {  
 // Some operation  
}  
console.timeEnd('Test');

# console.trace()

Usage: console.trace('Trace label');  
Purpose: Outputs a stack trace to the console, showing the sequence of function calls leading to the point in the code.  
Example: function test() {  
 console.trace('Trace in test function');  
}  
test();

# console.count()

Usage: console.count('Count label');  
Purpose: Logs the number of times it has been called with the same label.  
Example: console.count('Count');  
console.count('Count');  
console.count('Count'); // Outputs: Count: 3

# console.countReset()

Usage: console.countReset('Count label');  
Purpose: Resets the count for a specific label back to zero.  
Example: console.count('ResetCount');  
console.count('ResetCount');  
console.countReset('ResetCount');  
console.count('ResetCount'); // Outputs: ResetCount: 1

# console.assert()

Usage: console.assert(condition, 'Assertion failed message');  
Purpose: Logs an error message if the condition is falsy. Does nothing if true.  
Example: console.assert(2 + 2 === 4, 'Math error'); // Nothing happens  
console.assert(2 + 2 === 5, 'Math error'); // Logs: Assertion failed: Math error

# console.clear()

Usage: console.clear();  
Purpose: Clears the console, removing all previous log entries.  
Example: console.clear(); // Clears the console screen

**JavaScript Scopes: Detailed Explanation**

In JavaScript, **scope** refers to the context in which a variable is declared and where it can be accessed. It determines the **visibility** and **lifespan** of variables, functions, and objects within different parts of your program. Understanding scope is essential for managing variable access, avoiding conflicts, and keeping your code organized.

**Types of Scope in JavaScript**

There are primarily **three types of scope** in JavaScript:

1. **Global Scope**
2. **Function Scope**
3. **Block Scope**

Additionally, understanding concepts like **Lexical Scope** and **Closure** is crucial for deeper insights into how scope works.

**1. Global Scope**

* **Definition**: A variable declared outside of any function or block is said to have a global scope.
* **Characteristics**:
  + Accessible from anywhere in the code after its declaration.
  + It belongs to the global execution context, which is the outermost context in JavaScript (usually window in browsers or global in Node.js).
* **Example**:

let globalVar = "I'm a global variable";

function printGlobal() {

console.log(globalVar); // Can access the global variable

}

printGlobal(); // Output: I'm a global variable

console.log(globalVar); // Output: I'm a global variable

* **Notes**:
  + Be careful when declaring global variables, as they can be accessed and modified from any part of the program, potentially leading to bugs or unintended side effects.

**2. Function Scope**

* **Definition**: A variable declared within a function is accessible only within that function and any nested functions.
* **Characteristics**:
  + Variables declared with var inside a function are only available within that function and are not accessible outside.
  + let and const also follow function scope inside a function but are block-scoped (explained below).
  + **Hoisting** occurs for variables declared with var inside a function. The declaration is hoisted to the top, but the initialization stays in its original position.
* **Example**:

function myFunction() {

var functionVar = "I'm a function-scoped variable";

console.log(functionVar); // Output: I'm a function-scoped variable

}

myFunction();

console.log(functionVar); // ReferenceError: functionVar is not defined

* **Notes**:
  + Variables declared with var inside a function are confined to the function scope.
  + Variables declared with let and const inside a function are also confined to that function, but they are **not hoisted** in the same way var is.

**3. Block Scope**

* **Definition**: Block scope refers to the scope of variables declared inside a block (a block is any code within {} like inside loops, conditionals, or functions).
* **Characteristics**:
  + Variables declared with let or const inside a block are only accessible within that block.
  + var does **not** have block scope. It only respects function scope.
* **Example**:

if (true) {

let blockScopedVar = "I'm block-scoped";

console.log(blockScopedVar); // Output: I'm block-scoped

}

console.log(blockScopedVar); // ReferenceError: blockScopedVar is not defined

* **Example with var** (Notice the difference with let):

if (true) {

var functionScopedVar = "I'm function-scoped, not block-scoped";

}

console.log(functionScopedVar); // Output: I'm function-scoped, not block-scoped

* **Notes**:
  + **let** and **const** are block-scoped.
  + **var** is function-scoped, even when declared inside a block (e.g., inside loops or conditionals).

**4. Lexical Scope**

* **Definition**: Lexical scope refers to the fact that the scope of a variable is determined by the position of the code where the variable is declared.
* **Characteristics**:
  + A function’s scope is determined by where it was defined, not where it is called.
  + Inner functions have access to variables declared in their outer (enclosing) functions.
* **Example**:

function outer() {

let outerVar = "I'm an outer variable";

function inner() {

console.log(outerVar); // The inner function can access outerVar

}

inner();

}

outer(); // Output: I'm an outer variable

* **Closure**:
  + Lexical scoping creates **closures**. A closure is a function that "remembers" its lexical scope even when executed outside that scope.
* **Example of Closure**:

function outer() {

let counter = 0;

function increment() {

counter++;

console.log(counter);

}

return increment;

}

const incrementCounter = outer();

incrementCounter(); // Output: 1

incrementCounter(); // Output: 2

**5. Hoisting**

* **Definition**: Hoisting is the behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code has been executed.
* **Characteristics**:
  + **Function declarations** are hoisted along with their definition (i.e., both the declaration and the function body are moved to the top).
  + **Variables declared with var** are hoisted only with their declaration (not initialization).
  + **let** and **const** declarations are hoisted, but not initialized, and they cannot be accessed before their declaration due to the Temporal Dead Zone.
* **Example with var**:

console.log(x); // Output: undefined (due to hoisting)

var x = 5;

console.log(x); // Output: 5

* **Example with let and const**:

console.log(y); // ReferenceError: Cannot access 'y' before initialization

let y = 10;

**6. Global Object**

* In **browsers**, the global object is the window object. Any variable declared in the global scope is a property of the window object.
* **Example**:

var globalVar = "I am global!";

console.log(window.globalVar); // Output: I am global!

* In **Node.js**, the global object is the global object.
* **Example**:

global.globalVar = "I am global in Node.js";

console.log(global.globalVar); // Output: I am global in Node.js

**7. Temporal Dead Zone (TDZ)**

* **Definition**: The Temporal Dead Zone refers to the period between entering the scope and the actual declaration where a variable is not accessible.
* **Characteristics**:
  + Applies to variables declared with let and const.
  + Accessing a variable before its declaration in this period will throw a **ReferenceError**.
* **Example**:

console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 5;

**Conclusion:**

Understanding **JavaScript scope** is critical for avoiding bugs, managing variable access, and writing cleaner code. The main types of scope are **global**, **function**, and **block scope**. Additional concepts like **lexical scope**, **hoisting**, and **closures** help in understanding how variables and functions interact in different contexts. By managing scope properly, you can avoid unwanted behavior and ensure that variables are accessed only where they are intended to be.

* **Variable Declarations: var, let, const**

**var:**

* **Scope**:
  + **Function-scoped**: If declared inside a function, accessible only within that function.
  + **Global-scoped**: If declared outside a function, accessible throughout the entire code.
* **Hoisting**: The var declaration is moved to the top, but its value remains where it was initialized.
  + Example:

console.log(name); // undefined

var name = "John"; // 'name' is hoisted, but value is not assigned yet.

* **Re-declaration**: You can re-declare the same variable in the same scope.
  + Example:

var name = "John";

var name = "Jane"; // No error

**let:**

* **Scope**:
  + **Block-scoped**: Only accessible within the block {} in which it’s declared.
* **Hoisting**: let is hoisted but cannot be accessed before its declaration (known as **Temporal Dead Zone**).
  + Example:

console.log(age); // ReferenceError

let age = 25; // 'age' cannot be accessed before its declaration

* **Re-declaration**: Cannot re-declare the same variable in the same block.
  + Example:

let age = 25;

let age = 30; // SyntaxError: Identifier 'age' has already been declared

**const:**

* **Scope**:
  + **Block-scoped**: Like let, const is accessible only within its block.
* **Hoisting**: const is hoisted, but cannot be accessed before declaration.
  + Example:

console.log(country); // ReferenceError

const country = "USA"; // 'country' cannot be accessed before its declaration

* **Re-declaration & Assignment**:
  + **Re-declaration**: Cannot re-declare a const variable.
  + **Reassignment**: Once a value is assigned, it cannot be reassigned.
  + For objects/arrays: You cannot reassign the object/array itself, but can modify its properties/elements.
  + Example:

const country = "USA";

country = "Canada"; // TypeError: Assignment to constant variable.

const person = { name: "Alice" };

person.name = "Bob"; // Allowed

person = { name: "Charlie" }; // TypeError: Assignment to constant variable

**2. JavaScript Variable Types (Data Types)**

* **Primitive Data Types:**
* **String**: Text enclosed in quotes.

let name = "Alice";

* **Number**: Numeric values, including integers and decimals.

let age = 25;

* **Boolean**: Represents true or false.

let isActive = true;

* **Null**: Represents an intentional "empty" value.

let user = null;

* **Undefined**: A declared variable with no value assigned.

let address;

console.log(address); // undefined

* **Symbol** (ES6): A unique identifier.

let sym = Symbol('description');

* **BigInt** (ES11): Large integers that can't fit in Number.

let bigNumber = 1234567890123456789012345678901234567890n;

* **Non-Primitive Data Types:**
* **Object**: A collection of key-value pairs.

let person = { name: "Alice", age: 25 };

* **Array**: An ordered list of values.

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

**3. Best Practices and Rules for Using Variables**

* **Use let** for variables that will change (e.g., counters or iterators).

let count = 0;

* **Use const** for values that shouldn't change (e.g., constants or function references).

const pi = 3.14;

* **Avoid var** in modern JavaScript because of scoping and hoisting issues.
* **Re-declaring Variables**:
  + var allows re-declaration, which can lead to bugs.
  + let and const don’t allow re-declaration in the same scope, which helps avoid errors.
* **Immutability**:
  + Use const when the value should not change to ensure it isn’t reassigned by mistake.

const MAX\_COUNT = 100;

**4. Summary of Key Points**

* **var**: Function-scoped, can be re-declared, but use with caution due to potential issues with hoisting.
* **let**: Block-scoped, hoisted, but cannot be accessed before its declaration (TDZ).
* **const**: Block-scoped, cannot be reassigned, best for constants or references that shouldn’t change.
* **Primitive types** are immutable (e.g., String, Number), while **non-primitive types** (like objects and arrays) are mutable.
* **The (this) Keyword in JavaScript**

In JavaScript, the behavior of the this keyword is dynamic, meaning its value depends on the **execution context**—how and where a function is called. Here's a breakdown of how this behaves in different situations:

**1. In Global Scope**

* **Definition**: When this is used in the global scope (outside of any function or object), it refers to the global object.
* **In Browsers**: In the browser, this refers to the window object.
* **In Node.js**: In Node.js, this refers to the global object.
* **Example**:

console.log(this); // Logs the global object (window in browsers)

**2. Inside a Regular Function**

* **Definition**: When this is used inside a regular function, it refers to the object that **calls** the function. In **non-strict mode**, this will refer to the global object (i.e., window in browsers). In **strict mode**, this will be undefined.
* **Example** (non-strict mode):

function regularFunction() {

console.log(this); // In non-strict mode, logs the global object (window)

}

regularFunction();

* **Example** (strict mode):

'use strict';

function strictFunction() {

console.log(this); // Logs undefined in strict mode

}

strictFunction();

**3. Inside Arrow Functions**

* **Definition**: Arrow functions **do not have their own this context**. Instead, they inherit this from the **lexical context** in which they are defined. This means that inside an arrow function, this refers to the value of this from the surrounding scope where the arrow function is created.
* **Example**:

const arrowFunction = () => {

console.log(this); // Logs the lexical context of `this`, not the global object

};

arrowFunction();

**4. In Methods of Objects**

* **Definition**: When a function is called as a method of an object, this refers to the object itself. This is one of the most common uses of this.
* **Example**:

const person = {

name: "John",

greet: function() {

console.log(this.name); // `this` refers to the `person` object

}

};

person.greet(); // Output: John

**5. With call(), apply(), and bind()**

* **Definition**: You can explicitly set the value of this using the call(), apply(), and bind() methods. These methods allow you to call a function with a specific this context.
  + call() and apply() immediately invoke the function with the provided context.
  + bind() creates a new function with the specified this context, but does not invoke it immediately.
* **Example with call()**:

function sayHello() {

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

}

const person = { name: "Alice" };

sayHello.call(person); // Output: Hello, Alice

* **Example with bind()**:

const boundHello = sayHello.bind(person);

boundHello(); // Output: Hello, Alice

**6. Best Practices for Scoping in JavaScript**

1. **Use let and const instead of var**:
   * **Why**: var is function-scoped and can lead to issues like **hoisting**, where variable declarations are moved to the top of the function or global scope. let and const are block-scoped, reducing the likelihood of conflicts and mistakes.
   * **Recommendation**: Always prefer let and const over var for variable declarations.

**Example**:

let x = 5; // block-scoped

const y = 10; // block-scoped

1. **Avoid Global Variables**:
   * **Why**: Global variables are accessible from anywhere, which increases the risk of conflicts and unintended side effects. They can also be accidentally overwritten.
   * **Recommendation**: Use local variables within functions or modules. If you must use a global variable, consider using a module pattern or immediately invoked function expression (IIFE) to limit scope.

**Example**:

function example() {

let localVar = "I am local"; // Only accessible inside the function

}

1. **Use Closures for Data Encapsulation**:
   * **Why**: Closures allow you to create private variables and functions within a function. This helps to protect internal data and logic from external access, ensuring better data integrity and security.
   * **Recommendation**: Use closures to manage state privately, especially in object-oriented or modular code.

**Example**:

function createCounter() {

let count = 0;

return {

increment: function() {

count++;

console.log(count);

},

decrement: function() {

count--;

console.log(count);

}

};

}

const counter = createCounter();

counter.increment(); // Output: 1

counter.increment(); // Output: 2

counter.decrement(); // Output: 1

**Conclusion**

The this keyword in JavaScript is a powerful but often confusing concept, especially since its behavior depends on the execution context. Understanding how this works in global scope, functions, and arrow functions is essential for effective JavaScript programming. Following best practices like using let and const, avoiding global variables, and using closures for encapsulation can help you write cleaner, safer, and more maintainable code.

* **"use strict" in JavaScript**

**What it Does:**

* **Enables Strict Mode:** Activates a stricter version of JavaScript.
* **Improves Error Handling:** Detects and throws errors for issues like undeclared variables.
* **Enhances Security:** Disallows dangerous features like the with statement and accidental global variables.
* **Boosts Performance:** JavaScript engines optimize code better when strict mode is enabled.
* **Prevents Undefined this:** In strict mode, this is undefined when called without a proper context.

**Key Benefits:**

1. **Prevents Silent Errors:**
   * In non-strict mode, JavaScript silently ignores mistakes (e.g., using undeclared variables).
   * Strict mode throws errors.
   * **Example:**

'use strict';

x = 10; // Error: x is not declared

1. **Security Improvements:**
   * **No with Statement:**

'use strict';

with (Math) { // Error: `with` is not allowed

console.log(sin(0));

}

* + **Prevents Accidental Globals:** Variables must be declared explicitly.

1. **Performance Boost:**
   * Makes it easier for engines to optimize code.
2. **this in Functions:**
   * In strict mode, this is undefined in a function that’s not called with an object.

'use strict';

function showThis() {

console.log(this); // undefined

}

showThis();

* **How to Use "use strict":**
* **For the Entire Script:** Place at the top of the script.

'use strict';

// All code runs in strict mode

* **For a Single Function:** Place inside the function.

function myFunction() {

'use strict';

// Code runs in strict mode here

}

**Conclusion:**

1. "use strict" helps make JavaScript more secure, catches errors earlier, prevents unsafe practices, and can improve performance. It is a good practice to use it to write cleaner and more reliable JavaScript code.

**Naming Conventions in Programming**

**1. Camel Case**

* **Description**: The first word is in lowercase, and each subsequent word starts with an uppercase letter.
* **Example**:

let userName = "Amol";

function getTotalAmount() { }

* **Common Use**:
  + JavaScript variables
  + Function names

**2. Pascal Case**

* **Description**: Similar to camel case but with the first letter of the first word capitalized.
* **Example**:

class UserAccount { }

function GetUserData() { }

* **Common Use**:
  + Class names
  + C# and .NET naming conventions

**3. Snake Case**

* **Description**: All letters are lowercase, and words are separated by underscores.
* **Example**:

user\_name = "Amol"

* **Common Use**:
  + Python variables
  + Python function names

**4. Kebab Case**

* **Description**: Similar to snake case, but words are separated by hyphens instead of underscores.
* **Example**:

user-name="Amol"

* **Common Use**:
  + URLs
  + Filenames
  + JavaScript file naming (e.g., my-component.js)

**5. Uppercase (Screaming Snake Case)**

* **Description**: All letters are uppercase with words separated by underscores.
* **Example**:

const MAX\_SIZE = 100;

* **Common Use**:
  + Constants in programming languages

**6. Dot Notation**

* **Description**: Used to access properties or methods of an object or namespace.
* **Example**:

let user = {

firstName: "Amol",

lastName: "Kadam"

};

console.log(user.firstName);

* **Common Use**:
  + JavaScript objects
  + Accessing properties or methods

**7. Hungarian Notation**

* **Description**: Prefixes are used to indicate the type or intended use of a variable.
* **Example**:

let strName = "Amol";

let intAge = 30;

* **Common Use**:
  + Typing conventions in some older languages (less common today)

**8. Mixed Case**

* **Description**: A combination of uppercase and lowercase letters, typically without a set pattern.
* **Example**:

let userNameIsGood = true;

* **Common Use**:
  + Occasional use
  + Inconsistent naming

**typeof Operator Quirks in JavaScript:**

* The typeof operator is used to check the type of a variable.
  + typeof

number, string, Boolean, undefined

* **Special Cases:**
  + **null:**
    - typeof null → "object" (This is a historical bug in JavaScript.)
    - **Example:**

console.log(typeof null); // "object"

* + **Arrays:**
* typeof [] → "object" (Arrays are technically objects in JavaScript.)
* **Example:**

console.log(typeof []); // "object"

* + **Functions:**
* typeof returns "function" for functions, but they are technically objects.
* **Example:**

function myFunction() {}

console.log(typeof myFunction); // "function"

* + **Objects:**
    - typeof returns "object" for objects (including arrays, null, and plain objects).
    - **Example:**

const obj = {};

console.log(typeof obj); // "object"

* + **Other Primitive Types:**
    - typeof Symbol() → "symbol"
    - typeof BigInt(10) → "bigint"
    - **Examples:**

console.log(typeof Symbol()); // "symbol"

console.log(typeof BigInt(10)); // "bigint"

**Full List of JavaScript and Related Terms Full Forms:**

1. **AJAX** - Asynchronous JavaScript and XML
2. **API** - Application Programming Interface
3. **BOM** - Browser Object Model
4. **CORS** - Cross-Origin Resource Sharing
5. **CSS** - Cascading Style Sheets
6. **DOM** - Document Object Model
7. **ES** - ECMAScript
8. **ES6** - ECMAScript 2015 (6th version of ECMAScript)
9. **ESM** - ECMAScript Modules
10. **ES5** - ECMAScript 5 (5th version of ECMAScript)
11. **HTTPS** - Hypertext Transfer Protocol Secure
12. **HTTP** - Hypertext Transfer Protocol
13. **IDE** - Integrated Development Environment
14. **JSON** - JavaScript Object Notation
15. **JSONP** - JSON with Padding
16. **MVC** - Model-View-Controller
17. **NPM** - Node Package Manager
18. **ORM** - Object-Relational Mapping
19. **PWA** - Progressive Web Application
20. **REST** - Representational State Transfer
21. **SPA** - Single Page Application
22. **SSR** - Server-Side Rendering
23. **UI** - User Interface
24. **UX** - User Experience
25. **XHR** - XMLHttpRequest
26. **URL** - Uniform Resource Locator
27. **WYSIWYG** - What You See Is What You Get
28. **XML** - Extensible Markup Language
29. **JWT** - JSON Web Token
30. **OAuth** - Open Authorization
31. **SaaS** - Software as a Service
32. **IaaS** - Infrastructure as a Service
33. **PaaS** - Platform as a Service
34. **FaaS** - Function as a Service
35. **CDN** - Content Delivery Network
36. **CI/CD** - Continuous Integration / Continuous Delivery
37. **LAMP** - Linux, Apache, MySQL, PHP
38. **MEAN** - MongoDB, Express.js, Angular, Node.js
39. **MERN** - MongoDB, Express.js, React.js, Node.js
40. **AMP** - Accelerated Mobile Pages
41. **TDD** - Test-Driven Development
42. **BDD** - Behavior-Driven Development
43. **SVG** - Scalable Vector Graphics
44. **FSD** - Full-Stack Developer
45. **CRUD** - Create, Read, Update, Delete
46. **CI** - Continuous Integration
47. **CD** - Continuous Deployment
48. **NVM** - Node Version Manager
49. **CLI** - Command Line Interface
50. **VUE** - Virtual DOM User Environment
51. **TS** - TypeScript (A superset of JavaScript)
52. **jQuery** - A fast, small, and feature-rich JavaScript library
53. **Webpack** - A static module bundler for JavaScript applications
54. **Redux** - A predictable state container for JavaScript apps
55. **Bootstrap** - A front-end framework for developing responsive websites
56. **SASS** - Syntactically Awesome Stylesheets
57. **SCSS** - Sassy CSS
58. **GraphQL** - A query language for APIs
59. **Node.js** - JavaScript runtime built on Chrome's V8 engine
60. ECMAScript stands for **European Computer Manufacturers Association Script**.

**1. Implicit Type Conversion (Type Coercion):**

This happens automatically when JavaScript converts one data type to another as required by the operation.

**Examples of Implicit Conversion:**

* **String + Number → String**:  
  When adding a string and a number, JavaScript converts the number to a string and concatenates them.

let result = "5" + 10; // "510"

* **Number + String → String**:  
  If the number is added to a string, the number is converted to a string and then concatenated.

let result = 10 + "5"; // "105"

* **Boolean + Number → Number**:  
  When a boolean is added to a number, the boolean is converted to a number (true becomes 1, and false becomes 0).

let result = true + 5; // 6 (true is converted to 1)

* **String - Number → Number**:  
  When subtracting a string from a number, the string is converted to a number (if possible).

let result = "10" - 5; // 5

**2. Explicit Type Conversion (Manual Conversion):**

JavaScript allows you to manually convert one data type to another using built-in methods.

**A. Converting to String:**

1. **String() Constructor:**
   * Converts any data type to a string.

let str = String(123); // "123"

1. **.toString() Method:**
   * Converts any value to a string.

let num = 123;

let str = num.toString(); // "123"

1. **Template Literals:**
   * Can also be used to convert any value to a string.

let num = 123;

let str = `${num}`; // "123"

**B. Converting to Number:**

1. **Number() Constructor:**
   * Converts values to a number, if possible.

let num = Number("123"); // 123

let num2 = Number("abc"); // NaN (Not a Number)

1. **parseInt() Method:**
   * Converts a string to an integer (parses the first part of the string that is a number).

let num = parseInt("123.45"); // 123

let num2 = parseInt("abc"); // NaN

1. **parseFloat() Method:**
   * Converts a string to a floating-point number (decimal).

let num = parseFloat("123.45"); // 123.45

let num2 = parseFloat("123abc"); // 123

1. **Unary Plus (+):**
   * A shorthand for converting a value to a number.

let num = +"123"; // 123

let num2 = +"abc"; // NaN

**C. Converting to Boolean:**

1. **Boolean() Constructor:**
   * Converts any value to a boolean (true or false).

let bool = Boolean(0); // false

let bool2 = Boolean(1); // true

let bool3 = Boolean(""); // false

let bool4 = Boolean("hello"); // true

1. **Falsy Values:**
   * These are considered **false** when converted to boolean:
     + false, 0, "" (empty string), null, undefined, NaN
   * Everything else is considered **true**.

let bool1 = Boolean(0); // false

let bool2 = Boolean("hello"); // true

let bool3 = Boolean(""); // false

**3. Other Types of Conversions:**

**A. Object to Primitive:**

JavaScript automatically converts objects to primitive values when used in expressions, but you can also control this process by defining the valueOf() or toString() methods on an object.

* **valueOf():**  
  This method is used to return the primitive value of an object. It is called automatically in most cases when objects are used in operations like arithmetic.

let obj = {

valueOf: function() {

return 10;

}

};

let result = obj + 5; // 15

* **toString():**  
  This method is used to return a string representation of an object.

let obj = {

toString: function() {

return "Hello";

}

};

let result = obj + " World"; // "Hello World"

**B. Array to String:**

Arrays are automatically converted to strings when needed (e.g., in concatenation).

let arr = [1, 2, 3];

let str = String(arr); // "1,2,3"

**C. String to Array:**

You can convert a string into an array by using methods like split().

let str = "Hello";

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

**4. Type Conversion with Comparisons:**

JavaScript also performs type coercion in comparison operations, which can lead to surprising results.

* **Loose equality (==)** compares values after type conversion.

let result = "5" == 5; // true (both are coerced to numbers)

* **Strict equality (===)** compares values without any type conversion.

let result = "5" === 5; // false (types are different)

**5. Special Cases:**

* **Null & Undefined:**
  + null is considered **falsy** and can be converted to false.
  + undefined is also **falsy** and converts to false when used in boolean contexts.

let result = Boolean(null); // false

let result2 = Boolean(undefined); // false

* **NaN (Not a Number):**
  + Results from invalid or failed numeric conversion, such as attempting to convert non-numeric strings to numbers.

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

**Summary of Type Conversions:**

1. **String Conversion:** String(), .toString(), ${} (template literals)
2. **Number Conversion:** Number(), parseInt(), parseFloat(), +
3. **Boolean Conversion:** Boolean()
4. **Object to Primitive:** valueOf(), toString()
5. **Array to String:** .toString(), String()

These conversions allow JavaScript to handle operations involving mixed data types in a way that usually makes sense, but can sometimes lead to unexpected results, so it's important to understand the behavior of type coercion.

**1. Arithmetic Operators**

These are used to perform basic arithmetic operations.

* + (Addition)  
  Adds two operands.  
  Example: 5 + 3 = 8
* - (Subtraction)  
  Subtracts the second operand from the first.  
  Example: 5 - 3 = 2
* \* (Multiplication)  
  Multiplies two operands.  
  Example: 5 \* 3 = 15
* / (Division)  
  Divides the first operand by the second.  
  Example: 6 / 3 = 2
* % (Modulus)  
  Returns the remainder when the first operand is divided by the second.  
  Example: 5 % 3 = 2
* ++ (Increment)  
  Increases the value of the operand by 1.  
  Example: let x = 5; x++; (x becomes 6)
* -- (Decrement)  
  Decreases the value of the operand by 1.  
  Example: let x = 5; x--; (x becomes 4)

**2. Assignment Operators**

Used to assign values to variables.

* =  
  Assigns the value of the right operand to the left operand.  
  Example: let a = 5;
* +=  
  Adds the right operand to the left and assigns the result.  
  Example: a += 3; (a becomes 8)
* -=  
  Subtracts the right operand from the left and assigns the result.  
  Example: a -= 2; (a becomes 6)
* \*=  
  Multiplies the left operand by the right and assigns the result.  
  Example: a \*= 2; (a becomes 12)
* /=  
  Divides the left operand by the right and assigns the result.  
  Example: a /= 3; (a becomes 4)
* %=  
  Computes the modulus of the left operand by the right and assigns the result.  
  Example: a %= 3; (a becomes 0)

**3. Comparison Operators**

Used to compare two values.

* == (Equal to)  
  Returns true if the operands are equal (but not necessarily of the same type).  
  Example: '5' == 5 (true)
* === (Strict equal to)  
  Returns true if the operands are equal and of the same type.  
  Example: '5' === 5 (false)
* != (Not equal to)  
  Returns true if the operands are not equal.  
  Example: 5 != 3 (true)
* !== (Strict not equal to)  
  Returns true if the operands are not equal or not of the same type.  
  Example: '5' !== 5 (true)
* > (Greater than)  
  Returns true if the left operand is greater than the right.  
  Example: 5 > 3 (true)
* < (Less than)  
  Returns true if the left operand is less than the right.  
  Example: 3 < 5 (true)
* >= (Greater than or equal to)  
  Returns true if the left operand is greater than or equal to the right.  
  Example: 5 >= 3 (true)
* <= (Less than or equal to)  
  Returns true if the left operand is less than or equal to the right.  
  Example: 3 <= 5 (true)

**4. Logical Operators**

Used to perform logical operations.

* && (Logical AND)  
  Returns true if both operands are true.  
  Example: true && false (false)
* || (Logical OR)  
  Returns true if at least one operand is true.  
  Example: true || false (true)
* ! (Logical NOT)  
  Reverses the logical state of its operand. If the condition is true, it becomes false.  
  Example: !true (false)

**5. Bitwise Operators**

Used to perform bit-level operations.

* & (Bitwise AND)  
  Performs a bitwise AND operation.  
  Example: 5 & 3 (1)
* | (Bitwise OR)  
  Performs a bitwise OR operation.  
  Example: 5 | 3 (7)
* ^ (Bitwise XOR)  
  Performs a bitwise XOR operation.  
  Example: 5 ^ 3 (6)
* ~ (Bitwise NOT)  
  Inverts all the bits of the operand.  
  Example: ~5 (-6)
* << (Left Shift)  
  Shifts the bits to the left by the specified number of positions.  
  Example: 5 << 1 (10)
* >> (Right Shift)  
  Shifts the bits to the right by the specified number of positions.  
  Example: 5 >> 1 (2)

**6. Conditional (Ternary) Operator**

A shorthand for if-else conditions.

* Syntax:  
  condition ? value\_if\_true : value\_if\_false;
* Example:  
  let result = (5 > 3) ? "Yes" : "No";  
  Here, result will be "Yes".

**7. Type Operators**

Used to check types and work with types.

* typeof  
  Returns the type of the operand.  
  Example: typeof "hello" (string)
* instanceof  
  Tests whether an object is an instance of a specific class or type.  
  Example: [] instanceof Array (true)

**8. Spread and Rest Operators**

* ... (Spread/Rest)  
  Used to unpack or collect elements into an array or object.
  + **Spread**: Expands elements of an iterable.  
    Example: let arr = [1, 2]; let newArr = [...arr, 3]; (newArr becomes [1, 2, 3])
  + **Rest**: Collects the remaining arguments into an array.  
    Example: function foo(...args) { console.log(args); } (captures all arguments passed)

**9. Destructuring Assignment**

Used to unpack values from arrays or objects into distinct variables.

* Array Destructuring:  
  Example: let [a, b] = [1, 2]; (a = 1, b = 2)
* Object Destructuring:  
  Example: let {x, y} = {x: 1, y: 2}; (x = 1, y = 2)

**10. Unary Operators**

Operators that operate on a single operand.

* + (Unary plus)  
  Converts the operand to a number.  
  Example: +"5" (5)
* - (Unary minus)  
  Negates the operand.  
  Example: -5 (-5)
* ! (Logical NOT)  
  Converts the operand to boolean and inverts it.  
  Example: !true (false)

**Conclusion**

JavaScript operators are a crucial part of programming. They allow us to perform various actions on data, ranging from mathematical calculations to logical decisions and type conversions. Understanding each operator and when to use it is essential for building effective and efficient JavaScript programs.

**Increment and Decrement Operators in JavaScript**

The **increment** (++) and **decrement** (--) operators are used to increase or decrease the value of a variable by 1, respectively. These operators can be used in two forms: **pre-increment** (or pre-decrement) and **post-increment** (or post-decrement).

Let's break it down:

**1. Increment Operator (++)**

This operator increases the value of the operand by 1.

**Pre-increment (++x)**

* Increments the value of x by 1, then returns the updated value.
* The operand is increased **before** being used in an expression.

let x = 5;

let result = ++x; // x is incremented to 6, then assigned to result

console.log(x); // Output: 6

console.log(result); // Output: 6

**Post-increment (x++)**

* Returns the current value of x first, and then increments it by 1.
* The operand is increased **after** being used in an expression.

let x = 5;

let result = x++; // result gets the value of x (5), then x is incremented to 6

console.log(x); // Output: 6

console.log(result); // Output: 5

**Important Points:**

* **Pre-increment** increases the value first and then uses it.
* **Post-increment** uses the current value and then increases it.
* Both forms increment the value of the variable by 1, but how and when the increment occurs differs.

**2. Decrement Operator (--)**

This operator decreases the value of the operand by 1.

**Pre-decrement (--x)**

* Decreases the value of x by 1, then returns the updated value.
* The operand is decreased **before** being used in an expression.

let x = 5;

let result = --x; // x is decremented to 4, then assigned to result

console.log(x); // Output: 4

console.log(result); // Output: 4

**Post-decrement (x--)**

* Returns the current value of x first, and then decrements it by 1.
* The operand is decreased **after** being used in an expression.

let x = 5;

let result = x--; // result gets the value of x (5), then x is decremented to 4

console.log(x); // Output: 4

console.log(result); // Output: 5

**Important Points:**

* **Pre-decrement** decreases the value first and then uses it.
* **Post-decrement** uses the current value and then decreases it.
* Both forms decrement the value of the variable by 1, but how and when the decrement occurs differs.

**Summary of Behavior:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Type | Effect | Example |
| ++x | Pre-increment | Increment first, then use the value | let x = 5; ++x; console.log(x); (x becomes 6) |
| x++ | Post-increment | Use the value first, then increment | let x = 5; x++; console.log(x); (x becomes 6) |
| --x | Pre-decrement | Decrement first, then use the value | let x = 5; --x; console.log(x); (x becomes 4) |
| x-- | Post-decrement | Use the value first, then decrement | let x = 5; x--; console.log(x); (x becomes 4) |

Both **increment** and **decrement** operators are essential tools in programming for modifying variables and controlling flow, especially in loops or counter-based logic.

**Memory Types in JavaScript**

In JavaScript, memory management is crucial for efficient programming. Understanding the different types of memory, their lifecycle, and how they interact is key for optimizing performance and avoiding memory leaks.

**1. Types of Memory**

1. **Heap Memory**:
   * **Description**: Heap is the region where JavaScript stores objects and functions.
   * **Key Points**:
     + Objects, arrays, and closures are stored in the heap.
     + Memory allocation here is dynamic and can grow as needed.
   * **Example**:

let obj = { name: "Amol", age: 25 }; // Stored in heap memory

1. **Stack Memory**:
   * **Description**: Stack is the region where JavaScript stores primitive values and function calls.
   * **Key Points**:
     + Stores primitive data types like numbers, strings, booleans, etc.
     + Follows a Last In, First Out (LIFO) structure.
     + Memory is automatically cleared when the function call completes.
   * **Example**:

let a = 10; // Stored in stack memory

function greet() {

let message = "Hello"; // Function call and variables in stack

}

**2. Memory Allocation in JavaScript**

* **Primitive Types**:
  + Stored directly in stack memory because they are fixed in size.
  + Examples: string, number, boolean, undefined, null, symbol, and bigint.
* **Non-Primitive Types** (Objects, Arrays, Functions):
  + Stored in heap memory because their size can vary.
  + A reference to the object is stored in stack memory.

**3. Memory Life Cycle**

1. **Allocation**:
   * Happens when you declare variables or create objects.
   * Example:

let num = 42; // Allocates memory for a number

let obj = { name: "Amol" }; // Allocates memory for an object

1. **Usage**:
   * JavaScript uses memory to perform computations or manipulate data.
2. **Deallocation**:
   * Memory is released when it’s no longer needed.
   * Handled automatically by **Garbage Collection** in JavaScript.

**4. Garbage Collection**

JavaScript uses **automatic garbage collection** based on a technique called **Mark-and-Sweep**:

* **Mark Phase**: Identifies which objects are still reachable.
* **Sweep Phase**: Removes objects that are no longer reachable.

**5. Common Memory-Related Concepts**

1. **Memory Leak**:
   * Happens when memory that’s no longer used isn’t released.
   * **Causes**:
     + Unnecessary global variables.
     + Unclosed event listeners.
     + Detached DOM elements.
   * **Avoidance**:
     + Always remove unused event listeners.
     + Avoid excessive global variables.
     + Use tools like Chrome DevTools for debugging memory leaks.
2. **Closures**:
   * Closures retain references to the variables of their parent scope, which can sometimes cause memory leaks if not managed well.
   * Example:

function outer() {

let count = 0;

return function inner() {

console.log(count); // Closure retains reference to `count`

};

}

const closure = outer();

1. **Reference Types**:
   * Circular references can prevent garbage collection.
   * Example:

let obj1 = {};

let obj2 = {};

obj1.ref = obj2;

obj2.ref = obj1; // Circular reference

**6. Memory Optimization Tips**

1. **Use Local Variables**:
   * Scope variables appropriately to avoid polluting global memory.
2. **Manage Large Data**:
   * Avoid storing large data in memory unnecessarily. Use pagination or lazy loading.
3. **Clean Up Resources**:
   * Remove event listeners and clear intervals or timeouts when they are no longer needed.
4. **Avoid Memory Leaks**:
   * Use tools like Chrome DevTools to detect and debug memory issues.
5. **Use WeakMap and WeakSet**:
   * These collections do not prevent garbage collection of their keys.
   * Useful for caching or mapping data without risking memory leaks.

**Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Memory Type | Stored Data | Lifecycle | Example |
| Heap | Objects, arrays, functions | Dynamic (garbage-collected) | let obj = { name: "Amol" }; |
| Stack | Primitive values, function calls | Cleared after scope ends | let num = 42; |

**Essential Tools for Memory Management**

1. **Chrome DevTools**:
   * Analyze heap snapshots.
   * Check for memory leaks.
2. **Node.js Tools**:
   * Use --inspect flag for memory profiling.
   * Example: node --inspect app.js

* **JavaScript Strings**

**1. Creating Strings**

Strings can be created using:

* **Single quotes** (')
* **Double quotes** (")
* **Backticks** (`) for template literals.

let str1 = 'Hello';

let str2 = "World";

let str3 = `Hello World`;

**2. String Properties**

* **length**: Returns the number of characters in the string.

let text = "Hello";

console.log(text.length); // 5

**3. String Methods**

**a. Basic Methods**

* **charAt()**: Returns the character at a specified index.

let text = "Hello";

console.log(text.charAt(0)); // "H"

* **indexOf()**: Returns the index of the first occurrence of a substring.

let text = "Hello World";

console.log(text.indexOf("World")); // 6

* **lastIndexOf()**: Returns the index of the last occurrence of a substring.

let text = "Hello Hello";

console.log(text.lastIndexOf("Hello")); // 6

* **slice()**: Extracts a part of the string based on indices.

let text = "Hello World";

console.log(text.slice(0, 5)); // "Hello"

* **substring()**: Similar to slice(), but does not accept negative indices.

let text = "Hello World";

console.log(text.substring(0, 5)); // "Hello"

* **substr()**: Extracts a part of the string from a specified start index and length.

let text = "Hello World";

console.log(text.substr(0, 5)); // "Hello"

**b. Manipulating Strings**

* **toUpperCase()**: Converts all characters to uppercase.

let text = "hello";

console.log(text.toUpperCase()); // "HELLO"

* **toLowerCase()**: Converts all characters to lowercase.

let text = "HELLO";

console.log(text.toLowerCase()); // "hello"

* **replace()**: Replaces the first match of a substring.

let text = "Hello World";

console.log(text.replace("World", "JavaScript")); // "Hello JavaScript"

* **replaceAll()**: Replaces all matches of a substring.

let text = "Hello Hello";

console.log(text.replaceAll("Hello", "Hi")); // "Hi Hi"

* **concat()**: Joins two or more strings.

let text1 = "Hello";

let text2 = "World";

console.log(text1.concat(" ", text2)); // "Hello World"

* **trim()**: Removes whitespace from both ends of a string.

let text = " Hello World ";

console.log(text.trim()); // "Hello World"

**c. Splitting and Joining**

* **split()**: Splits a string into an array of substrings.

let text = "Hello World";

console.log(text.split(" ")); // ["Hello", "World"]

* **join()**: Joins an array of strings into a single string.

let arr = ["Hello", "World"];

console.log(arr.join(" ")); // "Hello World"

**d. Checking Strings**

* **includes()**: Checks if a substring is present.

let text = "Hello World";

console.log(text.includes("World")); // true

* **startsWith()**: Checks if a string starts with a specified substring.

let text = "Hello World";

console.log(text.startsWith("Hello")); // true

* **endsWith()**: Checks if a string ends with a specified substring.

let text = "Hello World";

console.log(text.endsWith("World")); // true

**e. Template Literals**

Template literals (using backticks) allow embedding expressions within strings.

let name = "John";

let greeting = `Hello, ${name}!`;

console.log(greeting); // "Hello, John!"

**4. Escaping Characters**

Use a backslash (\) to escape special characters like quotes, newlines, etc.

let quote = "He said, \"Hello!\"";

console.log(quote); // "He said, "Hello!""

**5. String Comparison**

Strings are compared lexicographically (dictionary order).

let str1 = "apple";

let str2 = "banana";

console.log(str1 < str2); // true

1. **Other Useful Methods**

* **localeCompare()**: Compares two strings based on local language settings.

let str1 = "apple";

let str2 = "banana";

console.log(str1.localeCompare(str2)); // -1 (apple comes before banana)

**7. String Encoding and Decoding**

* **encodeURIComponent()**: Encodes a URI component (special characters are replaced).

let text = "Hello World!";

console.log(encodeURIComponent(text)); // "Hello%20World%21"

* **decodeURIComponent()**: Decodes an encoded URI component.

let encodedText = "Hello%20World%21";

console.log(decodeURIComponent(encodedText)); // "Hello World!"

**Important Notes**

* **Immutability**: JavaScript strings are immutable, meaning you cannot change a string directly. Any operation that modifies a string returns a new string.
* **Default Behavior**: Methods like indexOf(), slice(), etc., return -1 or an empty string if the substring is not found.
* **Numbers and Math in JavaScript**

**Number Properties:**

1. **Number.MAX\_VALUE**
   * **Description**: Represents the largest possible number in JavaScript. It’s used to indicate the upper bound for numerical values in JavaScript. Any number larger than this will result in Infinity.
   * **Example**: console.log(Number.MAX\_VALUE);
   * **Value**: 1.7976931348623157e+308
2. **Number.MIN\_VALUE**
   * **Description**: Represents the smallest positive number greater than 0 that can be represented in JavaScript. This is not the same as 0; it is the smallest non-zero positive value.
   * **Example**: console.log(Number.MIN\_VALUE);
   * **Value**: 5e-324
3. **Number.NaN**
   * **Description**: Represents a value that is "Not-a-Number". This is typically returned when a mathematical operation fails, such as trying to divide 0 by 0 or parse an invalid string as a number.
   * **Example**: console.log(Number.NaN);
   * **Value**: NaN
4. **Number.NEGATIVE\_INFINITY**
   * **Description**: Represents negative infinity. It is used to indicate that the result of an operation is smaller than the lowest possible number that JavaScript can represent.
   * **Example**: console.log(Number.NEGATIVE\_INFINITY);
   * **Value**: -Infinity
5. **Number.POSITIVE\_INFINITY**
   * **Description**: Represents positive infinity. This value is returned when an operation exceeds the largest number JavaScript can represent.
   * **Example**: console.log(Number.POSITIVE\_INFINITY);
   * **Value**: Infinity
6. **Number.EPSILON**
   * **Description**: Represents the smallest possible difference between two representable numbers. This value is useful for comparisons to avoid issues caused by floating-point rounding errors.
   * **Example**: console.log(Number.EPSILON);
   * **Value**: 2.220446049250313e-16

**Number Methods:**

1. **Number.isFinite(value)**
   * **Description**: Checks whether the provided value is a finite number (i.e., not Infinity, -Infinity, or NaN).
   * **Example**: console.log(Number.isFinite(10));
   * **Value**: true
2. **Number.isInteger(value)**
   * **Description**: Checks if the provided value is an integer (whole number). This method will return false for non-integer numbers or any non-numeric values.
   * **Example**: console.log(Number.isInteger(10.5));
   * **Value**: false
3. **Number.isNaN(value)**
   * **Description**: Determines whether the value is actually NaN (Not a Number). This is a more reliable way to check for NaN than using the == or === operator.
   * **Example**: console.log(Number.isNaN('hello'));
   * **Value**: true
4. **Number.isSafeInteger(value)**
   * **Description**: Checks whether the value is a "safe" integer, meaning it can be represented accurately by JavaScript without losing precision (i.e., it lies between Number.MIN\_SAFE\_INTEGER and Number.MAX\_SAFE\_INTEGER).
   * **Example**: console.log(Number.isSafeInteger(9007199254740992));
   * **Value**: true
5. **Number.parseFloat(value)**
   * **Description**: Converts a string to a floating-point number. If the string starts with a valid number, it will return the number; otherwise, it returns NaN.
   * **Example**: console.log(Number.parseFloat('10.25abc'));
   * **Value**: 10.25
6. **Number.parseInt(value)**
   * **Description**: Converts a string to an integer. It will parse the number until it encounters a non-numeric character.
   * **Example**: console.log(Number.parseInt('10.25abc'));
   * **Value**: 10
7. **Number.toFixed(digits)**
   * **Description**: Formats a number with a specified number of decimal places, rounding if necessary. Returns a string, not a number.
   * **Example**: console.log((123.456).toFixed(2));
   * **Value**: "123.46"
8. **Number.toExponential(digits)**
   * **Description**: Returns a string representing the number in exponential notation. This is useful for displaying very large or very small numbers in a compact form.
   * **Example**: console.log((12345).toExponential(2));
   * **Value**: "1.23e+4"
9. **Number.toPrecision(precision)**
   * **Description**: Returns a string representing the number with a specified number of significant digits. It rounds the number if necessary.
   * **Example**: console.log((123.456).toPrecision(5));
   * **Value**: "123.46"
10. **Number.toString(radix)**
    * **Description**: Converts a number to a string, and optionally allows you to specify the base (radix) for the conversion (e.g., binary, hexadecimal).
    * **Example**: console.log((255).toString(16));
    * **Value**: "ff"

**Math Methods:**

1. **Math.abs(value)**

* **Description: Returns the absolute value of a number.**
* **Example: console.log(Math.abs(-10));**
* **Output: 10**

1. **Math.acos(value)**
   1. **Description: Returns the inverse cosine of a number, in radians.**
   2. **Example: console.log(Math.acos(0));**
   3. **Output: 1.5707963267948966**
2. **Math.asin(value)**

* **Description: Returns the inverse sine of a number, in radians.**
* **Example: console.log(Math.asin(1));**
* **Output: 1.5707963267948966**

1. **Math.atan(value)**

* **Description: Returns the inverse tangent of a number, in radians.**
* **Example: console.log(Math.atan(1));**
* **Output: 0.7853981633974483**

1. **Math.atan2(y, x)**

* **Description: Returns the inverse tangent of the quotient of its arguments, considering the signs of both arguments to determine the correct quadrant.**
* **Example: console.log(Math.atan2(1, 1));**
* **Output: 0.7853981633974483**

1. **Math.ceil(value)**

* **Description: Rounds a number upwards to the nearest integer.**
* **Example: console.log(Math.ceil(4.1));**
* **Output: 5**

1. **Math.cos(value)**

* **Description: Returns the cosine of an angle, specified in radians.**
* **Example: console.log(Math.cos(0));**
* **Output: 1**

1. **Math.exp(value)**

* **Description: Returns Euler's number (e) raised to the power of the given value.**
* **Example: console.log(Math.exp(2));**
* **Output: 7.3890560989306495**

1. **Math.floor(value)**

* **Description: Rounds a number downwards to the nearest integer.**
* **Example: console.log(Math.floor(4.9));**
* **Output: 4**

1. **Math.hypot(value1, value2, ...)**

* **Description: Returns the square root of the sum of squares of its arguments (Pythagorean theorem).**
* **Example: console.log(Math.hypot(3, 4));**
* **Output: 5**

1. **Math.log(value)**

* **Description: Returns the natural logarithm (base e) of a number.**
* **Example: console.log(Math.log(1));**
* **Output: 0**

1. **Math.log10(value)**

* **Description: Returns the base-10 logarithm of a number.**
* **Example: console.log(Math.log10(100));**
* **Output: 2**

1. **Math.log2(value)**

* **Description: Returns the base-2 logarithm of a number.**
* **Example: console.log(Math.log2(8));**
* **Output: 3**

1. **Math.max(value1, value2, ...)**

* **Description: Returns the largest number from the provided values.**
* **Example: console.log(Math.max(1, 5, 3));**
* **Output: 5**

1. **Math.min(value1, value2, ...)**

* **Description: Returns the smallest number from the provided values.**
* **Example: console.log(Math.min(1, 5, 3));**
* **Output: 1**

1. **Math.pow(base, exponent)**

* **Description: Returns the base raised to the power of the exponent.**
* **Example: console.log(Math.pow(2, 3));**
* **Output: 8**

1. **Math.random()**

* **Description: Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).**
* **Example: console.log(Math.random());**
* **Output: A random number like 0.123456789**

1. **Math.round(value)**

* **Description: Rounds a number to the nearest integer.**
* **Example: console.log(Math.round(4.6));**
* **Output: 5**

1. **Math.sin(value)**

* **Description: Returns the sine of an angle, specified in radians.**
* **Example: console.log(Math.sin(0));**
* **Output: 0**

1. **Math.sqrt(value)**

* **Description: Returns the square root of a number.**
* **Example: console.log(Math.sqrt(16));**
* **Output: 4**

**23. Math.tan(value)**

* **Description: Returns the tangent of an angle, specified in radians.**
* **Example: console.log(Math.tan(0));**
* **Output: 0**

**24. Math.trunc(value)**

* **Description: Removes the decimal part and returns the integer part of a number.**
* **Example: console.log(Math.trunc(4.9));**
* **Output: 4**

**Other Conversion Methods:**

1. **Number(value)**
   * **Description**: Converts a value to a number. If the value is already a number, it returns it as is. Otherwise, it converts strings or booleans to a number, returning NaN if conversion is not possible.
   * **Example**: console.log(Number('10'));
   * **Value**: 10
2. **parseFloat(value)**
   * **Description**: Converts a string to a floating-point number. It is useful when you expect a decimal value.
   * **Example**: console.log(parseFloat('10.25'));
   * **Value**: 10.25
3. **parseInt(value)**
   * **Description**: Converts a string to an integer, disregarding any decimals or non-numeric characters after the number.
   * **Example**: console.log(parseInt('10.25'));
   * **Value**: 10

**Rounding Methods:**

1. **Math.round(value)**
   * **Description**: Rounds the number to the nearest integer. If the decimal part is 0.5 or greater, it rounds up; otherwise, it rounds down.
   * **Example**: console.log(Math.round(4.5));
   * **Value**: 5
2. **Math.ceil(value)**
   * **Description**: Rounds a number upwards to the nearest integer, regardless of the decimal part.
   * **Example**: console.log(Math.ceil(4.1));
   * **Value**: 5
3. **Math.floor(value)**
   * **Description**: Rounds a number downwards to the nearest integer, truncating any decimal part.
   * **Example**: console.log(Math.floor(4.9));
   * **Value**: 4

* **JavaScript Date and Time**

In JavaScript, the Date object is fundamental for working with dates and times. It provides a variety of methods to retrieve and manipulate time data. Here's an in-depth guide on how to work with JavaScript's Date object, along with custom formatting options:

**1. Creating a Date Object**

The Date object represents a single point in time. You can create a new Date object in the following ways:

* **Current Date and Time:**

let myDate = new Date();

console.log(myDate); // Current date and time (local time zone)

* **Date with Specific Date and Time:**

let myDate = new Date('2025-01-21T10:00:00');

console.log(myDate); // Specific date and time

* **Date from Timestamp:**

let myDate = new Date(1684680000000); // Milliseconds since Jan 1, 1970

console.log(myDate); // Date based on timestamp

**2. Methods to Get Date and Time Information**

Once you have a Date object, there are various methods to extract information from it.

* **Human-readable String:**

console.log(myDate.toString()); // Returns a human-readable date string

* **ISO 8601 Format:**

console.log(myDate.toISOString()); // Converts to ISO 8601 format (YYYY-MM-DDTHH:mm:ss.sssZ)

* **Date Only String:**

console.log(myDate.toDateString()); // Date portion in a readable format (e.g., "Mon Jan 21 2025")

* **Localized Date String (based on browser's language and locale):**

console.log(myDate.toLocaleDateString()); // Localized date format

* **Localized Time String (based on browser's locale):**

console.log(myDate.toLocaleTimeString()); // Localized time format

* **Extract Year, Month, Day, Hour, Minute, Second, Millisecond:**

console.log(myDate.getFullYear()); // Year (2025)

console.log(myDate.getMonth()); // Month (0-11, where 0 = January)

console.log(myDate.getDate()); // Day of the month (1-31)

console.log(myDate.getDay()); // Day of the week (0-6, where 0 = Sunday)

console.log(myDate.getHours()); // Hour (0-23)

console.log(myDate.getMinutes()); // Minutes (0-59)

console.log(myDate.getSeconds()); // Seconds (0-59)

console.log(myDate.getMilliseconds()); // Milliseconds (0-999)

* **Get the Time in Milliseconds Since Jan 1, 1970 (Unix Timestamp):**

console.log(myDate.getTime()); // Timestamp in milliseconds

* **UTC Methods (Returns Coordinated Universal Time):**

console.log(myDate.getUTCFullYear()); // UTC year

console.log(myDate.getUTCMonth()); // UTC month (0-11)

console.log(myDate.getUTCDay()); // UTC day of the week (0-6)

console.log(myDate.getUTCHours()); // UTC hours (0-23)

console.log(myDate.getUTCMinutes()); // UTC minutes (0-59)

console.log(myDate.getUTCSeconds()); // UTC seconds (0-59)

**3. Methods to Set Date and Time**

You can also modify the components of a Date object using the following setter methods:

* **Set Year:**

myDate.setFullYear(2023); // Sets year to 2023

* **Set Month (0-11, where 0 = January):**

myDate.setMonth(5); // Sets month to June (0-indexed)

* **Set Day of the Month (1-31):**

myDate.setDate(15); // Sets the day to the 15th of the month

* **Set Hour (0-23):**

myDate.setHours(10); // Sets the hour to 10:00 AM

* **Set Minute (0-59):**

myDate.setMinutes(30); // Sets minutes to 30

* **Set Second (0-59):**

myDate.setSeconds(45); // Sets seconds to 45

* **Set Millisecond (0-999):**

myDate.setMilliseconds(500); // Sets milliseconds to 500

**4. Custom Formatting of Date and Time**

You may want to format dates and times in specific ways. JavaScript does not have a built-in way to format dates easily, but you can create custom formats using methods like getMonth(), getDate(), getFullYear(), and so on.

For example:

**Custom Date Format (YYYY-MM-DD):**

let customFormat = `${myDate.getFullYear()}-${String(myDate.getMonth() + 1).padStart(2, '0')}-${String(myDate.getDate()).padStart(2, '0')}`;

console.log(customFormat); // e.g., "2025-01-21"

**Custom Date and Time Format (DD/MM/YYYY HH:MM:SS):**

let customDateTimeFormat = `${String(myDate.getDate()).padStart(2, '0')}/${String(myDate.getMonth() + 1).padStart(2, '0')}/${myDate.getFullYear()} ${String(myDate.getHours()).padStart(2, '0')}:${String(myDate.getMinutes()).padStart(2, '0')}:${String(myDate.getSeconds()).padStart(2, '0')}`;

console.log(customDateTimeFormat); // e.g., "21/01/2025 10:30:45"

**Custom Weekday and Time Format:**

const weekdays = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];

let customWeekdayTimeFormat = `${weekdays[myDate.getDay()]}, ${myDate.getHours()}:${String(myDate.getMinutes()).padStart(2, '0')}`;

console.log(customWeekdayTimeFormat); // e.g., "Monday, 10:30"

**5. Important Notes on Date and Time Handling**

* **Time Zone Considerations:** JavaScript's Date object works with the local time zone of the machine running the code. If you need to handle different time zones, consider using libraries such as **Moment.js** or **date-fns**, or use methods like toISOString() (UTC) and toLocaleString() to format the date and time based on the user's locale.
* **Date Parsing:** When creating a Date object from a string, be cautious about browser inconsistencies with different date formats:

let myDate = new Date('2025-01-21'); // Consider using ISO 8601 format (YYYY-MM-DD)

Always use a consistent format (preferably ISO 8601) to avoid unexpected behaviors across different browsers.

**Example Code:**

// Create a new Date object

let myDate = new Date();

// Log various representations of the date and time

console.log(myDate); // Current date and time

console.log(myDate.toString()); // Human-readable string

console.log(myDate.toISOString()); // ISO 8601 format

console.log(myDate.toDateString()); // Date part only

console.log(myDate.toLocaleDateString());// Localized date format

console.log(myDate.toLocaleTimeString());// Localized time format

// Getting specific date and time components

console.log(myDate.getFullYear()); // Full year

console.log(myDate.getMonth()); // Month (0-11)

console.log(myDate.getDate()); // Day of the month (1-31)

console.log(myDate.getDay()); // Day of the week (0-6)

// Modifying date and time

myDate.setFullYear(2023);

myDate.setMonth(5); // June

console.log(myDate);

// Custom Date Format (YYYY-MM-DD)

let customFormat = `${myDate.getFullYear()}-${String(myDate.getMonth() + 1).padStart(2, '0')}-${String(myDate.getDate()).padStart(2, '0')}`;

console.log(customFormat); // e.g., "2023-06-15"

**JavaScript Arrays:**

**1. Basics of Arrays**

**What is an Array?**

An array is a special type of object in JavaScript used to store multiple values in a single variable. Arrays are indexed (starting from 0) and can store elements of any type (e.g., numbers, strings, objects).

**Syntax**

// Creating an array

let arr1 = [1, 2, 3]; // Literal notation

let arr2 = new Array(4, 5, 6); // Constructor notation

// Empty array

let emptyArr = [];

**2. Array Properties**

**Important Properties**

* **length**: Returns the number of elements in an array.

let arr = [1, 2, 3];

console.log(arr.length); // Output: 3

* **prototype**: Allows adding new methods to the array object.
* **constructor**: Returns the function that created the array prototype (Array).

console.log(arr.constructor); // Output: ƒ Array() { [native code] }

**3. Array Methods**

JavaScript provides mutator, accessor, and iteration methods to handle arrays.

**3.1 Mutator Methods (Modify the Original Array)**

* **push()**: Adds one or more elements to the end of the array.

let arr = [1, 2, 3];

arr.push(4, 5);

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

* **pop()**: Removes the last element from the array.

arr.pop();

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

* **unshift()**: Adds elements to the beginning of the array.

arr.unshift(0);

console.log(arr); // Output: [0, 1, 2, 3, 4]

* **shift()**: Removes the first element from the array.

arr.shift();

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

* **splice()**: Adds, removes, or replaces elements in the array.

let arr = [1, 2, 3, 4];

arr.splice(1, 2, 10, 20);

console.log(arr); // Output: [1, 10, 20, 4]

* **sort()**: Sorts elements in ascending order (default is lexicographic).

let nums = [3, 1, 2];

nums.sort();

console.log(nums); // Output: [1, 2, 3]

* **reverse()**: Reverses the elements in the array.

nums.reverse();

console.log(nums); // Output: [3, 2, 1]

**3.2 Accessor Methods (Do Not Modify the Array)**

* **concat()**: Combines two or more arrays.

let arr1 = [1, 2];

let arr2 = [3, 4];

let result = arr1.concat(arr2);

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

* **slice()**: Returns a shallow copy of part of the array.

let arr = [1, 2, 3, 4];

let sliced = arr.slice(1, 3);

console.log(sliced); // Output: [2, 3]

* **includes()**: Checks if an array contains a specific element.

console.log(arr.includes(3)); // Output: true

* **indexOf()**: Returns the first index of the specified element, or -1 if not found.

console.log(arr.indexOf(3)); // Output: 2

* **lastIndexOf()**: Returns the last index of the specified element.

let arr = [1, 2, 3, 2];

console.log(arr.lastIndexOf(2)); // Output: 3

* **join()**: Joins all elements into a string, separated by a specified separator.

console.log(arr.join('-')); // Output: "1-2-3"

* **toString()**: Converts the array to a string.

console.log(arr.toString()); // Output: "1,2,3"

**3.3 Iteration Methods**

* **forEach()**: Executes a function for each array element.

arr.forEach((el) => console.log(el));

* **map()**: Returns a new array with the result of calling a function on every element.

let squared = arr.map(x => x \* x);

console.log(squared); // Output: [1, 4, 9]

* **filter()**: Returns a new array with elements that pass a condition.

let filtered = arr.filter(x => x > 2);

console.log(filtered); // Output: [3]

* **reduce()**: Reduces the array to a single value.

let sum = arr.reduce((acc, curr) => acc + curr, 0);

console.log(sum); // Output: 6

* **find()**: Returns the first element that satisfies a condition.

console.log(arr.find(x => x > 2)); // Output: 3

* **findIndex()**: Returns the index of the first element that satisfies a condition.

console.log(arr.findIndex(x => x > 2)); // Output: 2

* **some()**: Checks if at least one element satisfies a condition.

console.log(arr.some(x => x > 2)); // Output: true

* **every()**: Checks if all elements satisfy a condition.

console.log(arr.every(x => x > 0)); // Output: true

**4. Advanced Techniques**

**1. Nested Arrays**

Handling multi-dimensional arrays (arrays inside arrays).

let nested = [[1, 2], [3, 4]];

console.log(nested[0][1]); // Output: 2

**2. Flattening Arrays**

Use **flat()** to flatten nested arrays.

let nested = [1, [2, 3], [4, [5, 6]]];

console.log(nested.flat(2)); // Output: [1, 2, 3, 4, 5, 6]

**3. Destructuring Arrays**

Extract values from arrays using destructuring.

let [a, b, c] = [1, 2, 3];

console.log(a, b, c); // Output: 1, 2, 3

**4. Spread Operator**

Use **...** to expand elements of an array.

let arr = [1, 2];

let newArr = [...arr, 3, 4];

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

**5. Copying Arrays**

Use **slice()** or spread to create a copy.

let arr = [1, 2];

let copy = [...arr];

**5. Array Best Practices**

* Use **const** for arrays you don’t want to reassign.
* Use **map**, **filter**, and **reduce** for clean and functional code.
* Avoid directly modifying arrays (mutator methods) when immutability is desired.
* Use **Array.isArray()** to check if a variable is an array.

console.log(Array.isArray([1, 2])); // Output: true

# JavaScript Objects: Detailed Explanation

## What is an Object in JavaScript?

An object in JavaScript is a collection of key-value pairs, where each key (also called a property) is a string or Symbol, and the value can be any data type (string, number, boolean, function, array, or even another object).

Objects are fundamental to JavaScript, as they allow you to store, manipulate, and organize data efficiently.

## 1. Creating Objects in JavaScript

There are multiple ways to create objects in JavaScript:

### 1.1 Using Object Literals

The easiest and most commonly used way to create an object is by using curly braces {}.

const person = {

name: "Amol",

age: 23,

location: "Pune",

email: "amolkadam174@gmail.com",

isLoggedIn: false

};

console.log(person);

### 1.2 Using the Object Constructor

Another way to create an object is by using the Object constructor.

const person = new Object(); // Creates an empty object

person.name = "Amol";

person.age = 23;

person.location = "Pune";

console.log(person);

### 1.3 Using Object.create()

The Object.create() method is used to create a new object with a specified prototype.

const prototypeObj = {

greet() {

console.log("Hello from prototype!");

}

};

// Create a new object with prototypeObj as its prototype

const newPerson = Object.create(prototypeObj);

newPerson.name = "Amol";

newPerson.age = 23;

console.log(newPerson);

newPerson.greet(); // Inherits method from prototype

## 2. Accessing Object Properties

There are two ways to access object properties:

### 2.1 Dot Notation (.)

Used when the property name is a valid identifier (without spaces or special characters).

console.log(person.name); // "Amol"

console.log(person.age); // 23

### 2.2 Bracket Notation ([])

Used when the property name has spaces, special characters, or is dynamic.

console.log(person["name"]); // "Amol"

const property = "age";

console.log(person[property]); // 23 (Dynamic property access)

## 3. Modifying Object Properties

Objects are mutable, meaning you can change their values after they are created.

person.age = 25; // Modifying an existing property

person.city = "Mumbai"; // Adding a new property

console.log(person);

## 4. Using Symbols as Object Keys

Symbols are unique identifiers that can be used as object keys.

const mySymbol = Symbol("uniqueKey");

const user = {

name: "Amol",

age: 23,

[mySymbol]: "SecretValue"

};

console.log(user[mySymbol]); // "SecretValue"

**Key points about symbols in objects:**

* They do not appear in Object.keys() or for...in loops.
* They help prevent property name collisions.

## 5. Freezing an Object (Object.freeze())

The Object.freeze() method prevents modifications to an object.

Object.freeze(person);

person.age = 30; // This change won't work

console.log(person.age); // Still 25

## 6. Adding Methods to an Object

Objects can have methods (functions inside objects).

const user = {

name: "Amol",

greet: function() {

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

}

};

user.greet(); // Output: Hello, Amol

**Note:** The this keyword refers to the current object.

## 7. Object Methods

### 7.1 Object.keys()

Returns an array of property names.

console.log(Object.keys(person)); // ["name", "age", "location", "email", "isLoggedIn"]

### 7.2 Object.values()

Returns an array of property values.

console.log(Object.values(person)); // ["Amol", 23, "Pune", "amolkadam174@gmail.com", false]

### 7.3 Object.entries()

Returns an array of key-value pairs.

console.log(Object.entries(person));

## 8. Looping Through Objects

### 8.1 Using for...in Loop

Loops through all enumerable properties of an object.

for (let key in person) {

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

}

### 8.2 Using Object.entries() and forEach()

Object.entries(person).forEach(([key, value]) => {

console.log(`${key}: ${value}`);

});

## 9. Nested Objects

An object can contain another object.

const student = {

name: "Amol",

marks: {

math: 90,

science: 85

}

};

// Accessing nested object properties

console.log(student.marks.math); // 90

console.log(student["marks"]["science"]); // 85

## 10. Object Destructuring

A shortcut to extract properties from objects.

const { name, age } = person;

console.log(name, age); // "Amol", 23

## 11. Merging Objects

### Using Object.assign()

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

const obj2 = { c: 3, d: 4 };

const mergedObj = Object.assign({}, obj1, obj2);

console.log(mergedObj);

### Using Spread Operator (...)

const mergedObj2 = { ...obj1, ...obj2 };

console.log(mergedObj2);

## 12. Checking if a Property Exists (hasOwnProperty())

console.log(person.hasOwnProperty("age")); // true

console.log(person.hasOwnProperty("salary")); // false

## 13. Object Sealing (Object.seal())

Prevents new properties from being added but allows modifying existing ones.

Object.seal(person);

person.age = 30; // Allowed

person.salary = 50000; // Not allowed

console.log(person);

## 14. Cloning an Object

### Using Object.assign()

const clone1 = Object.assign({}, person);

console.log(clone1);

### Using Spread Operator (...)

const clone2 = { ...person };

console.log(clone2);

## Final Thoughts

Objects are one of the most powerful features in JavaScript. They provide a way to structure, manage, and manipulate data efficiently. Mastering objects is essential for JavaScript development.

## **2. JSON API (JavaScript Object Notation API)**

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

### **Key Points:**

1. **Fetching Data from a JSON API:**

fetch("https://jsonplaceholder.typicode.com/users")

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

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

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

1. **Parsing JSON Data:**

JSON.parse(): Converts JSON string into a JavaScript object.

const jsonData = '{"name":"Amol", "age":25}';

const obj = JSON.parse(jsonData);

console.log(obj.name); // "Amol"

JSON.stringify(): Converts JavaScript object into a JSON string.

const user = { name: "Amol", age: 25 };

const jsonString = JSON.stringify(user);

console.log(jsonString); // '{"name":"Amol","age":25}'

1. **Async/Await with Fetch API:**

async function fetchData() {

try {

let response = await fetch("https://jsonplaceholder.typicode.com/posts");

let data = await response.json();

console.log(data);

} catch (error) {

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

}

}

fetchData();

1. **Handling HTTP Methods with Fetch:**

fetch("https://jsonplaceholder.typicode.com/posts", {

method: "POST",

headers: {

"Content-Type": "application/json"

},

body: JSON.stringify({

title: "New Post",

body: "This is the body of the post",

userId: 1

})

})

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

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

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

### **Common JSON API Status Codes:**

| **Status Code** | **Meaning** |
| --- | --- |
| 200 OK | Successful request |
| 201 Created | Resource successfully created |
| 400 Bad Request | Client-side error |
| 401 Unauthorized | Authentication required |
| 403 Forbidden | Access denied |
| 404 Not Found | Resource not found |
| 500 Internal Server Error | Server-side issue |

## **Conclusion**

* Object destructuring simplifies the extraction of properties from objects.
* JSON APIs are widely used for data exchange in web applications.
* Fetch API allows making HTTP requests to interact with JSON APIs.
* Understanding status codes helps in debugging API responses efficiently.

**Functions**

### 1. **Functions in JavaScript**

A function in JavaScript is a block of reusable code that performs a specific task. Functions help in modular programming, reducing code redundancy, improving maintainability, and enhancing readability. JavaScript functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

#### **Function Declaration**

A function can be declared using the function keyword. These functions are hoisted, meaning they can be used before being defined in the code.

function greet() {

console.log("Hello, World!");

}

greet();

#### **Function Expression**

A function expression is when a function is assigned to a variable. Unlike function declarations, function expressions are not hoisted.

const greet = function() {

console.log("Hello, World!");

};

greet();

#### **Arrow Function**

Introduced in ES6, arrow functions provide a concise syntax for writing functions. They do not bind their own this value, making them useful in certain situations like event handling and callbacks.

const greet = () => console.log("Hello, World!");

greet();

### 2. **Functions with Parameters**

Functions can accept parameters to perform operations dynamically. Parameters act as placeholders for values that are passed into the function when it is invoked.

function add(a, b) {

return a + b;

}

console.log(add(5, 3)); // Output: 8

### 3. **Function with Objects as Parameters**

Objects can be passed as arguments to functions to work with structured data efficiently. This technique is often used in APIs and data manipulation.

function displayPerson(person) {

console.log(`Name: ${person.name}, Age: ${person.age}`);

}

const personObj = { name: "John", age: 30 };

displayPerson(personObj);

// Output: Name: John, Age: 30

### 4. **Returning an Object from a Function**

A function can return an object to encapsulate related data and behavior.

function createPerson(name, age) {

return { name, age };

}

const person = createPerson("Alice", 25);

console.log(person);

// Output: { name: "Alice", age: 25 }

### 5. **Function with Default Parameters**

Default values can be assigned to function parameters to prevent undefined values when arguments are not provided.

function greet(name = "Guest") {

console.log(`Hello, ${name}!`);

}

greet(); // Output: Hello, Guest!

### 6. **Rest Parameters in Functions**

Rest parameters allow handling an indefinite number of arguments as an array.

function sum(...numbers) {

return numbers.reduce((acc, num) => acc + num, 0);

}

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

### 7. **Function with Object Destructuring**

Objects can be destructured within function parameters to extract specific properties directly.

function printDetails({ name, age }) {

console.log(`Name: ${name}, Age: ${age}`);

}

const user = { name: "Emma", age: 22 };

printDetails(user);

// Output: Name: Emma, Age: 22

### 8. **Immediately Invoked Function Expression (IIFE)**

IIFE is executed immediately after its definition, often used to create a local scope and avoid polluting the global namespace.

(function() {

console.log("IIFE executed");

})();

// Output: IIFE executed

### 9. **Higher-Order Functions**

Higher-order functions take other functions as arguments or return functions, enabling functional programming patterns.

function applyOperation(a, b, operation) {

return operation(a, b);

}

const multiply = (x, y) => x \* y;

console.log(applyOperation(5, 3, multiply)); // Output: 15

### 10. **Callback Functions**

A callback function is passed as an argument to another function and executed later, commonly used in asynchronous operations.

function fetchData(callback) {

setTimeout(() => {

callback("Data received");

}, 2000);

}

fetchData((message) => console.log(message));

// Output: Data received (after 2 seconds)

### 11. **Closures**

A closure is a function that retains access to its parent function's scope even after the parent function has executed. This is useful for data encapsulation and maintaining state.

function counter() {

let count = 0;

return function() {

count++;

console.log(count);

};

}

const increment = counter();

increment(); // Output: 1

increment(); // Output: 2

### 12. **Function Currying**

Currying is transforming a function with multiple arguments into a series of functions, each taking a single argument. This is useful in functional programming.

function multiply(a) {

return function(b) {

return a \* b;

};

}

const double = multiply(2);

console.log(double(5)); // Output: 10

### 13. **Arrow Functions and** this **Keyword**

Arrow functions do not bind their own this, instead inheriting it from the surrounding scope. This makes them useful for handling this inside event handlers and callbacks.

const obj = {

name: "John",

greet: function() {

setTimeout(() => {

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

}, 1000);

}

};

obj.greet(); // Output: Hello, John

### 14. **Function Hoisting**

Function declarations are hoisted, meaning they can be called before their definition in the code. However, function expressions are not hoisted.

console.log(sayHello());

function sayHello() {

return "Hello!";

}

# **Variable Scope in JavaScript**

In JavaScript, **scope** refers to the area of code where a variable is accessible. A variable's scope is determined by where it is declared. The primary types of scopes in JavaScript are:

1. **Global Scope**
2. **Local Scope**
3. **Block Scope**

## **1. Global Scope**

A variable declared outside of any function or block is said to have **global scope**. It is accessible from any part of the code, including inside functions.

### Example:

let globalVar = "I am a global variable";

function testGlobal() {

console.log(globalVar); // Accessible here

}

testGlobal();console.log(globalVar); // Accessible here too

### Key Points:

* Variables with global scope can be accessed anywhere in the code.
* Global variables are often considered bad practice in large applications because they can be changed unexpectedly, leading to bugs.

## **2. Local Scope (Function Scope)**

When a variable is declared inside a function, it is said to have **local scope** (also known as **function scope**). It can only be accessed within that function.

### Example:

function testLocal() {

let localVar = "I am a local variable";

console.log(localVar); // Accessible inside the function

}

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

### Key Points:

* A variable with local scope cannot be accessed outside the function where it is declared.
* Local variables are created when the function is invoked and destroyed when the function execution is completed.

## **3. Block Scope (ES6** let **and** const**)**

In JavaScript, a block scope is introduced with let and const. These variables are confined to the block (like loops or conditionals) where they are declared.

### Example:

if (true) {

let blockVar = "I am inside a block";

console.log(blockVar); // Accessible here

}

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

### Key Points:

* Variables declared with let and const are **block-scoped**.
* They are not accessible outside the block, unlike var.

## **4. Function Scope (vs Block Scope)**

In JavaScript, the var keyword is **function-scoped**, meaning it is available within the function in which it is declared, but not outside. However, let and const are **block-scoped**, meaning they are only accessible within the block they are declared in (such as inside {} in loops, conditionals, etc.).

### Example (Function vs Block Scope):

function scopeTest() {

if (true) {

var functionScoped = "I am function-scoped"; // `var` is function-scoped

let blockScoped = "I am block-scoped"; // `let` is block-scoped

}

console.log(functionScoped); // Accessible here (because it's `var`)

console.log(blockScoped); // Error: blockScoped is not defined (because it's `let`)

}

scopeTest();

### Key Points:

* var is function-scoped and can leak out of blocks.
* let and const are block-scoped and do not leak out of blocks.

## **5. Hoisting**

In JavaScript, variables declared with var are **hoisted** to the top of their scope. This means that the declaration is moved to the top during the execution phase, but not the assignment.

* var **hoisting:** Variable declarations are hoisted, but their assignments are not.
* let **and** const **hoisting:** Variables are hoisted but remain uninitialized, resulting in a **temporal dead zone** (TDZ) if accessed before declaration.

### Example:

console.log(varVar); // undefined (hoisted)var varVar = "I am var";

console.log(letVar); // Error: Cannot access 'letVar' before initializationlet letVar = "I am let";

### Key Points:

* var: Hoisted and initialized with undefined.
* let **and** const: Hoisted but not initialized, leading to TDZ if accessed early.

## **6. Lexical Scope (Closures)**

JavaScript uses **lexical scoping**, meaning that the scope of a variable is determined by the physical location of the variable within the source code. A **closure** is a function that retains access to its lexical scope even when the function is executed outside that scope.

### Example:

function outerFunction() {

let outerVar = "I am outside";

function innerFunction() {

console.log(outerVar); // Accessing outerVar from the outer function

}

return innerFunction;

}

const closureFunc = outerFunction();closureFunc(); // Outputs: "I am outside"

### Key Points:

* A closure allows an inner function to access variables from its outer function's scope.
* This is possible even when the outer function has finished executing.

## **7. Global Object (**window **in Browser)**

In the browser environment, variables declared globally are properties of the window object.

### Example:

let globalVar = "I am a global variable";console.log(window.globalVar); // Outputs: "I am a global variable"

### Key Points:

* Global variables can be accessed through the window object in the browser.
* It's a best practice to avoid polluting the global scope with too many global variables

## **8. Scope Chain**

The **scope chain** refers to the hierarchy of scopes that JavaScript uses to resolve variable lookups. It starts from the current scope and then looks outward to the outer scopes until it reaches the global scope.

### Example:

let outer = "I am outside";

function innerFunction() {

console.log(outer); // The engine looks for 'outer' in the current scope (innerFunction), then outer scope

}

innerFunction(); // Outputs: "I am outside"

### Key Points:

* The scope chain determines where JavaScript looks for variables.
* Variables are searched starting from the innermost scope and moving outward.

## **9. Shadowing**

**Variable shadowing** occurs when a variable declared in a local scope has the same name as a variable in an outer scope, causing the inner variable to "shadow" or "hide" the outer one.

### Example:

let outerVar = "I am outer";

function shadowTest() {

let outerVar = "I am inner"; // Shadowing the outer variable

console.log(outerVar); // Outputs: "I am inner"

}

shadowTest();console.log(outerVar); // Outputs: "I am outer"

### Key Points: Inner variables with the same name as outer variables can shadow the outer ones within the local scope.

**JavaScript Symbol Data Type**

## 1. What is Symbol in JavaScript?

* A **Symbol** is a unique and immutable primitive value introduced in ES6.
* It is used as an identifier for object properties.
* Unlike strings, every Symbol value is guaranteed to be **unique**.

## 2. Creating a Symbol

You can create a Symbol using the Symbol() function.

### Example:

const sym1 = Symbol();

const sym2 = Symbol();

console.log(sym1 === sym2); // false (Each Symbol is unique)

Even if both sym1 and sym2 are created without arguments, they are still unique.

## 3. Symbol with Description

You can provide an **optional description** while creating a Symbol for debugging purposes.

### Example:

const sym = Symbol("description");

console.log(sym.toString()); // Output: "Symbol(description)"

The description does not affect the uniqueness of the Symbol.

## 4. Using Symbols as Object Properties

Symbols are mainly used as **property keys** in objects to prevent property name conflicts.

### Example:

const id = Symbol("id");

let user = {

name: "Amol",

[id]: 101 // Symbol as a property key

};

console.log(user[id]); // Output: 101

**Note:** Square brackets [] are required when using a Symbol as a property key.

## 5. Symbols Are Unique

Even if two symbols have the same description, they are still unique.

### Example:

const symA = Symbol("test");

const symB = Symbol("test");

console.log(symA === symB); // false

Each Symbol() call generates a **unique** value.

## 6. Global Symbol Registry (Symbol.for())

JavaScript provides a **global registry** where you can create and reuse symbols.

### Example:

const sym1 = Symbol.for("key");

const sym2 = Symbol.for("key");

console.log(sym1 === sym2); // true

Symbol.for() checks if a symbol with the given key already exists. If it does, it returns the existing symbol; otherwise, it creates a new one.

## 7. Retrieving Symbol Key (Symbol.keyFor())

If a symbol is created using Symbol.for(), you can retrieve its key.

### Example:

const sym = Symbol.for("name");

console.log(Symbol.keyFor(sym)); // Output: "name"

This only works for symbols registered with Symbol.for().

## 8. Built-in Well-Known Symbols

JavaScript has predefined Symbol values called **well-known symbols**, which modify built-in behaviors.

| **Symbol** | **Description** |
| --- | --- |
| Symbol.iterator | Used for making objects iterable |
| Symbol.toStringTag | Customizes Object.prototype.toString() |
| Symbol.toPrimitive | Defines conversion behavior for objects |
| Symbol.hasInstance | Customizes instanceof behavior |
| Symbol.isConcatSpreadable | Controls array merging with concat() |
| Symbol.unscopables | Hides properties from with scope |

### Example: Symbol.iterator

let myIterable = {

\*[Symbol.iterator]() {

yield 1;

yield 2;

yield 3;

}

};

console.log([...myIterable]); // Output: [1, 2, 3]

Symbol.iterator allows objects to be iterable using for...of or spread syntax.

## 9. Checking if an Object Has a Symbol Property

Symbols are **not included** in Object.keys() or for...in, but can be retrieved using Object.getOwnPropertySymbols().

### Example:

const id = Symbol("id");

let person = {

name: "Amol",

[id]: 123

};

console.log(Object.keys(person)); // Output: ["name"]

console.log(Object.getOwnPropertySymbols(person)); // Output: [ Symbol(id) ]

Symbol properties are **hidden** from normal enumeration.

## 10. Converting Symbols to Strings

Symbols cannot be **directly** converted to a string.

### Example:

const sym = Symbol("test");

console.log(sym.toString()); // Output: "Symbol(test)"

console.log(String(sym)); // Output: "Symbol(test)"

You cannot concatenate symbols with strings directly ("Hello " + sym gives an error).

## 11. When to Use Symbols

✅ **Use Symbols when:**

* You need **unique** property keys.
* You want to prevent **property name clashes**.
* You want **hidden** object properties.
* You need to define **custom behavior** using well-known symbols.

❌ **Avoid Symbols when:**

* You need **JSON serialization** (Symbols are ignored in JSON.stringify()).
* You need properties that can be **enumerated**.

## 12. Summary

* **Symbol** is a **unique** and **immutable** primitive type.
* Symbols are mainly used as **object property keys**.
* Symbols created with Symbol() are **always unique**.
* Symbols created with Symbol.for() are **global and shared**.
* Symbols do **not** show up in normal object property loops.
* **Well-known symbols** modify built-in behaviors.

## **What is an IIFE?**

An **Immediately Invoked Function Expression (IIFE)** is a JavaScript function that **executes immediately after it is defined**.

**Syntax of IIFE:**

(function () {

console.log("IIFE executed");

})();

### **How It Works:**

1. **Function Declaration Inside Parentheses** (function() {...})
2. This creates an **anonymous function** inside parentheses.
3. The parentheses () around the function tell JavaScript that it's a function expression, **not a function declaration**.
4. **Function Invocation** ();
5. The () at the end immediately calls the function.
6. Without ();, the function would only be defined but **not executed**.

### **Example Execution**

(function () {

console.log("IIFE executed");

})();

* **Step 1:** The function is defined **inside parentheses**.
* **Step 2:** The function **immediately executes** when the interpreter encounters ();.
* **Output:** "IIFE executed"

## **Why Use IIFE?**

### ✅ **1. Avoids Polluting the Global Scope**

Variables inside an IIFE **stay inside** the function and do not interfere with other variables.

(function () {

let message = "IIFE Scope";

console.log(message);

})();

console.log(message); // ❌ ReferenceError: message is not defined

* message **only exists inside the IIFE**.
* Outside the IIFE, message is **not accessible**, preventing global scope pollution.

### ✅ **2. Creates Private Variables**

IIFE is useful for creating **private variables** that **cannot be accessed** from outside.

const counter = (function () {

let count = 0; // Private variable

return {

increment: function () {

count++;

console.log(count);

},

decrement: function () {

count--;

console.log(count);

}

};

})();

counter.increment(); // 1

counter.increment(); // 2

counter.decrement(); // 1console.log(counter.count); // ❌ Undefined (private variable)

* count is a **private variable** inside the IIFE.
* Only increment and decrement functions can **access** count.

### ✅ **3. Useful for Initialization Code**

IIFE is often used for **one-time initialization** in applications.

(function () {

console.log("Application Initialized");

})(); // Output: Application Initialized

* Runs once when the script loads.
* Used in frameworks/libraries for **setup configurations**.

### ✅ **4. Supports Parameter Passing**

You can pass parameters into an IIFE just like a normal function.

(function (name) {

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

})("Amol"); // Output: Hello, Amol

* The IIFE accepts "Amol" as an argument.

## **Variations of IIFE**

### **1. Named IIFE**

(function greet() {

console.log("Named IIFE");

})();

* The function is named greet, but still **executes immediately**.

### **2. Arrow Function IIFE**

(() => {

console.log("Arrow Function IIFE");

})();

* **Shorter syntax** using an **arrow function**.

### **3. IIFE with Multiple Parameters**

((a, b) => {

console.log(a + b);

})(5, 10);// Output: 15

* Passes multiple arguments to an IIFE.

| **Feature** | **Explanation** |
| --- | --- |
| **What is IIFE?** | A function that executes immediately after being defined. |
| **Why Use It?** | Avoids global scope pollution, creates private variables, useful for initialization. |
| **Syntax** | (function() { ... })(); |
| **Can Take Parameters?** | ✅ Yes, just like normal functions. |
| **Common Use Cases** | One-time setup, private variables, preventing name conflicts. |

**Execution Context in JavaScript**

### **Introduction**

Execution Context is a fundamental concept in JavaScript that defines the environment in which code is executed. It determines how variables, functions, and objects are stored and accessed during runtime.

### **Types of Execution Context**

JavaScript has three types of execution contexts:

1. **Global Execution Context (GEC)**
2. **Function Execution Context (FEC)**
3. **Eval Execution Context**

### **1. Global Execution Context (GEC)**

* Created when the JavaScript file is first executed.
* There is only one global execution context in the entire JavaScript program.
* It creates the global object (window in browsers, global in Node.js) and sets this to the global object.

**Example:**

console.log(this); // In browsers, it refers to the window object

### **2. Function Execution Context (FEC)**

* Created whenever a function is invoked.
* Each function has its own execution context.
* It includes its own variable environment, scope chain, and this binding.

**Example:**

function greet() {

console.log("Hello, World!");

}

greet(); // Creates a new Function Execution Context

### **3. Eval Execution Context**

* Created when the eval() function is used to execute JavaScript code inside a string.
* It is rarely used due to security and performance concerns.

**Example:**

eval("console.log('Eval Execution Context');");

### **Execution Context Phases**

An execution context goes through two main phases:

#### **1. Creation Phase**

* The JavaScript engine scans the code before execution and sets up the execution context.
* It creates:
  + A \*\*Global \*\***Object** (window in browsers, global in Node.js).
  + A **this** reference.
  + A **Memory Heap** for storing variables and function declarations.
  + A **Lexical Environment** (stores function and variable declarations).

#### **2. Execution Phase**

* The JavaScript engine executes the code line by line.
* Variables and functions are assigned values.

**Example:**

var x = 5; // Memory is allocated in the creation phase, assigned value in execution phase

function test() {

console.log("Inside function");

}

test();

### **Call Stack and Execution Context**

The **Call Stack** manages execution contexts in JavaScript. It follows the **LIFO (Last In, First Out)** principle.

**Example:**

function first() {

console.log("First Function");

second();

}

function second() {

console.log("Second Function");

}

first();

**Call Stack Process:**

1. first() is called → first() Execution Context is pushed onto the Call Stack.
2. first() calls second() → second() Execution Context is pushed onto the Call Stack.
3. second() finishes execution → second() Execution Context is removed.
4. first() finishes execution → first() Execution Context is removed.

### **Hoisting in Execution Context**

* Variables declared with var and function declarations are moved to the top during the creation phase.
* Only function declarations are fully hoisted, while var variables are hoisted with undefined.

**Example:**

console.log(a); // Undefined (due to hoisting)

var a = 10;

hoistedFunction(); // Works fine

function hoistedFunction() {

console.log("Hoisted Function");

}

### **Lexical Environment & Scope Chain**

* **Lexical Environment:** Stores variables and function declarations.
* **Scope Chain:** Determines the accessibility of variables from different levels of execution contexts.

**Example:**

function outer() {

let a = 10;

function inner() {

console.log(a); // Can access 'a' from outer function

}

inner();

}

outer();

### **Conclusion**

Execution Context is a crucial concept that helps JavaScript manage function calls, variable accessibility, and memory allocation. Understanding how it works improves debugging skills and optimizes JavaScript code performance.

# **Conditional Statements (Decision Making)**

Conditional statements allow us to execute different blocks of code based on certain conditions. JavaScript provides several conditional statements:

### **A. Basic Conditional Statements**

1. if Statement
2. if-else Statement
3. if-else-if Ladder
4. Nested if Statements

## **1.** if **Statement**

The if statement executes a block of code **only if** the given condition evaluates to true.

### **Syntax:**

if (condition) {

// Code to execute if the condition is true

}

### **Example:**

let age = 18;

if (age >= 18) {

console.log("You are eligible to vote.");

}

### **Explanation:**

* If age is greater than or equal to 18, the message **"You are eligible to vote."** will be printed.
* If the condition is false, nothing happens.

## **2. if-else Statement**

The if-else statement executes one block of code if the condition is true and another block if the condition is false.

### **Syntax:**

if (condition) {

// Code executes if the condition is true

} else {

// Code executes if the condition is false

}

### **Example:**

let num = 10;

if (num % 2 === 0) {

console.log("Even number");

} else {

console.log("Odd number");

}

### **Explanation:**

* If num is divisible by 2, it prints **"Even number"**.
* Otherwise, it prints **"Odd number"**.

## **3.** if-else-if **Ladder**

The if-else-if ladder is used to check **multiple conditions**. It executes the first true condition and ignores the rest.

### **Syntax:**

if (condition1) {

// Code for condition1

} else if (condition2) {

// Code for condition2

} else if (condition3) {

// Code for condition3

} else {

// Code if none of the conditions are true

}

### **Example:**

let marks = 85;

if (marks >= 90) {

console.log("Grade: A+");

} else if (marks >= 80) {

console.log("Grade: A");

} else if (marks >= 70) {

console.log("Grade: B");

} else if (marks >= 60) {

console.log("Grade: C");

} else {

console.log("Grade: F (Fail)");

}

### **Explanation:**

* If marks is 90 or above, it prints **"Grade: A+"**.
* If marks is between 80 and 89, it prints **"Grade: A"**.
* If marks is between 70 and 79, it prints **"Grade: B"**.
* If marks is between 60 and 69, it prints **"Grade: C"**.
* Otherwise, it prints **"Grade: F (Fail)"**.

## **4. Nested** if **Statements**

A **nested** if **statement** is when an if statement is inside another if or else block.

### **Syntax:**

if (condition1) {

if (condition2) {

// Code executes if both conditions are true

} else {

// Code executes if condition1 is true but condition2 is false

}

} else {

// Code executes if condition1 is false

}

### **Example:**

let age = 20;let hasID = true;

if (age >= 18) {

if (hasID) {

console.log("You can enter the club.");

} else {

console.log("You need an ID to enter.");

}

} else {

console.log("You are not allowed to enter.");

}

### **Explanation:**

* If age is **18 or above**, it checks if hasID is true.
* If both conditions are true, it prints **"You can enter the club."**.
* If age is **18 or above but has no ID**, it prints **"You need an ID to enter."**.
* If age is **below 18**, it prints **"You are not allowed to enter."**.

## **Conclusion**

* if: Executes code only if the condition is true.
* if-else: Provides an alternative block when the condition is false.
* if-else-if: Checks multiple conditions one by one.
* **Nested** if: Uses if inside another if to check **dependent conditions**.

# **Ternary Operator (? :)**

The **ternary operator** is a shorthand for an if-else statement. It is used for **conditional expressions** in a single line.

## **Syntax**

condition ? expr1 : expr2;

* If condition is **true**, expr1 is executed.
* If condition is **false**, expr2 is executed.

## **Example**

let age = 20;

let status = age >= 18 ? "Adult" : "Minor";

console.log(status); // Output: Adult

## **Nested Ternary Operators**

You can **nest** ternary operators inside each other for multiple conditions.

### **Syntax**

condition1 ? expr1 :

condition2 ? expr2 :

condition3 ? expr3 : expr4;

### **Example**

let marks = 85;

let result = marks >= 90 ? "A" :

marks >= 75 ? "B" :

marks >= 60 ? "C" : "Fail";

console.log(result); // Output: B

## **Using Ternary Operator for Assignments**

It is often used for **assigning values** based on conditions.

### **Example**

let num = 5;

let type = (num % 2 === 0) ? "Even" : "Odd";

console.log(type); // Output: Odd

## **Ternary Operator vs If-Else**

| **Feature** | **Ternary Operator (? :)** | **If-Else Statement** |
| --- | --- | --- |
| **Readability** | Concise & short | More readable for complex conditions |
| **Performance** | Faster in simple cases | Slightly slower but negligible |
| **Best Use Case** | Simple conditions | Complex logic & multiple statements |

### **Example**

**Using** if-else

let x = 10;let message;if (x > 5) {

message = "Greater than 5";

} else {

message = "Less than or equal to 5";

}

console.log(message);

**Using Ternary**

let x = 10;

let message = x > 5 ? "Greater than 5" : "Less than or equal to 5";

console.log(message);

✅ The ternary operator makes the code shorter and cleaner.

## **Important Points to Remember**

The ternary operator is an **expression**, so it **returns a value**.  
 It should be used for **simple conditions** to improve code readability.  
 Avoid **excessive nesting** as it makes code **hard to read**.  
 Parentheses can be used to make complex conditions clearer.

## **Advanced Usage: Ternary Inside Functions**

### **Example**

function getDiscount(price) {

return price > 100 ? "10% discount" : "No discount";

}

console.log(getDiscount(120)); // Output: 10% discount

console.log(getDiscount(80)); // Output: No discount

**Switch Case**

### 1. **switch Statement Syntax**

The switch statement is used to execute one block of code from multiple options based on the value of an expression.

**Syntax:**

switch(expression) {

case value1:

// Code block

break;

case value2:

// Code block

break;

default:

// Default code block (optional)

}

### 2. **case and default**

* Each case defines a potential match for the switch expression.
* The default case executes when none of the cases match (optional but recommended).

**Example:**

let day = 3;

switch (day) {

case 1:

console.log("Monday");

break;

case 2:

console.log("Tuesday");

break;

case 3:

console.log("Wednesday");

break;

default:

console.log("Invalid day");

}

### 3. **break in switch**

* The break statement prevents fall-through and exits the switch statement after a case is executed.
* Without break, execution continues to the next case (fall-through behavior).

### 4. **Fall-through Behavior (When Break is Omitted)**

* If break is omitted, execution continues to the next case even if it does not match.

**Example:**

let number = 2;

switch (number) {

case 1:

console.log("One");

case 2:

console.log("Two");

case 3:

console.log("Three");

}

**Output:**

Two

Three

### 5. **Using switch with Strings, Numbers, and Expressions**

#### **Switch with Strings**

let color = "Red";

switch (color) {

case "Red":

console.log("Stop");

break;

case "Yellow":

console.log("Slow Down");

break;

case "Green":

console.log("Go");

break;

}

#### **Switch with Numbers**

Switch statements can be used with numbers and strings.

let num = 10;

switch (num) {

case 5:

console.log("Five");

break;

case 10:

console.log("Ten");

break;

}

#### **Switch with Expressions**

* JavaScript supports expressions inside switch statements.

let score = 85;

switch (true) {

case score >= 90:

console.log("Grade A");

break;

case score >= 80:

console.log("Grade B");

break;

case score >= 70:

console.log("Grade C");

break;

default:

console.log("Fail");

}

### 6. **Strict Comparison in switch**

* switch uses **strict comparison (**===**)**, meaning type coercion does not occur.

let value = "5";

switch (value) {

case 5:

console.log("Number 5");

break;

case "5":

console.log("String 5");

break;

}

**Output:**

String 5

### 7. **Nested switch Statements**

* You can use a switch statement inside another switch statement.

let category = "Fruit";

let item = "Apple";

switch (category) {

case "Fruit":

switch (item) {

case "Apple":

console.log("It is an apple");

break;

case "Banana":

console.log("It is a banana");

break;

}

break;

case "Vegetable":

console.log("It is a vegetable");

break;

}

### **Comparing switch vs. if-else**

| **Feature** | **switch** | **if-else** |
| --- | --- | --- |
| Data types | Works well with numbers and strings | Works with all data types |
| Readability | Cleaner for multiple conditions | Can become complex with many conditions |
| Performance | Faster in some cases (internally optimized) | Slower in certain cases |
| Expression Handling | Supports specific discrete values | Supports complex boolean expressions |

### **Best Practices for Using switch**

* Always use break to prevent fall-through behavior unless intentional.
* Use default to handle unexpected cases.
* Prefer if-else for complex conditions and range-based comparisons.
* Avoid excessive nesting for readability.

### **Key Takeaways:**

* Use switch for **discrete values** (numbers, strings).
* Use if-else for **boolean expressions and range conditions**.
* Always use break to avoid fall-through behavior (unless needed).
* switch statements improve readability when dealing with multiple discrete conditions.

**Truthy and Falsy Values in JavaScript**

### **1. Truthy Values (Evaluate to true in Boolean context)**

A value is considered **truthy** if it evaluates to true when converted to a Boolean.

#### **Examples of Truthy Values:**

* **Non-empty strings**: "Hello", "0", "false" (even though "false" is a string, it is still truthy)
* **Non-zero numbers**: 1, -1, 3.14
* **Objects**: {}, [] (empty objects and arrays are truthy)
* **Functions**: function() {}
* **Infinity and -Infinity**: Infinity, -Infinity

#### **Example in JavaScript:**

if ("Hello") {

console.log("This is truthy");

} // Output: This is truthy

if ([]) {

console.log("Empty arrays are truthy");

} // Output: Empty arrays are truthy

if ({}) {

console.log("Empty objects are truthy");

} // Output: Empty objects are truthy

### **2. Falsy Values (Evaluate to false in Boolean context)**

A value is considered **falsy** if it evaluates to false when converted to a Boolean.

#### **Examples of Falsy Values:**

* false
* 0 (Number Zero)
* -0 (Negative Zero)
* "" (Empty String)
* null
* undefined
* NaN (Not-a-Number)

#### **Example in JavaScript:**

if (false) {

console.log("This will not print");

} else {

console.log("False is falsy");

} // Output: False is falsy

if (0) {

console.log("Zero is falsy");

} else {

console.log("0 is falsy");

} // Output: 0 is falsy

if ("") {

console.log("Empty string is falsy");

} else {

console.log("Empty string is falsy");

} // Output: Empty string is falsy

### **3. Implicit Boolean Conversion in JavaScript**

JavaScript automatically converts values to Boolean in certain scenarios:

#### **Methods of Implicit Conversion:**

1. **Using** if(value)
2. **Using** !!value **(Double NOT Operator)**
3. **Using** Boolean(value) **Constructor**

#### **Examples:**

let val1 = "Hello";

if (val1) {

console.log("Truthy value");

} // Output: Truthy value

console.log(!!val1); // Output: true

console.log(Boolean(val1)); // Output: true

let val2 = 0;

console.log(!!val2); // Output: false

console.log(Boolean(val2)); // Output: false

### **4. Short-Circuiting with** &&**,** ||**, and** ?? **Operators**

Short-circuiting is when JavaScript stops evaluating an expression as soon as the result is determined.

#### **4.1 Logical AND (**&&**)**

* Returns **first falsy value** or the **last truthy value**.
* Used for **guard clauses**.

console.log("Hello" && "World"); // Output: World (both are truthy, so last value is returned)

console.log("Hello" && 0); // Output: 0 (first falsy value)

console.log(0 && "Hello"); // Output: 0 (first falsy value)

#### **4.2 Logical OR (||)**

* Returns **first truthy value** or the **last falsy value**.
* Used for **default values**.

console.log("" || "Default"); // Output: Default (first truthy value)

console.log(0 || false || "Hello"); // Output: Hello (first truthy value)

#### **4.3 Nullish Coalescing (**??**)**

* Returns the **right-hand side operand** if the **left-hand side is null or undefined**.
* Used for **handling null or undefined values**.

console.log(null ?? "Fallback"); // Output: Fallback

console.log(undefined ?? "Default"); // Output: Default

console.log(0 ?? "Default"); // Output: 0 (because 0 is NOT null or undefined)

### **5. Practical Use Cases of Truthy and Falsy Values**

#### **5.1 Providing Default Values**

Using ||:

let user = "";

let username = user || "Guest";

console.log(username); // Output: Guest

Using ?? (to handle only null and undefined):

let user2 = null;

let username2 = user2 ?? "Guest";

console.log(username2); // Output: Guest

#### **5.2 Conditional Execution Using** &&

let isLoggedIn = true;

isLoggedIn && console.log("User is logged in"); // Output: User is logged in

#### **5.3 Checking for Valid Inputs**

function greet(name) {

console.log(name || "Stranger");

}

greet("Amol"); // Output: Amol

greet(""); // Output: Stranger

### **Summary**

* **Truthy values**: Any value that evaluates to true in a Boolean context.
* **Falsy values**: false, 0, -0, "", null, undefined, NaN.
* **Implicit Boolean conversion**: if(value), !!value, Boolean(value).
* **Short-circuiting**: && (first falsy), || (first truthy), ?? (null/undefined handling).
* **Common use cases**: Default values, conditional execution, and input validation.

**Loops (Iteration Statements) in JavaScript**

Loops are used to execute a block of code multiple times until a specified condition is met.

## **A. Entry-Controlled Loops**

These loops check the condition before executing the loop body.

### **1. for Loop**

Used when the number of iterations is known.

**Syntax:**

for (initialization; condition; increment/decrement) {

// Code to execute

}

**Example: Print numbers from 1 to 5**

for (let i = 1; i <= 5; i++) {

console.log(i);

}

### **Looping Over Arrays**

Used to iterate through array elements.

**Example:**

const arr = [10, 20, 30, 40, 50];

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

console.log(arr[i]);

}

### **Looping in Reverse Order**

Used to iterate in reverse.

**Example:**

const nums = [1, 2, 3, 4, 5];

for (let i = nums.length - 1; i >= 0; i--) {

console.log(nums[i]);

}

### **Nested for Loops**

A loop inside another loop.

**Example: Print a pattern**

for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

console.log(`i = ${i}, j = ${j}`);

}

}

### **Using break and continue inside for**

* break exits the loop immediately.
* continue skips the current iteration.

**Example: Skip even numbers**

for (let i = 1; i <= 5; i++) {

if (i % 2 === 0) continue;

console.log(i);

}

**Example: Stop loop when** i == 3

for (let i = 1; i <= 5; i++) {

if (i === 3) break;

console.log(i);

}

### **2. while Loop**

Executes while a condition is true.

**Syntax:**

while (condition) {

// Code to execute

}

**Example:** Print numbers from 1 to 5

let i = 1;

while (i <= 5) {

console.log(i);

i++;

}

### **Condition-Based Execution**

Loop runs only if the condition is initially true.

**Example:**

let count = 3;

while (count > 0) {

console.log(count);

count--;

}

### **Infinite Loops & Breaking Conditions**

If the condition never becomes false, it becomes an infinite loop.

**Example:**

while (true) {

console.log("Running...");

break; // Exit to prevent infinite loop

}

## **B. Exit-Controlled Loops**

These loops execute at least once, as the condition is checked after execution.

### **1. do-while Loop**

Executes the code block at least once, even if the condition is false.

**Syntax:**

do {

// Code to execute

} while (condition);

**Example:** Execute at least once

let num = 5;

do {

console.log(num);

num--;

} while (num > 0);

**Real-World Use Cases:**

* User login attempts
* Menu-based programs (ATM, vending machines)
* Data validation

## **C. Looping Over Objects & Arrays**

### **1. for-in Loop**

Used to iterate over object properties.

**Syntax:**

for (key in object) {

// Code to execute

}

**Example:**

const person = { name: "John", age: 30, city: "New York" };

for (let key in person) {

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

}

### **2. for-of Loop**

Used to iterate over iterable objects like arrays, strings, sets, and maps.

**Syntax:**

for (element of iterable) {

// Code to execute

}

**Example:** Iterate through an array

const fruits = ["Apple", "Banana", "Mango"];

for (let fruit of fruits) {

console.log(fruit);

}

**Example:** Iterate through a string

const str = "Hello";

for (let char of str) {

console.log(char);

}

## **D. Loop Control Statements**

Control statements modify loop execution.

### **1. break Statement**

Immediately exits the loop.

**Example:** Stop at 3

for (let i = 1; i <= 5; i++) {

if (i === 3) break;

console.log(i);

}

### **2. continue Statement**

Skips the current iteration and moves to the next.

**Example:** Skip even numbers

for (let i = 1; i <= 5; i++) {

if (i % 2 === 0) continue;

console.log(i);

}

### **3. return Statement**

Exits from a function, stopping loop execution.

**Example:**

function findNumber(arr, target) {

for (let num of arr) {

if (num === target) {

console.log(`Found ${target}`);

return;

}

}

console.log("Number not found");

}

findNumber([10, 20, 30], 20);

### **4. label Statement**

Used to break or continue a specific loop in nested structures.

**Example:** Breaking a specific loop

outerLoop: for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

if (j === 2) break outerLoop;

console.log(`i=${i}, j=${j}`);

}

}

## **Key Takeaways**

✅ for loop is best when the number of iterations is known. ✅ while loop is used when condition-based execution is needed. ✅ do-while guarantees at least one execution. ✅ for-in is used for object properties, for-of for iterable elements. ✅ break stops the loop, continue skips an iteration, return exits a function. ✅ Labels help break specific loops in nested structures.

**JavaScript Loops: Comparison & Differences**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **for Loop** | **while Loop** | **do...while Loop** | **for...in Loop** | **for...of Loop** |
| **Use Case** | When the number of iterations is **known**. | When the number of iterations is **unknown** (based on a condition). | When the loop should execute **at least once**, even if the condition is false. | Used to **iterate over object properties**. | Used to **iterate over iterable objects** like **arrays, strings, Maps, Sets, etc.** |
| **Structure** | for(initialization; condition; increment/decrement) { } | while(condition) { } | do { } while(condition); | for (key in object) { } | for (value of iterable) { } |
| **Condition Check** | **Before** executing the loop body. | **Before** executing the loop body. | **After** executing the loop body. | Iterates over **property names** of an object. | Iterates over **values** of an iterable. |
| **Guaranteed Execution?** | ❌ No, if the condition is false initially, it won’t run. | ❌ No, if the condition is false initially, it won’t run. | ✅ Yes, runs **at least once** even if the condition is false. | ✅ Iterates through **all enumerable properties** of an object. | ✅ Iterates directly over **values** rather than indexes. |
| **Best For?** | Iterating a **fixed number of times**. | Running a loop **until a condition becomes false**. | Ensuring the loop **runs at least once**. | **Looping through object properties**. | **Looping through arrays, strings, Sets, and Maps efficiently**. |
| **Can Iterate Over Arrays?** | ✅ Yes | ✅ Yes | ✅ Yes | ⚠️ **Not Recommended** (gives keys, not values). | ✅ **Best choice for arrays**. |
| **Can Iterate Over Objects?** | ❌ No | ❌ No | ❌ No | ✅ **Yes (for properties)**. | ❌ No (Use Object.values() or Object.entries()). |
| **Performance** | **Fastest** for indexed arrays. | Similar to for, but **condition-based**. | **Slightly slower** than while. | **Slow** for arrays (**includes prototype properties**). | **Faster** than for...in for arrays. |
| **Risk of Infinite Loop?** | ❌ No (if written correctly). | ✅ **Yes** (if condition never changes). | ✅ **Yes** (if condition never changes). | ❌ No. | ❌ No. |

**JavaScript Exception Handling (Error Control)**

### 1. Introduction to Exception Handling

Exception handling in JavaScript is a mechanism to handle runtime errors gracefully and prevent the program from crashing. It allows developers to manage errors and take appropriate actions when an error occurs.

### 2. try...catch...finally Structure

#### **2.1 try Block**

* The try block contains the code that might cause an error.
* If an error occurs, the control jumps to the catch block.

**Example:**

try {

let result = 10 / 0; // No error, but division by zero may cause issues

console.log(result);

} catch (error) {

console.error("An error occurred:", error.message);

}

#### **2.2 catch Block**

* The catch block handles the error thrown in the try block.
* It receives an error object that contains information about the error.

**Example:**

try {

let x = y; // ReferenceError: y is not defined

} catch (error) {

console.error("Error caught:", error.name, error.message);

}

#### **2.3 finally Block**

* The finally block executes regardless of whether an error occurred or not.
* It is useful for cleanup operations like closing files or freeing resources.

**Example:**

try {

let data = JSON.parse('{"name": "John"}');

console.log("Parsed data:", data);

} catch (error) {

console.error("Invalid JSON:", error.message);

} finally {

console.log("Execution completed.");

}

### 3. throw Statement (Manually Throwing Errors)

* The throw statement allows developers to create custom errors.
* It can throw built-in or user-defined errors.

**Example:**

function checkAge(age) {

if (age < 18) {

throw new Error("Age must be 18 or above.");

}

return "Access granted";

}

try {

console.log(checkAge(16));

} catch (error) {

console.error("Caught Exception:", error.message);

}

### 4. Built-in Error Types in JavaScript

#### **4.1 SyntaxError**

Occurs when there is a syntax mistake in the code.

try {

eval("var a =;"); // SyntaxError

} catch (error) {

console.error(error.name, error.message);

}

#### **4.2 ReferenceError**

Occurs when accessing an undefined variable.

try {

console.log(notDefinedVar); // ReferenceError

} catch (error) {

console.error(error.name, error.message);

}

#### **4.3 TypeError**

Occurs when a value is not of the expected type.

try {

let num = 5;

num.toUpperCase(); // TypeError

} catch (error) {

console.error(error.name, error.message);

}

#### **4.4 RangeError**

Occurs when a number is outside its valid range.

try {

let arr = new Array(-1); // RangeError

} catch (error) {

console.error(error.name, error.message);

}

#### **4.5 EvalError** (Rarely used)

Occurs when an error is related to the eval() function.

try {

throw new EvalError("Custom EvalError");

} catch (error) {

console.error(error.name, error.message);

}

#### **4.6 URIError**

Occurs when an invalid URI encoding/decoding operation is performed.

try {

decodeURIComponent('%'); // URIError

} catch (error) {

console.error(error.name, error.message);

}

### 5. Custom Error Handling

JavaScript allows developers to create custom error classes by extending the built-in Error class.

#### **Example: Custom Error Class**

class CustomError extends Error {

constructor(message) {

super(message);

this.name = "CustomError";

}

}

try {

throw new CustomError("This is a custom error!");

} catch (error) {

console.error(error.name, error.message);

}

### 6. Important Points

* Always use try...catch to handle unexpected runtime errors.
* finally block is useful for cleanup operations.
* throw allows for custom error handling.
* Use specific error types for better debugging.
* Create custom error classes for application-specific error handling.

By properly implementing exception handling, developers can write more robust and error-resilient JavaScript applications.

## **Asynchronous Control Flow in JavaScript**

### A. Callbacks (Traditional Approach)

#### 1. Synchronous vs. Asynchronous Execution

* **Synchronous Execution:** Code runs line by line, blocking further execution until the current task completes.
* **Asynchronous Execution:** Code runs in the background without blocking further execution, allowing other tasks to continue while waiting for a response.

#### 2. Callback Functions

* A callback function is a function passed as an argument to another function and executed later.
* Used in setTimeout, event listeners, file reading, database calls, API calls, etc.

**Example: Callback Function**

function greet(name, callback) {

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

callback();

}

function sayGoodbye() {

console.log("Goodbye!");

}

greet("Amol", sayGoodbye);

#### 3. Callback Hell (Pyramid of Doom)

* Occurs when multiple nested callbacks make code unreadable.
* Common in asynchronous operations like file handling, API requests, and database queries.

**Example: Callback Hell**

function step1(callback) {

setTimeout(() => {

console.log("Step 1 completed");

callback();

}, 1000);

}

function step2(callback) {

setTimeout(() => {

console.log("Step 2 completed");

callback();

}, 1000);

}

function step3(callback) {

setTimeout(() => {

console.log("Step 3 completed");

callback();

}, 1000);

}

// Nested Callbacks (Pyramid of Doom)

step1(() => {

step2(() => {

step3(() => {

console.log("All steps completed");

});

});

});

### B. Promises (Modern Approach)

* A Promise represents the eventual completion (or failure) of an asynchronous operation.
* **States of a Promise:**
  + **Pending:** Initial state, operation not completed.
  + **Fulfilled:** Operation completed successfully.
  + **Rejected:** Operation failed.

#### 1. Creating a Promise

let myPromise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve("Promise fulfilled!");

} else {

reject("Promise rejected!");

}

});

#### 2. .then(), .catch(), and .finally() Methods

* .then(): Executes when the promise is resolved.
* .catch(): Executes when the promise is rejected.
* .finally(): Executes regardless of success or failure.

#### 3. Promise Chaining

function fetchData() {

return new Promise((resolve) => {

setTimeout(() => {

resolve("Data fetched");

}, 1000);

});

}

fetchData()

.then((result) => {

console.log(result);

return "Processing data...";

})

.then((processedData) => {

console.log(processedData);

return "Finalizing data...";

})

.then((finalData) => {

console.log(finalData);

})

.catch((error) => {

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

});

### C. Async-Await (Better Readability)

* Introduced in ES8, async-await makes asynchronous code look synchronous.
* async function → Declares a function that returns a promise.
* await → Waits for the result of a promise before moving forward.

#### Using await Inside async Functions

function fetchData() {

return new Promise((resolve) => {

setTimeout(() => resolve("Data received!"), 2000);

});

}

async function fetchAsyncData() {

console.log("Fetching...");

let data = await fetchData();

console.log(data);

}

fetchAsyncData();

#### 2. Error Handling with try-catch in async Functions

function fetchDataWithError() {

return new Promise((resolve, reject) => {

setTimeout(() => reject("Network error!"), 2000);

});

}

async function fetchWithHandling() {

try {

let data = await fetchDataWithError();

console.log(data);

} catch (error) {

console.log("Caught Error:", error);

}

}

fetchWithHandling();

### Important Concepts

#### 1. Parallel Execution with Promise.all()

* Runs multiple promises simultaneously and waits for all to resolve/reject.

let p1 = new Promise((resolve) => setTimeout(() => resolve("One"), 1000));

let p2 = new Promise((resolve) => setTimeout(() => resolve("Two"), 2000));

let p3 = new Promise((resolve) => setTimeout(() => resolve("Three"), 3000));

Promise.all([p1, p2, p3]).then((values) => console.log(values));

#### 2. Handling First Resolved Promise with Promise.race()

* Returns the first resolved or rejected promise.

Promise.race([p1, p2, p3]).then((value) => console.log(value));

#### 3. Promise.any() (ES2021)

* Returns the first fulfilled promise, ignoring rejected ones.

Promise.any([p1, Promise.reject("Error"), p3]).then((value) => console.log(value));

### Summary Table

| **Feature** | **Callbacks** | **Promises** | **Async-Await** |
| --- | --- | --- | --- |
| **Readability** | Poor | Moderate | Best |
| **Error Handling** | Complex | .catch() | try-catch |
| **Multiple Async Calls** | Callback Hell | Promise.all() | await multiple calls |
| **Best Use Case** | Legacy Code | API Calls, Async Operations | Clean & Readable Async Code |

### F. Best Practices

1. **Use Promises instead of Callbacks** to avoid callback hell.
2. **Prefer Async-Await** for better readability and maintainability.
3. **Always handle errors** using .catch() or try-catch.
4. **Use** Promise.all() **for parallel execution** when multiple async tasks are independent.
5. **Use** Promise.race() **or** Promise.any() to manage fast responses in API calls.
6. **Avoid unnecessary** await **statements** that can slow down execution.

### G. Final Thoughts

* **Callbacks** are useful but lead to Callback Hell.
* **Promises** improve async handling and avoid nesting.
* **Async-Await** is the best approach for modern JavaScript due to its simplicity and better error handling.

**Event Loop & Execution Order**

### Call Stack

* The **Call Stack** is a data structure that records function calls in JavaScript.
* JavaScript follows a **LIFO (Last In, First Out)** approach to manage function execution.
* When a function is called, it is pushed onto the stack. Once execution completes, it is popped off.
* If a function calls another function, the new function is added to the top of the stack.

Example:

function first() {

second();

console.log("First function");

}

function second() {

console.log("Second function");

}

first();

* The execution order would be:
  1. first() is pushed onto the stack.
  2. first() calls second(), so second() is pushed onto the stack.
  3. second() executes and is popped off the stack.
  4. Execution returns to first() and logs "First function".
  5. first() completes and is removed from the stack.

### Task Queue (Microtasks & Macrotasks)

* JavaScript has two types of queues: **Macrotask Queue** and **Microtask Queue**.
* **Microtasks**: Promises, Mutation Observers, queueMicrotask().
* **Macrotasks**: setTimeout(), setInterval(), setImmediate(), requestAnimationFrame().
* The **Event Loop** prioritizes executing all microtasks before moving to the next macrotask.
* **Important Point:** If a macrotask adds a microtask, that microtask is executed before moving to the next macrotask.

**Example with Nested Tasks:**

console.log("Start");

setTimeout(() => console.log("Macrotask"), 0);

Promise.resolve().then(() => {

console.log("Microtask 1");

Promise.resolve().then(() => console.log("Microtask 2"));

});

console.log("End");

**Output Order:**

Start

End

Microtask 1

Microtask 2

Macrotask

### Web APIs in JavaScript

* JavaScript in the browser environment relies on **Web APIs** for asynchronous tasks.
* Examples include:
  + **DOM Manipulation API** (document, elements, event listeners)
  + **Timer API** (setTimeout, setInterval)
  + **Fetch API** (fetch() for network requests)
  + **Storage API** (localStorage, sessionStorage)
  + **Geolocation API**
* These APIs operate outside the Call Stack and interact with the Event Loop.

### Event Loop Mechanism

* The **Event Loop** continuously checks the Call Stack and Task Queues.
* If the Call Stack is empty, it takes the first task from the Task Queue and executes it.
* Steps:
  1. The Call Stack executes synchronous code.
  2. Web API tasks (like setTimeout) move to respective queues.
  3. The Event Loop checks for pending microtasks and executes them.
  4. Macrotasks are picked up after microtasks complete.

**Advanced Flow Control Topics**

### A. Short-Circuiting in JavaScript

#### && (Logical AND)

* Returns the first falsy value or the last truthy value.
* Used in conditional execution.

Example:

let user = { loggedIn: true, name: "John" };

user.loggedIn && console.log(user.name); // "John"

#### || (Logical OR)

* Returns the first truthy value or the last falsy value.

Example:

let username = "";

console.log(username || "Guest"); // "Guest"

#### ?? (Nullish Coalescing Operator)

* Returns the right-hand operand only if the left-hand operand is null or undefined.

Example:

let user;

console.log(user ?? "Guest"); // "Guest"

### B. Optional Chaining (?.)

#### Safe Object Property Access

* Prevents errors when accessing deeply nested properties.

Example:

let user = { profile: { name: "John" } };

console.log(user?.profile?.age ?? "No Age"); // "No Age"

### C. Generator Functions (function\* & yield)

#### Definition & Syntax

* A special function that returns an iterator and allows pausing execution using yield.

Example:

function\* generator() {

yield 1;

yield 2;

yield 3;

}

let iter = generator();

console.log(iter.next().value); // 1

console.log(iter.next().value); // 2

console.log(iter.next().value); // 3

#### Use Cases in Asynchronous Programming

* Helps in managing async operations.

Example:

function\* fetchData() {

yield fetch("https://api.example.com/data");

console.log("Data fetched");

}

let fetcher = fetchData();

fetcher.next().value.then(() => fetcher.next());

### D. State Machines

#### Finite State Machines (FSM) for UI Flow Control

* FSM is a design pattern for managing state transitions.

Example FSM:

const trafficLight = {

state: "red",

transition() {

this.state = this.state === "red" ? "green" : "red";

}

};

console.log(trafficLight.state); // red

trafficLight.transition();

console.log(trafficLight.state); // green

#### Handling Complex UI Logic with FSM

* FSMs help in managing UI state changes efficiently.
* Example Use Case:
  + Navigation menus
  + Modal dialog states
  + Form validation workflows
  + Game state management