JavaScript Introduction

JavaScript is a versatile, high-level programming language primarily used to create dynamic and interactive content on websites. It is one of the core technologies of web development, along with HTML and CSS. JavaScript enables websites to go beyond static pages by allowing user interaction, responding to events, manipulating the DOM, and communicating with servers.

**Key Features of JavaScript:**

1. **Client-Side Scripting**:
   * JavaScript runs directly in the user's browser, enabling interactivity without reloading the page.
   * It's widely supported by all modern browsers.
2. **Event-Driven**:
   * JavaScript responds to user interactions (clicks, keypresses, etc.) and browser events (loading, resizing) through event handlers.
3. **Dynamically Typed**:
   * Variables in JavaScript don’t need to be declared with a specific type. The data type is determined at runtime.
   * Example:

javascript

Copy code

let x = 10; // x is a number

x = "Hello"; // Now x is a string

1. **Prototype-Based Object-Oriented**:
   * JavaScript supports object-oriented programming (OOP) through prototypes, meaning objects can inherit properties and methods from other objects.
2. **Asynchronous Programming**:
   * JavaScript handles asynchronous operations (e.g., making network requests) using callbacks, promises, and async/await syntax, allowing for non-blocking execution.
3. **Cross-Platform**:
   * JavaScript can be run on a variety of platforms (browsers, Node.js servers, mobile devices).

**Basic Syntax:**

1. **Variables**: Variables store data and can be declared using var, let, or const:
   * var: Function-scoped (older method).
   * let: Block-scoped, preferred for reassignable variables.
   * const: Block-scoped, used for constants that cannot be reassigned.

Example:

javascript

Copy code

let name = "Alice";

const age = 25;

1. **Data Types**: JavaScript supports several primitive data types:
   * **String**: Text data, e.g., "Hello".
   * **Number**: Integer or floating-point, e.g., 42, 3.14.
   * **Boolean**: Logical values, true or false.
   * **Object**: Complex data structures, e.g., { name: "Alice", age: 25 }.
   * **Array**: List of values, e.g., [1, 2, 3].
   * **Null**: Represents intentional absence of any object value.
   * **Undefined**: Variables declared but not initialized.

Example:

javascript

Copy code

let isLoggedIn = false;

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

let colors = ["red", "blue", "green"];

1. **Operators**: JavaScript has a variety of operators:
   * **Arithmetic**: +, -, \*, /, %.
   * **Assignment**: =, +=, -=, \*=.
   * **Comparison**: ==, ===, !=, !==, >, <.
   * **Logical**: &&, ||, !.

Example:

javascript

Copy code

let a = 10;

let b = 5;

let result = a + b; // 15

1. **Conditional Statements**: JavaScript uses if, else if, and else for decision-making.

javascript

Copy code

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

1. **Loops**: JavaScript supports several types of loops, including for, while, and do...while.

Example of a for loop:

javascript

Copy code

for (let i = 0; i < 5; i++) {

console.log(i); // Outputs: 0, 1, 2, 3, 4

}

**Functions:**

Functions are reusable blocks of code that can take input (parameters) and return an output.

1. **Function Declaration**:

javascript

Copy code

function greet(name) {

return "Hello, " + name;

}

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

1. **Arrow Functions**: A shorter syntax for functions introduced in ES6 (ECMAScript 2015):

javascript

Copy code

const greet = (name) => "Hello, " + name;

1. **Anonymous Functions**: Functions without a name, often used as callbacks.

javascript

Copy code

setTimeout(function() {

console.log("This runs after 2 seconds");

}, 2000);

**DOM Manipulation:**

JavaScript interacts with the Document Object Model (DOM) to change HTML and CSS dynamically.

1. **Selecting Elements**:
   * document.getElementById("id"): Selects an element by its ID.
   * document.querySelector(".class"): Selects the first element that matches the CSS selector.

Example:

javascript

Copy code

let element = document.getElementById("myElement");

let button = document.querySelector(".myButton");

1. **Modifying Elements**:
   * element.textContent: Changes the text inside an element.
   * element.style: Changes the CSS of an element.
   * element.setAttribute(): Modifies the attributes of an element.

Example:

javascript

Copy code

document.getElementById("title").textContent = "New Title";

document.querySelector(".box").style.backgroundColor = "blue";

1. **Event Handling**: JavaScript listens for user actions (clicks, keypresses) and responds using event listeners.

Example:

javascript

Copy code

let button = document.querySelector(".myButton");

button.addEventListener("click", function() {

alert("Button clicked!");

});

**JavaScript Events:**

JavaScript can detect and respond to various events, such as:

* **click**: When an element is clicked.
* **submit**: When a form is submitted.
* **load**: When the webpage has finished loading.

Exampl e:

javascript

Copy code

window.addEventListener("load", function() {

console.log("Page loaded");

});

**Asynchronous JavaScript:**

1. **Callbacks**: A function passed as an argument to another function and executed after some operation completes.

javascript

Copy code

function fetchData(callback) {

setTimeout(() => {

callback("Data received");

}, 1000);

}

fetchData(function(data) {

console.log(data); // Outputs "Data received" after 1 second

});

1. **Promises**: Used to handle asynchronous operations. Promises can either be resolved (successful) or rejected (failed).

javascript

Copy code

let promise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve("Operation successful");

} else {

reject("Operation failed");

}

});

promise.then((message) => {

console.log(message);

}).catch((error) => {

console.error(error);

});

1. **async and await**: Syntax sugar for handling promises in a more readable way.

javascript

Copy code

async function fetchData() {

let data = await fetch("https://api.example.com/data");

let json = await data.json();

console.log(json);

}

**JavaScript Frameworks and Libraries:**

JavaScript can be extended with libraries and frameworks to make development easier and faster:

* **jQuery**: A popular library for simplifying DOM manipulation and event handling.
* **React**: A JavaScript library for building user interfaces, particularly single-page applications.
* **Angular** and **Vue.js**: Frameworks for building complex web applications.

**Type coercion** in JavaScript refers to the automatic or implicit conversion of values from one data type to another. This happens when operators or functions expect a certain data type, and JavaScript converts the given value to a suitable type to make the operation work.

**Types of Coercion**

1. **Implicit Coercion**: Happens automatically when JavaScript tries to perform operations on values of different types.
2. **Explicit Coercion**: Happens when the developer intentionally converts a value from one type to another using functions like **Number(), String(), or Boolean().**

**1. Implicit Coercion**

JavaScript automatically converts one type to another based on the context (whether it's a string, number, or boolean). This can sometimes lead to unexpected results.

**Examples:**

* **String Coercion**: When you use the + operator with a string and another data type, JavaScript converts the other type to a string and concatenates them.

javascript

Copy code

let result = ‘5’ + 10; // "510"

let output = 'Hello' + true; // "Hellotrue"

* **Number Coercion**: When non-numeric values are used in mathematical operations (other than +), JavaScript converts them to numbers.

javascript

Copy code

let result = '5' \* 2; // 10 ('5' is coerced to 5)

let result2 = '10' - 2; // 8 ('10' is coerced to 10)

let invalid = 'abc' \* 2; // NaN (cannot convert 'abc' to a number)

* **Boolean Coercion**: JavaScript automatically converts values to true or false in certain contexts, like conditionals (if, while loops).

Values that are coerced to false are called **falsy** values:

* + false, 0, "" (empty string), null, undefined, and NaN.

All other values are **truthy**.

javascript

Copy code

if (0) {

console.log("This will not be printed"); // 0 is falsy

}

if ("Hello") {

console.log("This will be printed"); // non-empty string is truthy

}

**2. Explicit Coercion**

This is when you manually convert a value from one type to another using built-in methods like Number(), String(), or Boolean().

**Examples:**

* **Converting to String**: Use String() or .toString() to explicitly convert a value to a string.

javascript

Copy code

let num = 123;

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

let str2 = num.toString(); // "123"

* **Converting to Number**: Use Number(), parseInt(), or parseFloat() to convert a string to a number.

javascript

Copy code

let str = "42";

let num = Number(str); // 42

let intNum = parseInt("10.5"); // 10

let floatNum = parseFloat("10.5"); // 10.5

* **Converting to Boolean**: Use Boolean() to convert any value to a boolean.

javascript

Copy code

let isTrue = Boolean(1); // true

let isFalse = Boolean(0); // false

let isTruthy = Boolean("Hello"); // true (non-empty string)

let isFalsy = Boolean(""); // false (empty string)

**Coercion in Comparisons**

* **Loose Equality (==)**: The == operator performs type coercion before comparing values. This means it tries to convert the operands to the same type before checking equality.

javascript

Copy code

5 == '5'; // true ('5' is coerced to 5)

null == undefined; // true (both are coerced to a falsy value)

0 == false; // true (0 is coerced to false)

* **Strict Equality (===)**: The === operator does **not** perform type coercion. Both the type and the value must be the same for the comparison to return true.

javascript

Copy code

5 === '5'; // false (no coercion, different types)

null === undefined; // false (different types)

0 === false; // false (different types)

**Coercion Examples:**

**String Coercion in Concatenation:**

javascript

Copy code

let a = 10;

let b = "20";

let result = a + b; // "1020"

Here, a is converted to a string and concatenated with b.

**Implicit Number Coercion:**

javascript

Copy code

let x = "5" \* 2; // 10 ('5' is coerced to 5)

let y = "10" - 5; // 5 ('10' is coerced to 10)

let z = "10" / "2"; // 5 (both strings are coerced to numbers)

**Potential Pitfalls of Type Coercion:**

1. **Unintended Results**: Implicit coercion can sometimes lead to unexpected results, especially in comparisons or arithmetic operations.

javascript

Copy code

let x = null + 5; // 5 (null is coerced to 0)

let y = true + 3; // 4 (true is coerced to 1)

let z = "5" - 1; // 4 ('5' is coerced to 5)

1. **Loose Equality (==) vs Strict Equality (===)**: Using == can lead to confusing results because of automatic type coercion.

javascript

Copy code

0 == false; // true (0 is coerced to false)

0 === false; // false (different types: number vs boolean)

Variable Scope

In JavaScript, **variable scope** defines the accessibility or visibility of variables within different parts of the code. Variables can have different scopes based on where they are declared, and these scopes determine where the variables can be used. There are primarily three types of scopes:

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

**1. Global Scope**

A variable declared outside any function or block becomes globally scoped. These variables are accessible throughout the entire script or program, including inside functions or blocks.

**Example:**

javascript

Copy code

let globalVar = "I am global"; // Globally scoped

function exampleFunction() {

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

}

exampleFunction(); // Output: "I am global"

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

**Key Points:**

* **Global variables** can be accessed and modified from anywhere in the program.
* In a browser environment, variables declared with var in the global scope become properties of the window object. Variables declared with let and const do not.

**2. Function Scope**

Variables declared inside a function using var, let, or const are **function-scoped**. These variables are only accessible within the function where they are defined.

**Example:**

javascript

Copy code

function exampleFunction() {

let functionVar = "I am inside a function"; // Function-scoped

console.log(functionVar); // Accessible here

}

exampleFunction(); // Output: "I am inside a function"

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

**Key Points:**

* **Function-scoped variables** cannot be accessed outside the function where they are declared.
* Variables declared with var are function-scoped, even if declared inside a block (like if or for statements).

**3. Block Scope**

Variables declared with let or const inside a block (such as an if statement, for loop, or curly braces {}) are **block-scoped**. Block-scoped variables can only be accessed within that specific block.

**Example:**

javascript

Copy code

if (true) {

let blockScopedVar = "I am inside a block"; // Block-scoped

console.log(blockScopedVar); // Accessible here

}

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

**Key Points:**

* **Block-scoped variables** (declared with let or const) cannot be accessed outside of the block in which they were declared.
* var does not follow block scope rules, and it gets hoisted to the function or global scope, depending on where it’s declared.

**Block Scope in Loops:**

javascript

Copy code

for (let i = 0; i < 3; i++) {

console.log(i); // Accessible inside the loop

}

console.log(i); // Error: i is not defined (because i is block-scoped)

**4. Differences Between var, let, and const**

* var is **function-scoped** and can be redeclared. It does not follow block scope rules.
* let is **block-scoped** and cannot be redeclared within the same scope. It's preferred over var because it avoids hoisting-related issues.
* const is also **block-scoped**, but it creates a **constant** variable whose value cannot be reassigned.

**Example with var, let, and const:**

javascript

Copy code

function example() {

if (true) {

var varVariable = "var: function-scoped";

let letVariable = "let: block-scoped";

const constVariable = "const: block-scoped";

}

console.log(varVariable); // Accessible: "var: function-scoped"

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

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

}

example();

**5. Lexical Scope**

JavaScript uses **lexical (static) scoping**, meaning the accessibility of variables is determined by their location in the source code (when the code is written) rather than during runtime. Functions can access variables that were defined in their outer scope.

**Example:**

javascript

Copy code

function outerFunction() {

let outerVar = "I am outside";

function innerFunction() {

console.log(outerVar); // Can access outerVar from outerFunction

}

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

}

outerFunction();

**Key Points:**

* **Inner functions** can access variables from their outer (enclosing) function scope, but outer functions cannot access variables inside the inner functions.

**6. Scope Chain**

When JavaScript looks for a variable, it starts from the current scope and moves outward through the scope chain, checking parent scopes until it finds the variable or reaches the global scope.

**Example:**

javascript

Copy code

let globalVar = "Global";

function outerFunction() {

let outerVar = "Outer";

function innerFunction() {

let innerVar = "Inner";

console.log(globalVar); // Can access globalVar

console.log(outerVar); // Can access outerVar

console.log(innerVar); // Can access innerVar

}

innerFunction();

}

outerFunction();

Here, innerFunction() can access globalVar, outerVar, and innerVar, but code in the global scope cannot access innerVar or outerVar.

**7. Hoisting**

JavaScript hoists the declaration of variables and functions to the top of their scope before executing the code. However, only the declarations are hoisted, not the initializations.

**Example of Hoisting:**

javascript

Copy code

console.log(x); // Undefined (due to hoisting, x is declared but not initialized)

var x = 5;

Variables declared with var are hoisted, but their values are not assigned until the line where they are initialized is executed. Variables declared with let and const are hoisted, but they are not initialized and cannot be used before their declaration.

Datatypes

In JavaScript, **datatypes** define the types of values that variables can hold. JavaScript datatypes are divided into two main categories: **Primitive Types** and **Non-Primitive (Reference) Types**.

**1. Primitive Data Types**

Primitive types represent simple values and are immutable (they cannot be changed).

**a. Number**

Represents numeric values, including both integers and floating-point numbers.

javascript

Copy code

let age = 25;

let price = 99.99;

* Special numeric values: Infinity, -Infinity, and NaN (Not a Number).

**b. String**

Represents a sequence of characters. Strings are created using single ('), double ("), or backtick (`) quotes.

javascript

Copy code

let name = "John";

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

**c. Boolean**

Represents two values: true or false. Used for logical operations and condition checks.

javascript

Copy code

let isLoggedIn = true;

let hasPermission = false;

**d. Null**

Represents the intentional absence of any value. It signifies an empty or non-existent reference.

javascript

Copy code

let emptyValue = null;

**e. Undefined**

Indicates a variable that has been declared but has not yet been assigned a value.

javascript

Copy code

let uninitializedVar;

console.log(uninitializedVar); // undefined

**f. Symbol (ES6)**

A unique and immutable value, primarily used to create unique object keys.

javascript

Copy code

let sym = Symbol('id');

**g. BigInt (ES2020)**

Represents large integers that are greater than Number.MAX\_SAFE\_INTEGER (2^53 - 1). It’s used for very large numbers.

javascript

Copy code

let bigInt = BigInt(9007199254740991);

**2. Non-Primitive (Reference) Data Types**

Non-primitive types, also called **objects**, are mutable and can store collections of data or more complex entities. Examples include:

**a. Object**

Objects are collections of key-value pairs. Values in objects can be of any type, including other objects.

javascript

Copy code

let person = {

name: 'Alice',

age: 30,

greet: function() {

return `Hello, I'm ${this.name}`;

}

};

**b. Array**

Arrays are a type of object used to store ordered lists of values.

javascript

Copy code

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

**c. Function**

Functions are reusable blocks of code. In JavaScript, functions are first-class objects, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

javascript

Copy code

function sayHello() {

return 'Hello!';

}

**d. Date**

Represents date and time.

javascript

Copy code

let today = new Date();

**e. RegExp**

Represents regular expressions, used for pattern matching in strings.

javascript

Copy code

let regex = /hello/i; // Case-insensitive match for 'hello'

**Type Checking**

To check the type of a variable, JavaScript provides the typeof operator:

javascript

Copy code

console.log(typeof 42); // "number"

console.log(typeof 'Hello'); // "string"

console.log(typeof true); // "boolean"

console.log(typeof undefined); // "undefined"

console.log(typeof null); // "object" (This is a known quirk in JavaScript)

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

console.log(typeof {}); // "object"

console.log(typeof []); // "object" (arrays are objects in JavaScript)

**Key Differences:**

1. **Primitive Types** are **immutable**, meaning their values cannot be changed. For example, modifying a string creates a new string.
2. **Non-Primitive Types** (objects) are **mutable**, meaning you can modify their content without changing the reference.

Template Literals

**Template literals** are a powerful feature in JavaScript that allow for easier string creation and manipulation. They provide a more readable and flexible way to work with strings compared to traditional string concatenation. Template literals are enclosed by backticks (`) and can contain placeholders, which are indicated by the dollar sign and curly braces (${expression}).

**Key Features of Template Literals**

1. **Multiline Strings** Template literals can span multiple lines without needing escape characters.

javascript

Copy code

let message = `This is a string

that spans multiple

lines.`;

console.log(message);

1. **String Interpolation** You can embed expressions directly within the string using ${}. This allows you to include variables and expressions without the need for concatenation.

javascript

Copy code

let name = 'Alice';

let age = 25;

let greeting = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(greeting); // Output: Hello, my name is Alice and I am 25 years old.

1. **Expression Evaluation** Any valid JavaScript expression can be placed within the ${} placeholder, including function calls.

javascript

Copy code

function getGreeting() {

return 'Welcome!';

}

let message = `${getGreeting()} My name is Bob.`;

console.log(message); // Output: Welcome! My name is Bob.

1. **Tagged Templates** You can create a function (tag) that processes a template literal. This allows for custom formatting or processing of the string.

javascript

Copy code

function tag(strings, ...values) {

let result = '';

strings.forEach((str, index) => {

result += str + (values[index] ? `<strong>${values[index]}</strong>` : '');

});

return result;

}

let name = 'Alice';

let formattedMessage = tag`Hello, ${name}! Welcome to the site.`;

console.log(formattedMessage); // Output: Hello, <strong>Alice</strong>! Welcome to the site.

**Benefits of Template Literals**

* **Readability**: They make the code cleaner and easier to read, especially when constructing strings with multiple lines or complex expressions.
* **Convenience**: Eliminates the need for cumbersome concatenation using the + operator.
* **Flexibility**: Allows for dynamic string creation with expressions embedded directly.

Strict Mode

**Strict Mode** is a feature in JavaScript that helps you write cleaner, more secure code by enforcing stricter parsing and error handling. It was introduced in ECMAScript 5 (ES5) and can be activated on a per-function or per-script basis. Using strict mode can help catch common coding mistakes and "unsafe" actions.

**How to Enable Strict Mode**

Strict mode can be enabled by adding the directive "use strict"; at the beginning of a script or a function.

**Global Strict Mode**

To apply strict mode to an entire script:

javascript

Copy code

"use strict";

let x = 3.14; // Strict mode is enabled for the entire script

**Function-Specific Strict Mode**

To apply strict mode only within a specific function:

javascript

Copy code

function myFunction() {

"use strict";

let y = 3.14; // Strict mode is enabled only within this function

}

**Benefits of Strict Mode**

1. **Eliminates Silent Errors**: In strict mode, errors that would normally fail silently (e.g., assignments to undeclared variables) throw an error, making it easier to identify problems in your code.

javascript

Copy code

"use strict";

undeclaredVariable = 5; // Throws ReferenceError: undeclaredVariable is not defined

1. **Prevents Variable Hoisting Issues**: Variables must be declared before use, reducing confusion caused by hoisting.

javascript

Copy code

"use strict";

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

let x = 3;

1. **Disallows Duplicate Parameter Names**: Strict mode prevents function parameters from having the same name, which can lead to unexpected behavior.

javascript

Copy code

function myFunction(a, a) { // Throws SyntaxError: Duplicate parameter name not allowed in this context

return a;

}

1. **Makes this Work Differently**: In strict mode, this in a function that is called without a context (e.g., someFunction()) will be undefined instead of the global object (e.g., window in browsers).

javascript

Copy code

"use strict";

function myFunction() {

console.log(this); // undefined

}

myFunction();

1. **Disallows Certain Syntax**: Strict mode disallows some syntax that is considered bad practice, such as using with statements, which can create ambiguous code.

javascript

Copy code

"use strict";

with (Math) { // Throws SyntaxError: Strict mode code may not include a with statement

console.log(sqrt(16));

}

**Limitations of Strict Mode**

* **Not Everywhere**: Some older libraries or code may not be compatible with strict mode, which can lead to issues if strict mode is applied globally.
* **Performance**: While strict mode can improve error handling and code clarity, there might be a negligible performance impact due to stricter parsing.

Spread & Rest Operators

The **spread** (...) and **rest** (...) operators in JavaScript are syntactically identical but serve different purposes depending on how they are used.

**Spread Operator**

The spread operator allows you to expand or "spread" elements of an iterable (like an array or object) into individual elements. It is often used in function calls, array literals, and object literals.

**Usage Examples:**

1. **Function Calls** You can use the spread operator to pass elements of an array as separate arguments to a function.

javascript

Copy code

function sum(a, b, c) {

return a + b + c;

}

const numbers = [1, 2, 3];

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

1. **Array Literals** You can create a new array by combining existing arrays or adding new elements.

javascript

Copy code

const arr1 = [1, 2];

const arr2 = [3, 4];

const combined = [...arr1, ...arr2]; // [1, 2, 3, 4]

1. **Object Literals** You can use the spread operator to create a new object by copying properties from an existing object.

javascript

Copy code

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

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

**Rest Operator**

The rest operator **collects multiple elements into a single array**. It is used in function parameter lists to gather remaining arguments into an array.

**Usage Examples:**

1. **Function Parameters** You can use the rest operator to accept a variable number of arguments in a function.

javascript

Copy code

function multiply(...args) {

return args.reduce((acc, val) => acc \* val, 1);

}

console.log(multiply(2, 3, 4)); // Output: 24

1. **Destructuring Assignment** You can use the rest operator to collect remaining properties from an object or elements from an array during destructuring.

javascript

Copy code

const array = [1, 2, 3, 4, 5];

const [first, second, ...rest] = array;

console.log(first); // Output: 1

console.log(rest); // Output: [3, 4, 5]

javascript

Copy code

const person = { name: 'Alice', age: 25, city: 'Wonderland' };

const { name, ...restInfo } = person;

console.log(name); // Output: Alice

console.log(restInfo); // Output: { age: 25, city: 'Wonderland' }

Arrow Functions

**Arrow functions** are a concise way to write function expressions in JavaScript. Introduced in ES6 (ECMAScript 2015), arrow functions provide a shorter syntax compared to traditional function expressions and have some unique features, particularly in how they handle the this keyword.

**Syntax**

Arrow functions use the => syntax. Here are the basic forms:

1. **Basic Syntax**

javascript

Copy code

const functionName = (parameters) => {

// function body

};

1. **Single Parameter (No Parentheses)** If there’s only one parameter, parentheses are optional:

javascript

Copy code

const square = x => x \* x;

1. **Implicit Return** If the function body consists of a single expression, you can omit the braces {} and the return statement:

javascript

Copy code

const add = (a, b) => a + b; // Implicit return

1. **Multiple Parameters** When there are multiple parameters, you must use parentheses:

javascript

Copy code

const multiply = (a, b) => a \* b;

1. **No Parameters** If there are no parameters, use empty parentheses:

javascript

Copy code

const sayHello = () => console.log("Hello!");

**Features of Arrow Functions**

1. **Lexical this Binding** Arrow functions do not have their own this context; they inherit this from the surrounding lexical context. This is particularly useful in scenarios like callbacks and methods in classes.

javascript

Copy code

function Person() {

this.age = 0;

setInterval(() => {

this.age++; // 'this' refers to the Person instance

console.log(this.age);

}, 1000);

}

const person = new Person(); // Outputs: 1, 2, 3, ...

1. **Cannot be Used as Constructors** Arrow functions cannot be used with the new keyword to create instances because they do not have a prototype.

javascript

Copy code

const Foo = () => {};

const instance = new Foo(); // TypeError: Foo is not a constructor

1. **No arguments Object** Arrow functions do not have their own arguments object. If you need to access the arguments passed to an arrow function, you can use the rest parameter syntax instead:

javascript

Copy code

const sum = (...args) => {

return args.reduce((acc, val) => acc + val, 0);

};

1. **Not Suitable for Methods** When defining methods in objects, you may want to use regular function syntax to preserve the expected this behavior. Arrow functions are not recommended for methods that are intended to be called with an object context.

javascript

Copy code

const obj = {

value: 10,

getValue: () => this.value // 'this' does not refer to obj

};

console.log(obj.getValue()); // Output: undefined

Errors

In JavaScript, errors occur when something goes wrong during the execution of the code. JavaScript provides several types of errors, along with mechanisms to handle them effectively.

**Common Types of Errors:**

1. **SyntaxError**
   * Occurs when there is a mistake in the syntax of the code.
   * Example:

javascript

Copy code

console.log("Hello) // Missing closing quote, causes a SyntaxError

* + **Error message**: Unexpected string

1. **ReferenceError**
   * Occurs when a non-existent variable is referenced.
   * Example:

javascript

Copy code

console.log(nonExistentVariable); // `nonExistentVariable` is not defined

* + **Error message**: nonExistentVariable is not defined

1. **TypeError**
   * Occurs when a value is not of the expected type or when a method is called on an incompatible data type.
   * Example:

javascript

Copy code

let num = 42;

num.toUpperCase(); // You can't call toUpperCase() on a number, causes a TypeError

* + **Error message**: num.toUpperCase is not a function

1. **RangeError**
   * Occurs when a number is outside of its allowed range, such as when using an invalid length for an array or calling Number.toPrecision() with an invalid value.
   * Example:

javascript

Copy code

let arr = new Array(-1); // Invalid array length

* + **Error message**: Invalid array length

1. **URIError**
   * Occurs when encodeURI(), decodeURI(), encodeURIComponent(), or decodeURIComponent() is used incorrectly.
   * Example:

javascript

Copy code

decodeURIComponent('%'); // Invalid URI character

* + **Error message**: URI malformed

1. **EvalError**
   * Occurs in relation to the eval() function, which executes a string of JavaScript code. This error type is rare in modern JavaScript, as improper use of eval() is discouraged.
   * Example:

javascript

Copy code

eval('foo bar'); // Invalid code in eval

* + **Error message**: Unexpected identifier (though EvalError is rarely thrown directly)

**Custom Errors:**

You can throw your own errors using the throw statement. This is useful when you want to explicitly create an error in your code.

javascript

Copy code

function checkAge(age) {

if (age < 18) {

throw new Error("You must be at least 18 years old.");

}

}

checkAge(15); // Throws a custom error

**Error Handling with try...catch:**

JavaScript allows you to handle errors gracefully using the try...catch statement.

* **try block**: Contains code that might throw an error.
* **catch block**: Executes if an error occurs in the try block.
* **finally block**: (optional) Executes code after the try/catch regardless of the outcome.

javascript

Copy code

try {

let result = riskyFunction();

console.log(result);

} catch (error) {

console.error("An error occurred:", error.message);

} finally {

console.log("Execution completed.");

}

Spread and Rest operators

The **spread** and **rest** operators in JavaScript are represented by the three dots (...). They look the same but are used in different contexts, serving different purposes:

1. **Spread Operator (...)**: Expands or "spreads" an iterable (like an array or object) into individual elements.
2. **Rest Operator (...)**: Collects multiple elements into a single array (or object).

**1. Spread Operator (...)**

The **spread operator** allows an iterable (e.g., arrays, objects, strings) to be expanded into individual elements.

**Examples of Spread Operator:**

**In Arrays:**

* Spreads the elements of an array into individual elements.

javascript

Copy code

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

// Combine arrays using spread

const combined = [...arr1, ...arr2];

console.log(combined); // [1, 2, 3, 4, 5, 6]

// Copying an array

const copy = [...arr1];

console.log(copy); // [1, 2, 3]

**In Function Calls:**

* Spreads an array into individual arguments when passing to a function.

javascript

Copy code

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers)); // 6

**In Objects:**

* Spreads properties of an object into another object, creating a shallow copy.

javascript

Copy code

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

const job = { title: "Engineer", salary: 50000 };

// Merge objects using spread

const employee = { ...person, ...job };

console.log(employee); // { name: "Alice", age: 25, title: "Engineer", salary: 50000 }

**2. Rest Operator (...)**

The **rest operator** collects multiple elements and "bundles" them into an array (or, in the case of objects, into an object).

**Examples of Rest Operator:**

**In Function Parameters:**

* Gathers all remaining arguments into a single array. This is particularly useful for functions that accept variable numbers of arguments.

javascript

Copy code

function sum(...numbers) {

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

}

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

console.log(sum(5, 10)); // 15

**In Arrays:**

* Rest operator can be used to collect the "rest" of the items in an array.

javascript

Copy code

const [first, second, ...rest] = [1, 2, 3, 4, 5];

console.log(first); // 1

console.log(second); // 2

console.log(rest); // [3, 4, 5]

**In Objects:**

* Similar to arrays, the rest operator can be used to extract some properties and collect the rest into a new object.

javascript

Copy code

const user = { id: 1, name: "John", age: 30, country: "USA" };

const { name, ...details } = user;

console.log(name); // "John"

console.log(details); // { id: 1, age: 30, country: "USA" }

**Key Differences Between Spread and Rest:**

* **Spread** is used to "spread" elements from an iterable (e.g., arrays, objects) into individual elements.
* **Rest** is used to "collect" multiple elements into a single array or object.

**Usage of Spread and Rest in Various Scenarios:**

**Spread in Arrays:**

* Useful for copying arrays, combining arrays, or passing array elements as function arguments.

**Rest in Functions:**

* Useful when you want a function to accept any number of arguments.

**Spread in Objects:**

* Great for merging objects or copying an object, especially in cases where immutability is important (such as in React).

**Rest in Object Destructuring:**

* Allows you to extract a few specific properties from an object and group the rest into a separate object.

**Practical Examples:**

**Combining Objects (Spread in Objects):**

javascript

Copy code

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

const job = { title: "Engineer", company: "Tech Co" };

const fullProfile = { ...user, ...job };

console.log(fullProfile);

// { name: "Alice", age: 25, title: "Engineer", company: "Tech Co" }

**Handling Variable Arguments (Rest in Functions):**

javascript

Copy code

function multiply(multiplier, ...numbers) {

return numbers.map(num => num \* multiplier);

}

console.log(multiply(2, 1, 2, 3)); // [2, 4, 6]

**Using Spread in Function Calls:**

javascript

Copy code

const arr = [1, 2, 3];

console.log(Math.max(...arr)); // 3

**Extracting Properties (Rest in Object Destructuring):**

javascript

Copy code

const person = { name: "Bob", age: 30, country: "Canada" };

const { name, ...info } = person;

console.log(name); // "Bob"

console.log(info); // { age: 30, country: "Canada" }

**Conclusion:**

* **Spread operator** expands elements or properties.
* **Rest operator** collects remaining elements into a new array or object.

**Bubbling and Capturing**

In web development, **bubbling** and **capturing** refer to two phases in event propagation within the Document Object Model (DOM). They describe how events move through the DOM hierarchy when an event is triggered.

**1. Capturing (Event Capture Phase):**

* This phase occurs first.
* The event starts at the root of the DOM and propagates downwards towards the target element (from parent to child).
* Any event listeners set for capturing will trigger as the event moves down the DOM hierarchy.
* To set an event listener in the capturing phase, the third parameter of the addEventListener() method should be true (e.g., element.addEventListener(event, handler, true)).

**2. Bubbling (Event Bubbling Phase):**

* This phase happens after capturing.
* The event starts at the target element and bubbles up through its ancestors (from child to parent) until it reaches the root of the DOM.
* This is the most commonly used phase for event handling.
* By default, event listeners in JavaScript use event bubbling. You can attach a listener using element.addEventListener(event, handler).

**Example:**

Suppose you have nested elements like:

html

Copy code

<div id="parent">

<button id="child">Click me!</button>

</div>

When you click on the button (#child), the event can travel from #parent to #child (capturing phase) and then back up from #child to #parent (bubbling phase).

**Stopping Propagation:**

* **event.stopPropagation()**: Stops the event from propagating further in either direction (capture or bubble).
* **event.stopImmediatePropagation()**: Stops the event from propagating and prevents any other event listeners on the same element from firing.

These mechanisms allow developers to control how events flow through a page's structure and respond to user actions at different points in the hierarchy.