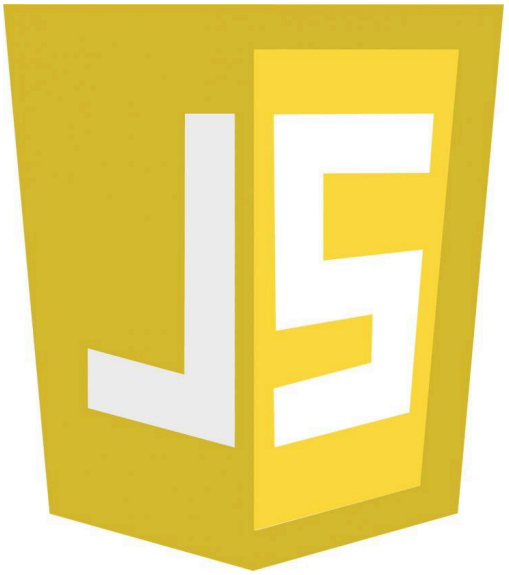
What is JavaScript?

JavaScript, abbreviated as JS, is a lightweight, dynamic programming language essential for modern web development. It is used by developers to create interactive and dynamic website features, enhancing user engagement and responsiveness. This scripting language is commonly used for the development of web pages and various non-browser environments.

In this first chapter of our JavaScript Tutorial, we will understand the **definition of JavaScript**, its features, benefits, applications, and other basic details required to get started with JS programming.



What is Javascript Language?

JavaScript is a programming language that's widely used to create interactive effects within web browsers. It's a key technology for web development, alongside HTML and CSS. JavaScript allows developers to build dynamic websites, handle events (like button clicks), manipulate webpage content, and communicate with servers. It's versatile and can be used on both the client side (in the browser) and the server side (with environments like Node.js).

As a high-level, interpreted programming language, JS is used to build and enhance interactive elements on web pages. It allows developers to manipulate web content, respond to user actions, and communicate with servers in real-time. Initially designed for client-side scripting, JavaScript has expanded to support server-side development through frameworks like Node.js, making it a comprehensive tool for an array of applications, from dynamic websites to mobile apps and server-side services.

As we go through **JavaScript introduction**, it is essential to know that JS is a weakly typed language and can be used for server-side and client-side development. It is imperative and declarative type language with a standard library of objects, such as array, math, date, and a set of language elements like control structures, operators, and statements.

- For server-side development, JS provides objects to run JavaScript on a server. The server-side extension allows an application to interact with the database, provide information to continue from one invocation to another of the app and manipulate a file on the server. Node.js is the most popular framework for server-side development.
- For client-side development, JS offers objects to control a browser and its DOM (Document Object Model). The client-side extension lets an application place elements on the HTML form and respond to user events, such as form input, mouse clicks, and page navigation. The most popular JS libraries for client-side development are ReactJS, AngularJS, and VueJS.

How to Link JavaScript in an HTML File?

We can add JavaScript to an HTML file in the following ways:

Internal JS- Directly adding JS to the HTML file by writing the code within the <script> tag, which can be placed inside the <head> or the <body> tag based on the requirement.

External JS- Write JS code in another file with the extension .js and link that file within the <head> tag of the HTML file in which we wish to add the code.

Syntax of Javascript

Declare a variable:

```
var x;
```

Initialize/Assign a value to the variable:

```
x =10
```

Compute a value of variables

```
x= y;
```

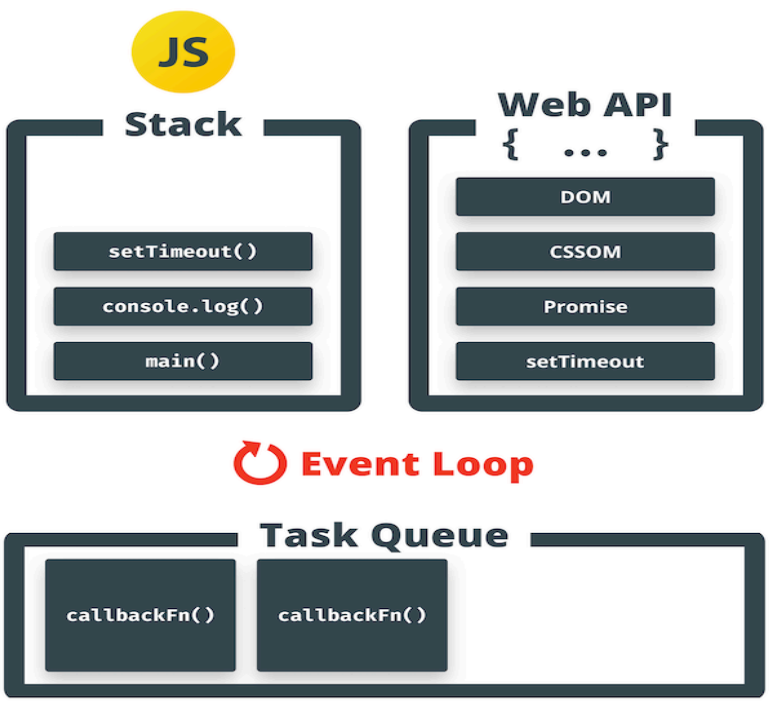
```
z = x / y;
```

Key Features of Javascript

The key features of Javascript are as discussed below.

Javascript is Lightweight: JavaScript is considered a lightweight programming language due to its simple core design –**Just-In-Time (JIT) Compilation**. This means the compilation happens just as it's about to run. These features allow JavaScript to execute code quickly and handle asynchronous operations without extensive resource overhead, making it ideal for enhancing user interactions on websites and web applications.

Javascript is Single-threaded: JavaScript is single-threaded, which means it executes code one operation at a time. This single-threaded nature is primarily due to JavaScript's use of the **event loop** and its execution model. Here's a quick rundown of how it works:



- **Event Loop**: JavaScript uses an event loop to manage the execution of code. The event loop allows JavaScript to perform non-blocking operations, like handling asynchronous tasks, by delegating them to the web APIs or other environments (e.g., Node.js). When these operations are complete, their callbacks are added to the event queue.
- **Call Stack**: The call stack is where JavaScript keeps track of function calls. Each function call is added to the stack and executed in order. When a function finishes, it

is removed from the stack. Since JavaScript executes code from the stack one at a time, it is considered single-threaded.

- **Asynchronous Operations**: Although JavaScript is single-threaded, it can handle asynchronous operations through mechanisms like callbacks, promises, and async/await. These operations don't block the main thread. Instead, they are handled by the event loop, which schedules their callbacks to be executed once the main thread is free.
- **Web APIs and Worker Threads**: In a browser environment, Web APIs (like setTimeout, fetch, and DOM events) handle tasks outside the main thread. Similarly, in Node.js, you have worker threads for concurrent operations. These allow JavaScript to perform tasks without blocking the main thread, but the main thread itself is still single-threaded.

So, while JavaScript can handle many tasks asynchronously, the execution of JavaScript code itself remains single-threaded, meaning only one piece of code executes at a time on the main thread.

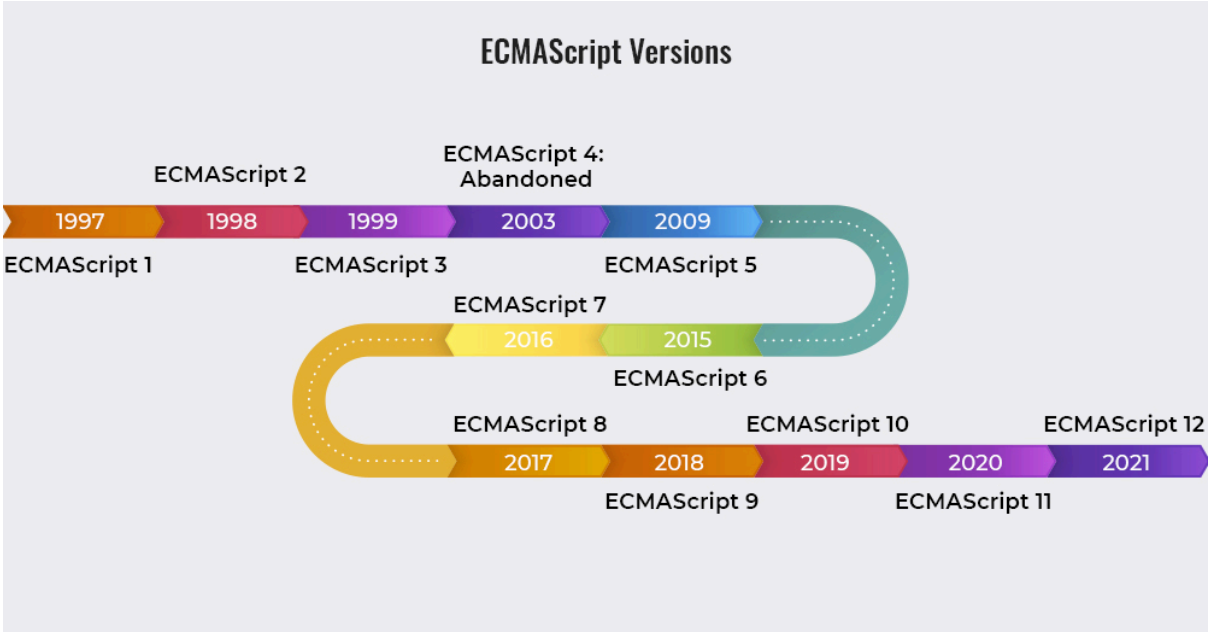
Dynamic Typing: Dynamic Typing is a fundamental feature of JavaScript that allows variables to hold values of any type and change types at runtime.

To elaborate, in other languages like Java and Typescript, we need to declare the type of the value we will be assigning to a variable. (i.e, we cannot assign any other datatype to that particular datatype).

Event Handling: Event handling in JavaScript involves responding to user actions or system events, such as clicks, key presses, or form submissions. This capability is crucial for building responsive web applications, where the interface can update instantly based on user interactions like clicks, typing, or mouse movements. Unlike server-side languages, JavaScript operates directly in the browser, making it especially effective for managing real-time events without requiring page reloads.

Javascript(ECMA) - Versions

JavaScript, standardized under ECMAScript (often abbreviated as ES), defines the core features and syntax of JavaScript, ensuring consistency across different implementations and environments.



ECMAScript was developed and maintained by the ECMA International Organization from the year 1997. Each ECMAScript version is identified by a version number, such as ES5, ES6, ES7, etc.

However, Starting with ECMAScript 2015, the specification began using a year-based naming convention to align with the year of publication. For instance:

- **ECMAScript 2015** (formerly ES6)
- **ECMAScript 2016** (ES7)
- **ECMAScript 2017** (ES8)
- **ECMAScript 2018** (ES9)
- **ECMAScript 2019** (ES10)
- **ECMAScript 2020** (ES11)
- **ECMAScript 2021** (ES12)
- **ECMAScript 2022** (ES13)
- **ECMAScript 2023** (ES14)

Here's a more detailed table of ECMAScript versions, including their release years and significant features:

Version	Release Year	Notable Features

ES1	**1997**	**Basic language syntax, operators, functions, objects, and control structures.**
ES2	**1998**	**Minor updates and bug fixes.**
ES3	**1999**	**Regular expressions, try/catch for error handling, new Array methods (push, pop), and String methods (trim).**
ES4	**2000**	**(un-released)**
ES5	**2009**	**Strict mode, JSON support, new Array methods (forEach, map, filter), Object.defineProperty, getters/setters.**
ES6/ECMAScript 2015 **(Current Standard Version)**	**2015**	**Arrow functions, classes, modules, promises, let and const, template literals, destructuring, Map and Set.**
ES7/ECMAScript 2016	**2016**	**Exponentiation operator (**), Array.prototype.includes() .**
ES8/ECMAScript 2017	**2017**	**async/await, Object.entries(), Object.values(), string padding methods (padStart, padEnd).**
ES9/ECMAScript 2018	**2018**	**Asynchronous iteration (for await...of), rest/spread properties for objects.**
ES10/ECMAScript 2019	**2019**	**Array.prototype.flat(), Array.prototype.flatMap(), Object.fromEntries().**

ES11/ECMAScript 2020	**2020**	**Optional chaining (?.), nullish coalescing operator (??), BigInt.**
ES12/ECMAScript 2021	**2021**	**Logical assignment operators (&&=, `**
ES13/ECMAScript 2022	**2022**	**Top-level await, class fields (public and private), and private methods.**
ES14/ECMAScript 2023	**2023**	**Array.prototype.findLast(), Array.prototype.findLastIndex(), hashbang (#!) grammar support.**

.

JavaScript console.log()

In JavaScript, console.log() is a crucial method used to print a message on the web console. It is an efficient and useful method often used for testing and debugging. In this chapter of the JavaScript tutorial, we will explore **console.log() in JavaScript** through various examples.

What is console.log() in JavaScript?

The **JavaScript console.log()** method displays messages or data in the browser console. By printing the output in the console, we use it to debug the code.

Suppose a developer requests an API and wants to analyze the data received in response. So, they might use the console.log() method for this purpose. There are different use cases of this method that we will discuss later. First, let's take a look at its syntax.

Syntax

console.log("");

The message within "..." can be a value or a variable whose value must be printed to the console.

Parameters

In the syntax, the message can contain a number, string, or array object.

Return value

The method returns the value of the parameter.

Console.log() with Client-side JavaScript

When we use the **JS console.log()** method in the frontend or client-side code, it prints the message in the browser's console. To open the console in the Chrome browser, use the following shortcut:

Windows/Linux

Press Ctrl + Shift + J

macOS

Cmd + Option + J

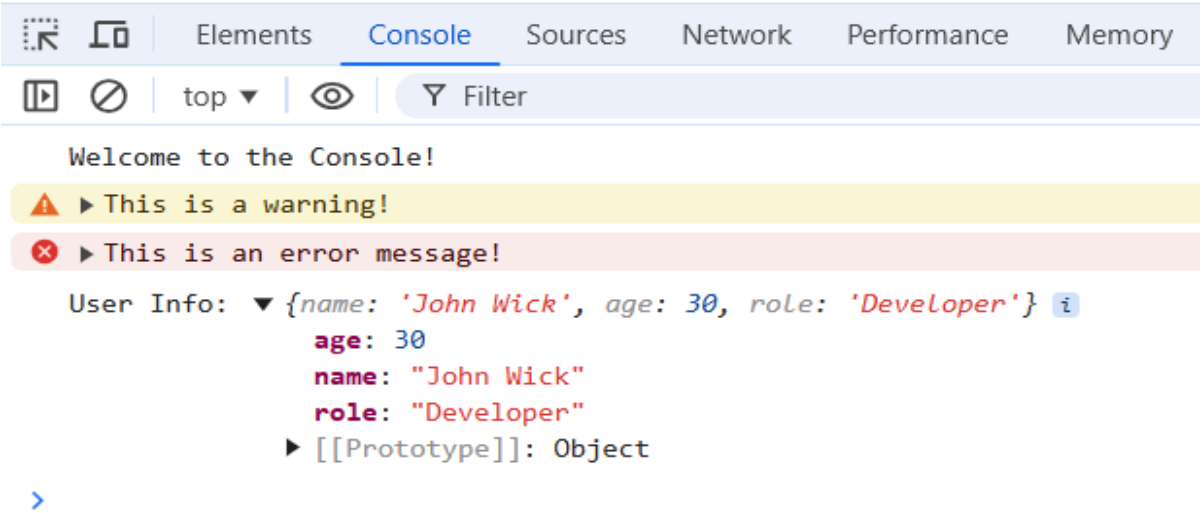
You can also right-click on the mouse to open the console and follow Inspect→Console options.

Example :

```
console.log("Welcome to the Console!");
console.warn("This is a warning!");
console.error("This is an error message!");

const user = { name: "John Wick", age: 30, role: "Developer" };
console.log("User Info:", user);
```

Output



Can we format output with console.log()?

Yes, we can use format specifiers, such as %d, %c, and %s, in console.log().

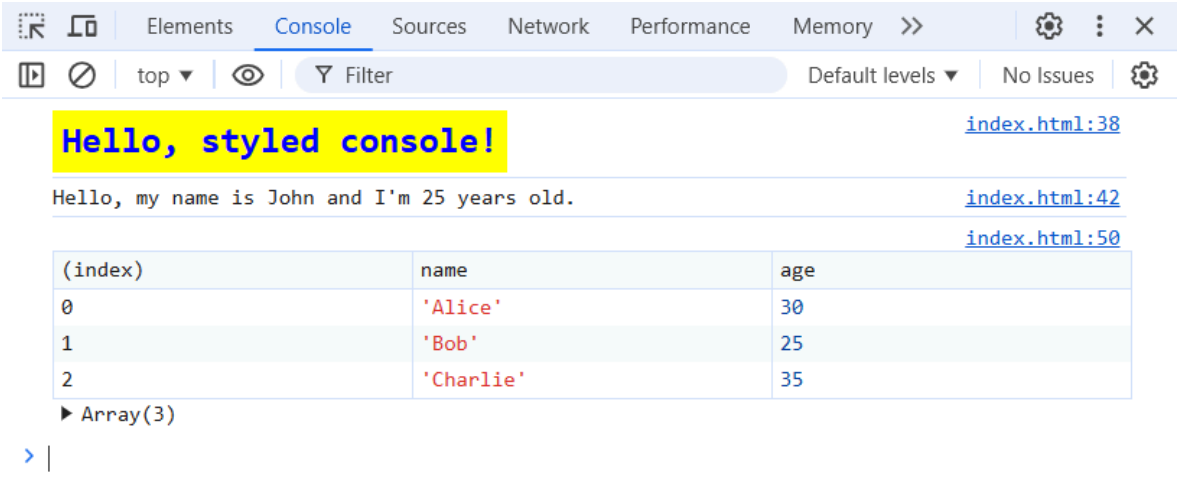
```
     console.log("%cHello, styled console!", "color: blue; font-size:
20px; font-weight: bold; background: yellow; padding: 5px;");

        const name = "John";
        const age = 25;
        console.log("Hello, my name is %s and I'm %d years old.", name,
age);


 const users = [
          { name: "Alice", age: 30 },
          { name: "Bob", age: 25 },
          { name: "Charlie", age: 35 },
          ];

        console.table(users);

Output -
```



How can we log errors with console.log()?

You can log errors using console.log(), but using console.error() for error messages is a better option because it differentiates them visually, and filtering the console is easier

How to use console.log() to debug JavaScript code?

Insert console.log() statements at different places in the code to check variable values, understand the flow of execution, and inspect objects.

How can we print multiple values with console.log()?

To print multiple values with console.log(), we separate them using commas.

Is console.log() asynchronous or synchronous?

In JavaScript, console.log() is synchronous, so it executes instantly in the current thread before the next line of code runs.

JavaScript Variables and Constants

A **variable in JavaScript** is used to store data. It is like a container for storing reusable values.

The assignment operator (=) assigns the value of the right-side operand to the left-side operand.

For example

```
Let num = 7;
```

Here, num is a variable storing the value 7.

JavaScript Constants are variables whose values can't be changed throughout the execution of the program. We can declare constants using the const keyword.

For example

```
// assign 5 to num

const num = 5;
```

Once the constant variable is initialized, we can't change its value.

This tutorial will focus on JavaScript variables and constants, exploring methods of declaring them, and more through examples.

What is a JavaScript Variable?

JavaScript variables are storage locations, storing data that we can change later. They act like containers, where we can place our data and refer to it by naming the container. There are two types of variables in JavaScript- Local variables and Global variables. We will discuss the two later in the chapter.

Declaring a Variable in JavaScript

Before using a variable in a program, we must declare it. As JS is a dynamically typed language, the variable type is decided at the runtime. Hence, we don't need to explicitly define the variable type.

Rules for Declaring a Variable in JS

- JS variables are case-sensitive. It means WORLD and world are two different variables.
- A variable name can't be a reserved keyword (special words reserved for specific purposes).
- Variable names can begin only with a letter (a to z or A to Z), dollar sign ($), or underscore (_).
- After the first letter, a variable name can also contain digits (0 to 9), but you can't use a number at the beginning.

JavaScript variable declaration can be done in three ways:

- Using the var keyword
- Using the let keyword
- Using the const keyword

Syntax

```
// Declaration using var
var Wscube = "Welcome"


// Declaration using let
let $ = "Welcome"


// Declaration using const
const _example = "Welcome"
```

Using the three keywords, we can perform the basic task of declaring a variable in JavaScript. The const keyword assigns a permanent value to a variable; if we try to change the variable's value with the const keyword, it throws an error. The var and let keywords are mutable, so we can change their values. Moreover, let and const keywords were introduced in JavaScript in ES6 and are block-scoped. Before ES6, only the var variable was used to declare variables, and it was scoped.

Let’s compare let, var, and const keywords in JavaScript based on different properties:

Property	**var**	**let**	**const**
Scope	Function scoped	Block scoped	Block scoped
Support	The old version of the browser supports it.	The old version of the browser doesn’t support it.	The old version of the browser doesn’t support it.
Hoisting	At top	At top	At top
Redeclaration	Can be redeclared	Can’t be redeclared	Can’t be redeclared
Updation	Mutable	Mutable	Immutable
Origin	Pre ES2015	ES2015(ES6)	ES2015(ES6)

Var :

Example:

```
var x = 10;
var x = 15;
```

```
x = 20
console.log(x);
```

Output:

20

Explanation:

- **var allows redeclaration: Variables declared with var can be redeclared within the same scope without errors. This is why the second declaration (var x = 20;) overwrites the first one.**
- **Value can be reassigned: Variables declared with var can have their values changed after initialization.**

Let :

let does not allow redeclaration:

```
let y = 10;
let y = 20;
```

Output:
SyntaxError: Identifier 'y' has already been

Variables declared with let can have their values changed after initialization.

Initializing a Variable in JavaScript

Initialization refers to storing a value in a **JS variable**, and we can do this while creating a variable or later when we need it. As JavaScript is an untyped language, its variable can store a value of any data type. Therefore, unlike many other programming languages, we don't need to tell JavaScript during variable declaration what type of value it must hold. We can change the type of value of a variable during program execution, and JS will handle it automatically.

We can assign a value to a variable using the assignment operator (=).

Example

```
// Declare variable num

let num;


// Assign 9 to num

num = 9;
```

Here, we have assigned 9 to the variable num. We can also initialize variables while declaring them.

```
// Declare variable num1 and assign 9 to it

let num1 = 9;


// Declare variable num2 and assign 6 to it

let num2 = 6;
```

Changing the Value of a Variable

As the value of a JavaScript variable can vary, it is called a variable.

Example

```
let data = 10;
data = 20;
console.log(data);
```

Output:

```
20
```

JavaScript Data Types

JS variables can hold values of dynamic data type. This means we can store a value of a boolean, string, number, or object data type.

Example

```
let data = 10; // Initially a Number
console.log(data);

data = "Hello, world!"; // Changed to a String
console.log(data);

data = true; // Changed to a Boolean
console.log(data);
```

Output:

```
10
Hello, world!
true
```

JavaScript Variable Names (Identifiers)

Variables in JavaScript must be named using unique characters. These names are known as identifiers. The following are the rules for naming identifiers in JavaScript:

- They are case-sensitive.
- They can contain the Unicode, so developers can define variables using any language.
- Identifiers are different from keywords.

Example - 1

```
let userName = "John";
let username = "Doe";
console.log(userName);
console.log(username);
```

Output:

John

Doe

Example - 2

```
let имя = "Иван"; // Russian
let 名字 = "张伟"; // Chinese
console.log(имя);
console.log(名字);
```

Output:
Иван
张伟

```
Example - 3
Using JavaScript reserved keywords as variable names is not allowed.
let var = 10;

Output:
SyntaxError: Unexpected token 'var'
```

Example - 4

```
let _name = "Alice";  // Valid, starts with an underscore
let $price = 100;     // Valid, starts with a dollar sign
let age1 = 25;        // Valid, contains letters and digits
let 1name = "Bob"; // Invalid, starts with a number
let first-name = "John"; // Invalid, contains a hyphen
```

Undefined Variable Value in JavaScript

When we don't initialize a variable once it is declared, it has an undefined value. We can also assign an undefined value to variables.

Example

```
// Declared but not initialized
let uninitializedVar;
console.log(uninitializedVar);
// Explicitly assigned undefined
let explicitUndefined = undefined;
console.log(explicitUndefined);
```

Output:

```
Undefined
undefined
```

JavaScript Variable Scope

The scope of a **variable in JavaScript** refers to the area of the program in which it is defined. Variables have two scopes:

1. Global Variables

A global variable can be defined anywhere in the code and is accessible from any function. It can be declared outside the function or with the window object.

Example

```
var globalVar = "I am a global variable";
function displayGlobalVar() {
console.log(globalVar); // Accessible here
 }
displayGlobalVar();
```

Output:

```
I am a global variable
```

2. Local Variables

A local variable can be declared within a block or function. It is accessible only within the function it is defined in. Function parameters are local to that function. In a function body, a local variable takes precedence over a global variable that has the same name. If the declared variable or function parameter has the same name as the global variable, we can hide the lat effectively.

Example

```
function showMessage() {
  let message = "Hello, I am a local variable!";
  console.log(message);
}

showMessage();
console.log(message);
```

Output:

```
Hello, I am a local variable!
Error: message is not defined
```

What is a JavaScript Constant?

The const keyword in JavaScript is used to define variables whose values can't be changed once assigned. It restricts any modifications to the value of a variable. It was introduced in ES6 to create immutable variables. We can say that a variable defined using the const keyword has constant reference.

Additionally, a variable defined with **const in JavaScript** can't be redeclared or reassigned within the same scope, and it provides block scope.

Syntax

```
const const_name;

const x;
```

Features of Values Defined Using Const in JavaScript

- It is a primitive value.
- It can't be reassigned.
- It can be assigned to a variable on the declaration line.
- Has a block scope.
- It can't be hoisted.
- Values within the const array can be changed. It is possible to add new items to const arrays, but referencing a new array is not allowed.
- The value has a read-only reference.
- We can redeclare a const variable within different block scopes.

Declaring Constants in JavaScript

If a variable is declared using the const keyword, we must assign a value to it at the time of declaration. We can't declare a variable with the const keyword without initialization.

Example

```
const PI = 3.14; // Declared and initialized
console.log(PI);
const radius;
```

Output

```
3.14
SyntaxError: Missing initializer in const declaration
```

Examples of Const in JavaScript

Cannot be Reassigned

Example -

`const` does not allow reassignment or redeclaration:

```
const z = 10;
z = 20;
```

Output:

```
TypeError: Assignment to constant variable
```

Block Scope

A variable in JS declared using the const keyword has the Block scope, so one variable is treated differently inside and outside the block. We can't redeclare the const keyword within the same block.

Example

```
{
    const blockScoped = "Inside Block";
    console.log(blockScoped);
}
console.log(blockScoped);
```

Output

```
Inside Block
Error: blockScoped is not defined
```

Cannot be Hoisted

Variables defined with the const keyword can't be hoisted at the top of the code.

Example

```
console.log(a);
const a = 5;
console.log(a);
```

Output

```
Error: Cannot access 'a' before initialization
5
```

What is a Data Type in JavaScript?

JavaScript data types are the types of values that we store or work with. The data type depends on the attributes associated with the value. It relates the declared variable with a specific type so the compiler can perform operations efficiently.

In any programming language, one of the most crucial features is the set of data types it supports, as they are used to represent and manipulate values. In JS, data types are broadly divided into two categories:

1. Primitive data types

2. Non-primitive data types

Primitive Data Type in JavaScript

These are the predefined data types in JavaScript, which are also known as built-in data types. They are further classified into seven data types:

1. Number
2. String
3. Null
4. Boolean
5. BigInt
6. Undefined
7. Symbol

Data Type	**Description**
Number	A numeric value, which can be integer, floating-point, or exponential. They are stored in double-precision 64-bit binary format IEEE 754.
String	Comprises a list of characters, essentially an array of characters.
Null	Has only one value, which is null.
Boolean	Used to represent a logical entity and has two values: true or false.

BigInt	A built-in object used to represent whole numbers that are larger than 253-1.
Undefined	A variable without an assigned value is known as undefined.
Symbol	Returns unique identifiers used as an identifier.

Non-Primitive Data Type in JavaScript

These data types are derived from primitive data types and are also known as reference or derived data types. There are two types of non-primitive data types:

1. Objects
2. Arrays

Data Type	**Description**
Object	A collection of key-value pairs or properties, where each key is a string and each value can be any data type.
Array	A collection of elements, where each element can be of any data type, which also includes other arrays and objects.

Data Types in JavaScript- Explanation and Examples

Let's dive deeper into **JS data types** with detailed explanations of each data type and their examples.

Primitive Data Types

In JavaScript, primitive data types are defined on the basic level. They are predefined and can't be destructured further. They don't have any predefined properties and are used only to store items associated with their respective type.

Number

We are all familiar with numbers and deal with them almost daily in our lives. Whether you share your phone number, calculate the price of a product, or perform mathematical calculations, many tasks involve different numbers, like fractions, decimals, integers, etc.

The number data type in JavaScript represents integer and float-point numbers, and JS doesn't differentiate between them. Hence, it has the same data type for long integers, decimals, and floating-point numbers. Some of the special numbers used in JavaScript are 'Infinity,' '-Infinity,' and 'NaN' (which denotes computational error).

Declaration

The number data type can be declared using the let, var, or const keyword followed by the variable name and the assigned number.

Syntax

```
var x = 1;
```

Example

```
let num = 7;       // Integer
```

```
let num2 = 56.8;     // Floating point number
let num3 = Infinity;      // Infinity
let num4 = ‘mention the number’/8;     // NaN
```

String

We may not realize it, but we often see elements that represent a string. It can be a name, text in a note, or paragraph in a novel. So, a string data type is a series of characters or texts surrounded by quotes. They store elements in a textual form. Strings are different from numbers and have certain constraints and operations. Strings in JavaScript use three types of quotes:

Let str1 = ‘Hello’ (Single quotes)

let str2 = "Hello"; (Double quotes)

let str3 = `Hello`; (Backticks)

Single and double quotes are similar, but backsticks offer additional functionality by allowing us to embed variables inside them.

Declaration

The string **data types of JavaScript** can be declared by assigning texts to the declared literals. We use the following ways for a string declaration:

Single Quotes- Enclosing the text within single quotes (‘...’).

Syntax

```
var str = 'Hello';
```

Double Quotes- Enclosing texts between double quotes (“...”).

Syntax

```
var str = "Hello";
```

Backticks Declaration- Enclosing the text between backticks (`...`).

Syntax

```
var str = `Hello`;
```

Key points to remember

- Strings follow 0-based indexing, so the first element of a string is considered at the 0th position, the second one at the 1st position, and so on.
- To concatenate two strings, we use the + operator.
- Once a string is created, we can't manipulate it.

Example - 1

```
const str = "Hello, World!";
console.log(str[0]);
console.log(str[7]);
```

Output:

H

W

Example - 2

```
const str1 = "Hello";
const str2 = "World";
const result = str1 + ", " + str2 + "!";
console.log(result);

// Using concat() method
const result2 = str1.concat(", ", str2, "!");
console.log(result2);
```

Output

```
Hello, World!
Hello, World!
```

Example - 3

```
let str = "Hello";
str[0] = "J"; // Attempt to change the first character
console.log(str);

// To "modify," create a new string
str = "J" + str.slice(1);
console.log(str);
```

Output

```
Hello
Jello
```

Null

There can be elements that hold no value and don’t belong to any default data type. These elements are represented by null in JavaScript. It is a separate data type that contains only null values.

The null data type is an empty object representing any object's intentional absence.

Declaration

To declare a null value, we use the var, let, or const keyword and assign it the value null.

Syntax

```
let x = null;
```

Key points to remember

- It is falsy, which means in a conditional situation, it will return False.

Example

```
let y = null;
if (y) {
console.log("This will not execute");
} else {
 console.log("Null is falsy");
}
```

Output

```
Null is falsy
```

- Typeof null is an object.

Example

```
let z = null;
console.log(typeof z);
```

Output

```
object
```

Boolean

When a situation has only two possible outcomes- True or False, we use Boolean data type. This data type in JavaScript is used to check for true or false conditions. Also known as logical data type, boolean is generally used in conditional statements or places that require authentication.

Syntax

```
let isCoding = True;
let isOld = False;
```

Example

```
let isAuthenticated = true;

if (isAuthenticated) {
    console.log("Access Granted!");
} else {
    console.log("Access Denied!");
}
```

Output

```
Access Granted!
```

```
//Type Conversion to Boolean

console.log(Boolean(1));
console.log(Boolean(0));
console.log(Boolean("Hello"));
console.log(Boolean(""));
console.log(Boolean(null));
```

Output

```
True
false
True
False
false
```

BigInt

What will you do if you want to represent a really large number? Yes, you can use paper for certain data, but what if certain data types have memory constraints? In such cases, we use a built-in object called BigInt that provides a way to represent whole numbers greater than 253.

JavaScript uses the **64-bit IEEE 754 floating-point format** to store numbers, which can accurately represent integers from **-9007199254740991** to **9007199254740991**. Numbers outside this range lose precision. For larger integers, use **BigInt**.

The **BigInt data type in JavaScript** is used for integers beyond the range of the regular number data type. We can create a BigInt number by appending n to the end of an integer. Unlike the number data type, BigInt doesn't represent decimal values and is used only for whole numbers.

Declaration

We can declare BigInt values using two methods:

- Appending n at the end of the number.

Syntax

```
var num = 954298000127632n;
```

- Passing the number as an argument to the BigInt() constructor.

Syntax

```
var num = BigInt(954298000127632);
```

Example

```
// Maximum safe integer in JavaScript
const maxSafeInteger = Number.MAX_SAFE_INTEGER; // 9007199254740991
console.log("Max Safe Integer (Number):", maxSafeInteger);

// Adding 1 to the maximum safe integer with Number
const unsafeAddition = maxSafeInteger + 1;
console.log("Unsafe Addition (Number):", unsafeAddition);

// Using BigInt to safely handle and increase the maximum safe integer
const bigIntValue = BigInt(maxSafeInteger);
const incrementedBigInt = bigIntValue + 1n;

console.log("BigInt Representation:", bigIntValue);
console.log("Incremented BigInt:", incrementedBigInt);
```

Output

Max Safe Integer (Number): 9007199254740991

Unsafe Addition (Number): 9007199254740992

BigInt Representation: 9007199254740991n

Incremented BigInt: 9007199254740992n

Undefined

An **undefined data type in JavaScript** represents the absence of a value. Any variable that has been declared but hasn't been initialized with a value is automatically assigned the undefined value, indicating that it exists but has no value assigned to it.

Declaration

We use the var, let, or const keyword to declare an undefined data type in JavaScript. There is no need to assign anything to the literal.

Syntax

```
let name = undefinedValue;
```

Example

```
// Declaration of variables without assigning a value
let myVariable; // myVariable is declared but not initialized
console.log(myVariable);
```

Output:

```
undefined
```

```
// Assigning the undefined value explicitly
let anotherVariable = undefined;
console.log(anotherVariable);
```

Output:

```
Undefined
```

```
// Function example where no return statement is provided
function myFunction() {
// No return statement
}
 const result = myFunction();
console.log(result);
```

Output:

```
Undefined
```

Symbol

A symbol is a new primitive data type in JS introduced in ES6. It represents a unique and immutable value used as an identifier for object properties. When we create a Symbol, JavaScript ensures it is different from other symbols, even if the description is the same. They don't have a literal form and are used to create unique property keys in objects to avoid conflicts with other properties.

Declaration

We can declare a symbol by calling the Symbol() function. Also, it is not a constructor, so if called with the new keyword, it throws an error. Every time a symbol() is called, it creates a new symbol, and it has nothing to do with the key passed.

Syntax

```
let s1 = Symbol("Hello");

let s2 = Symbol("Hello");
```

Example - 1

```
// Symbols are unique, even if descriptions are the same
let symbol1 = Symbol("description");
let symbol2 = Symbol("description");
console.log(symbol1 === symbol2);
```

Output
False

Example - 2

```
// Prevents accidental property name conflicts
const id = Symbol("id");
const user = {
 name: "Alice",
 [id]: 12345 // Using a symbol as a key
};
 console.log(user[id]);
console.log(user.id);
```

Output
12345
undefined (no conflict with string 'id')

```
// Symbols are not accessible via standard iteration
console.log(Object.keys(user));
console.log(Object.getOwnPropertySymbols(user));
```

Output

["name"]

[Symbol(id)]

Example - 3

```
// Hiding internal details in a class using Symbols
const secretKey = Symbol("secret");

class SecretHolder {
  constructor(secret) {
    this[secretKey] = secret; // Store secret using a symbol
  }

  getSecret() {
    return this[secretKey];
  }
}

const holder = new SecretHolder("Top Secret");
console.log(holder.getSecret());
console.log(holder.secretKey);
```

Output

```
Top Secret
undefined
```

Non-Primitive Data Types

Non-primitive data types are not defined at the basic level but represent complex data types that are formed upon operations with primitive data types. They are also known as reference data types in JavaScript and can perform multiple functions.

Object

Objects in JavaScript are fundamental data structures that store data collections. They hold data in key-value pairs. We can create objects using curly braces {} or the new keyword. As everything in JavaScript is an object, it is crucial to understand this concept.

Declaration

We can create objects using two methods:

Object Literal Method- We use a let, var, or const keyword to create a literal and assign a set of key-value pairs wrapped inside curly braces {}.

Syntax

```
let person = {};
```

Object Constructor Method- We use a let, var, or const keyword to create a literal and assign a set of key-value pairs wrapped inside curly braces {}.

```
let person = new Object();
```

Example-1

```
// Creating an object with key-value pairs
let person = {
  name: "John",
  age: 30,
  profession: "Engineer",
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  }
};

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

Output:

```
John
30
```

```
// Calling a method
person.greet();
```

Output:

```
Hello, my name is John
```

```
// Adding a new property
person.hobby = "Reading";
console.log(person.hobby);
```

Output:

```
Reading
```

Example-2

```
// Creating an object using the constructor method
let car = new Object();
car.brand = "Toyota";
car.model = "Corolla";
car.year = 2020;

// Accessing properties
console.log(car.brand);
console.log(car["model"]);
```

Output:

```
Toyota
Corolla
```

```
// Adding a method
car.getCarDetails = function () {
  return `${this.brand} ${this.model}, ${this.year}`;
};

console.log(car.getCarDetails());
```

Output:

```
Toyota Corolla, 2020
```

Example-2

```
let student = {
  name: "Alice",
  grades: {
    math: 85,
    science: 90,
    history: 78
  },
  displayGrades: function () {
    for (let subject in this.grades) {
      console.log(`${subject}: ${this.grades[subject]}`);
    }
  }
};

// Accessing nested object properties
console.log(student.grades.math);

Output:
85
```

Explanation: Objects can hold other objects as values

Array

An array in JavaScript is a special object that stores an ordered collection of elements or values, which can be of any data type. They are like containers to store different or similar items. We use square brackets [] to create an array and separate values using commas.

Declaration

We can declare an array using the following methods:

Array Literal- We declare arrays using the let, const, or var keyword and assign values using square brackets [].

Syntax

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

Example

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

Output:

['Apple', 'Banana', 'Cherry']

Constructor Method- We can also call the array() constructor and use the new keyword.

Syntax

```
let ar = new Array();
```

Example

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

console.log(numbers);

```
let numbers = new Array(1, 2, 3, 4, 5);
console.log(numbers);  :
```

[1, 2, 3, 4, 5]

JavaScript Operators

Most programming languages use operators to perform different operations or tasks on one or more operands, which can be values or variables. For example, using the + operator, we can add operands/values 7 and 2. So, 7 + 2 will return the output 9

Similarly, there are numerous **operators in JavaScript** represented by specific symbols. They are used to perform various operations. Let's delve deeper and learn about different JavaScript operators.

What are Operators in JavaScript?

Operators in JS are special symbols used to perform specific logical or mathematical computations on one or more operands. These fundamental elements in JS allow programmers to manipulate data and control the flow of the program. Anyone who aspires to be a proficient JavaScript programmer must gain a fair understanding of different types of operators and how they work to be able to write optimized and effective code.

Types of Operators in JavaScript

JavaScript supports various types of operators that we have listed below:

1. Assignment Operators

2. Arithmetic Operators
3. Logical Operators
4. Bitwise Operators
5. Comparison Operators
6. Ternary Operators
7. String Operators
8. Unary Operators
9. BigInt Operators
10. Comma Operators
11. Miscellaneous Operators

Let's have a look at all these **types of operators in JS** in detail, exploring their uses, syntax, and examples.

Assignment Operators

In JavaScript,assignment operators are used to assign values to a variable. It evaluates the assigned value. We can chain the assignment operator to assign a single value to multiple variables.

Operator	**Description**	**Example**
Assignment (=)	Assigns the value of the right operand to the left operand.	A = X, B = A, then B = X
Addition Assignment (+=)	adds the right and left operands and assigns the result to the left operand.	A += X gives A = A + X
Subtraction Assignment (−=)	Subtracts the right operand from the left one and assigns the result to the left operand.	A -= X gives A = A - X
Multiplication Assignment (*=)	Multiplies the right and left operands and assigns the result to the left operand.	A *= X gives A = A * X

Division Assignment (/=)	Divides the left operand by the right one and assigns the result to the left operand.	A /= X gives A = A / X
Modules Assignment (%=)	Divides the left operand by the right one and assigns the remainder to the left operand.	A %= X gives A = A % X

Example

A.)

```
let x = 2 * 2 + 1;

alert( x );
```

Output:

5

Explanation:

we assign a variable, like `x = 2 * 2 + 1`, the calculations are done first and then the = is evaluated, storing the result in `x`

B.) Returns the assigned value.

.When you write `x = value`, it stores the value in x and returns that same value
.This allows you to use the assignment operation as part of a larger expression.

Example

```
let a = 1;
let b = 2;

// Assign (b + 1) to a, then subtract it from 3 to calculate c
let c = 3 - (a = b + 1);

console.log(a);
console.log(c);
```

Output:
3, because a was set to b + 1 (2 + 1 = 3)

0, because 3 - 3 = 0

Explanation:

1. **The expression `(a = b + 1)`:**
    - **Calculates b + 1 (which is 2 + 1 = 3).**
    - **Assigns 3 to a.**
    - **Returns 3.**
2. **Then 3 - 3 is calculated, resulting in `0`.**

This is why a becomes 3 and c becomes `0`. The assignment = behaves like any other operator by returning a value.

C.) Chaining assignment

```
let x, y, z;
x = y = z = 5;

console.log(x);
console.log(y);
console.log(z);
```

Output:

5

5

5

Arithmetic Operators

Arithmetic operators in JavaScript perform mathematical or arithmetic calculations on the operands. These calculations include addition, subtraction, division, and multiplication. Below, we have explained each arithmetic **operator in JavaScript with an example.**

Operator	**Description**	**Example**

Addition (+)	Adds two operands.	20 + 10 = 30
Subtraction (-)	Subtracts the second operand from the first one.	20 - 10 = 10.
* (Multiplication)	Multiplies two operands.	20 * 10 = 200.
/ (Division)	Divides the numerator by the denominator.	20 / 10 = 2
% (Modulus)	The result is the remainder of an integer division.	20 % 10 = 0
++ (Increment)	Increases an integer value by one.	10++ = 11.
-- (Decrement)	Decreases an integer value by one.	10-- = 9.

Example

The first four are straightforward, while `%` (Modulus) , ++ (Increment) and -- (Decrement) need a few words about them -

A.) % (Modulus)

Modulus alos called remainder operator %, despite its appearance, is not related to percents.
The result of `a % b` is the remainder of the integer division of `a` by `b`

```
console.log(5 % 2);
console.log(8 % 3);
console.log(10 % 5);
```

Output:

```
1

2

0
```

B.) `++` (Increment) increases a variable by 1:

```
let x = 10;
x++;
console.log(x);
```

Output:

```
11
```

C.) `--` (Decrement) decreases a variable by 1:

```
let x = 10;
x-;
console.log(x);
```

Output:

```
9
```

Important:
The operators `++` and `--` can be placed either before or after a variable.

When the operator goes after the variable, it is in "postfix form": `counter++`.

The "prefix form" is when the operator goes before the variable: `++counter`.

```
let counter = 1;
let a = ++counter;
let b = counter++;
let c = -counter;
let d = counter-;

console.log(a);
console.log(b);
console.log(c);
console.log(d);
```

```
Output:
2
1
0
1
```

Logical Operators

JavaScript logical operators perform logical operations, such as And, OR, and NOT. They evaluate expressions and return boolean values.

Operator	Description	Example
Logical AND (&&)	Returns true if both expressions are true.	x && y
Logical OR (||)	Returns true if at least one operand is true.	x || y
Logical NOT (!)	Reverses the boolean value of the operand.	!x

Example

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

// Logical AND (&&)
console.log(x && y);

// Logical OR (||)
console.log(x || y);

// Logical NOT (!)
console.log(!x);
```

Output:

false

true

false

Bitwise Operators

In JavaScript, bitwise operators perform bit-level operations on integers. These **types of operators in JavaScript** convert the number to a 32-bit binary number, performing bitwise operations on them. Once the result is computed, it converts the number back to the 64-bit number.

Operator	**Description**	**Example**
Bitwise AND (&)	Performs a Boolean AND operation on each bit of the integer argument.	6 & 1 = 0
Bitwise OR (|)	Performs a Boolean OR operation on each bit of the integer argument.	5 | 3 = 7
Bitwise XOR (^)	Exclusive OR means either the first operand is true or the second but not both. This operator performs the Boolean exclusive OR operation on each bit of the integer argument.	6 ^ 1 = 7
Bitwise Not (~)	Reverses all the bits of the operand.	~5 = -6
Bitwise Left Shift (<<)	Shifts all the bits in the first operand to the left by the number of places given in the second operand.	(x << 1) is 4.
Bitwise Right Shift (>>)	Shifts the left operand's value to the right by the number of bits given by the right operand.	(x >> 1) is 1.
Right shift with Zero (>>>)	Similar to the bitwise right shift >> operator, but the bits shifted in on the left are zero.	(x >>> 1) is 1.

Example

```
let a = 6; // 110 in binary
let b = 3; // 011 in binary


// Bitwise AND (&)
console.log(a & b);
```

```
// Bitwise OR (|)
console.log(a | b);

// Bitwise XOR (^)
console.log(a ^ b);

// Bitwise NOT (~)
console.log(~a);

// Bitwise Left Shift (<<)
console.log(a << 1);

// Bitwise Right Shift (>>)
console.log(a >> 1);

// Right shift with Zero (>>>)
console.log(a >>> 1);
```

Output:

2 (010)

7 (111)

5 (101)

-7

12 (1100)

3 (011)

Comparison Operators

Comparison operators in JavaScript are used to compare two values. They perform logical operations to find the equality or difference between the values.

Operator	**Description**	**Example**
Equality (==)	Checks if the two operands are equal. If yes, it returns true; else, false.	(a == b) is false

Inequality (!=)	Checks the inequality of two operands. If the values of operands are not equal, it returns true; else, false.	(a != b) is true
Strict equality (===)	Returns true if the value and data type of two operands are equal; else, returns false.	(a === b) is false
Strict inequality (!==)	Checks the inequality of values and data type of the operands. It returns true if they are not equal; else, false.	(a !== b) is true
Greater than (>)	Checks if the left operand value is greater than the right operand value. If yes, it returns true.	(a > b) is false
Greater than or Equal to (>=)	Returns true if the value of the left operand is greater than or equal to the value of the right operand; else, returns false.	(a>=b) is false
Less than (<)	Checks if the left operand value is less than the right operand value. If yes, it returns true.	(a < b) is true.
Less than or Equal to (<=)	Returns true if the value of the left operand is less than or equal to the value of the right operand; else, returns false.	(x <= y) is true.

Example

```
let a = 5;
let b = 10;

// Equality (==)
console.log(a == b);

// Inequality (!=)
console.log(a != b);

// Strict equality (===)
console.log(a === b);
```

```
// Strict inequality (!==)
console.log(a !== b);
// Greater than (>)
console.log(a > b);

// Greater than or Equal to (>=)
console.log(a >= b);

// Less than (<)
console.log(a < b);
// Less than or Equal to (<=)
console.log(a <= b);
```

Output:

false

true

false

true

false

false

true

true

Ternary Operators

In JavaScript, a ternary operator evaluates the given condition and executes a block of code according to that condition. It is a simplified operator of if/else. Hence, it is also known as the conditional operator. The ternary operator has three operands.

Name	Description	Example

Ternary Operator (?:)	It is the shorthand of the if-else condition.	X = ? exp1 : exp2 If the condition is true then X = exp1; else, X = exp2.

Example

```
let age = 18;
let canVote = (age >= 18) ? "Yes, you can vote!" : "No, you cannot vote yet.";
console.log(canVote);
```

Output:

Yes, you can vote!

String Operators

JavaScript string operators use concatenation (+) to join two strings.

Name	**Description**	**Example**
concatenation (+)	It joins two string values and returns another string that concatenates the two operand strings.	str1 + str2

Example

```
let firstName = "John";
let lastName = "Wick";
let fullName = firstName + " " + lastName;
console.log(fullName);
```

Output:

John Wick

Unary Operators

A unary operator in JavaScript performs an operation with only one operand.

Name	**Description**	**Example**
typeof	Returns the operand type, which can include string, function, undefined, symbol, boolean, number, and function.	typeof variable
Delete	Deletes JavaScript object properties.	delete object // or delete object.property // or delete object[‘property’]

Example

A.)
```
let num = 42;
console.log(typeof num);
```

Output:

number

B.)
```
let person = { name: "Alice", age: 25 };
delete person.age;
console.log(person);
```

Output:

{ name: "Alice" }

BigInt Operators

BigInt operators in JavaScript support arithmetic operations on BigInt data type, which include addition, subtraction, multiplication, and division. Most operators that we can use between numbers can also be used between BigInt values.

Name	Description	Example
BigInt	It returns a new BigInt object representing the original value.	BigInt(value);

Example
```
let bigNumber = BigInt(1234567890123456789012345678 90);
console.log(bigNumber + BigInt(10));
```

Output:

1234567890123456789012345679 00n

Comma Operators

Comma operators in JavaScript evaluate operands from left to right, returning the value of the last or rightmost operand.

Name	**Description**	**Example**
comma operator (,)	Placing a comma operator in an expression executes every expression, returning the rightmost or last expression.	Expression1, Expression2, Expression3, …so on

Example

```
let result = (1 + 2, 3 + 4, 5 + 6);

console.log(result);
```

Output

11

Explanation:

(1 + 2) evaluates to 3

(3 + 4) evaluates to 7

(5 + 6) evaluates to 11 (returned as the final value)

Miscellaneous Operators

JavaScript has a few other operators besides the one we listed above. We have shared them in detail below:

Operator	Description
, (Comma)	Evaluates operands from left to right and returns the last operand.
?? (Nullish Coalescing)	Returns the right operand when the left operand is null or undefined; else, returns the left operand.
** (Exponentiation)	Raises the left operand to the power of the right operand.
yield	Pauses and resumes a generator function.
delete	Removes a property from an object.
instanceof	If the given object is a valid object of the specified class, it returns true.
() (Grouping)	Allows to change the operator precedence.
void	Discards the return value of an expression.
… (Spread)	Expands the iterables such as array or string.

Example:

A.)
```
let a = 1, b = 2, c = (a++, b++, a + b);
console.log(c);
```

Output:

5

B.)

```
let value = null;
let defaultValue = value ?? "Default";
console.log(defaultValue);
```

Output:
Default

C.)

```
let base = 2;
let exponent = 3;
console.log(base ** exponent);
```

Output:
8

D.)

```
let today = new Date();
console.log(today instanceof Date);
```

Output:
true

E.)

```
let result = (2 + 3) * 4;
console.log(result);
```

Output:

20

F.)

```
let arr = [1, 2, 3];
let newArr = [...arr, 4, 5];
console.log(newArr);
```

Output:

[1, 2, 3, 4, 5]

JavaScript Comparison and Logical Operators

In the last chapter, you learned about different operators in JavaScript that are used for mathematical and logical operations. Now, let's dig deeper and explore comparison and logical operators in detail through various examples.

Comparison operators in JavaScript are used to compare two values. They return a boolean value, which can be True or False. Logical operators evaluate boolean expressions and return a boolean value. This chapter of the JavaScript tutorial will explore different comparison and logical operators in JS, how they work, their examples, and more so you can have a fair understanding of how to use these operators to make JavaScript code more efficient and responsive.

What are Comparison Operators in JavaScript?

JavaScript comparison operators are powerful tools used to check certain conditions by comparing two variables or values and returning Boolean values as either true or false. Comparison operators play a vital part in logical operations by determining the equality or inequality between two values.

List of JavaScript Comparison Operators

Here are different **logical operators in JS** that we use to compare two variables or values:

Operator	**Description**	**Example**
Equality (==)	Compares the equality of two operands.	a == b

Inequality(!=)	Compares the inequality of two operands.	a != b
Strict Equality (===)	Checks equality between the value and type of operands.	a === b
Strict Inequality (!==)	Compares inequality with data type.	a !== b
Greater than (>)	Checks if the left operand is greater than the right one.	a > b
Greater than or equal (>=)	Checks if the left operand is greater than or equal to the right one.	a >= b
Less than (<)	Checks if the left operand is less than the right one.	a < b
Less than or equal (<=)	Checks if the left operand is less than or equal to the right one.	a <= b

JavaScript Equality (==) Operator

The equality operator in JavaScript compares the two operands, checking whether their values are equal. If they are equal, it returns true; else, it returns false. In the case of different types of operands, it performs type conversion and then compares the two.

Example

```
console.log(5 == '5');
console.log(5 == 6);
```

Output -

```
true
false
```

JavaScript Inequality (!=) Operator

The inequality operator in JavaScript compares the inequality of two operands, checking if they are not equal. If the two operands are not equal, it returns true; else, it returns false. Similar to the equality operator, it also performs type conversion for two different types of operands before comparing them.

Example

```
console.log(5 != '6');
console.log(5 != 5);
```

Output

```
true
false
```

JavaScript Strict Equality (===) Operator

The strict equality operator compares the two operands and their data types, checking whether they are equal. Basically, it checks the equality between two operands without type conversion. If both operands are equal and have the same type, it returns true; else, it returns false. If both operands are of different types, it returns false without checking their values.

Example

console.log(5 === 5);

console.log(5 === '5');

Output

true

false

JavaScript Strict Inequality (!==) Operator

The strict inequality operator in JavaScript compares the inequality of two operands and their type. It checks the inequality between two operands without type conversion. If the operands are not equal and are not the same type, it returns true; otherwise, it returns false.

Example

```
console.log(5 !== '5');

console.log(5 !== 5);
```

Output

true

false

JavaScript Greater Than (>) Operator

The greater than operator in JavaScript checks whether the value of the left operand is greater than the value of the right one. If yes, it returns true; else, it returns false.

Example

```
console.log(10 > 5);

console.log(5 > 10);
```

Output

true

false

JavaScript Greater Than or Equal (>=) Operator

The greater than or equal to the operator in JavaScript returns true if the value of the left operand is greater than or equal to the value of the right one. If not, it returns false.

Example

```
console.log(10 >= 5);

console.log(5 >= 5);

console.log(4 >= 5);
```

Output

true

true

false

JavaScript Less Than (<) Operator

The less than operator in JavaScript checks if the value of the left operand is less than the value of the right one. If yes, it returns true; else, it returns false.

Example

console.log(3 < 7);

console.log(7 < 3);

Output

true

false

JavaScript Less Than or Equal (<=) Operator

The less than or equal to operator in JavaScript returns true if the value of the left operand is less than or equal to the right operand. If not, it returns false.

Example

console.log(5 <= 5);

console.log(3 <= 5);

console.log(6 <= 5);

Output

true

true

false

What is a Logical Operator in JavaScript?

JavaScript logical operators are used to manipulate a boolean or set conditions for termination in loops. They help make logical decisions based on given conditions.

Types of Logical Operators in JavaScript

Operator	Description	Example
Logical AND (&&)	Returns true only when both operands are true.	a && b
Logical OR (||)	Returns true if at least one operand is true.	a || b
Logical NOT (!)	Returns flipped value.	!expression

Logical AND Operator

The logical AND operator in JavaScript accepts multiple arguments, evaluating operands from left to right. It converts each operand into a boolean. If both expressions are true, it returns true; else, it returns false.

Syntax

```
result = m && n; // Can have multiple arguments
```

Example

const a = true; const b = false;

console.log(a && b);

console.log(a && true);

Output

false

false

true

Logical OR Operator

The logical OR operator in JavaScript also evaluates the operands from left to right and converts them into a boolean first. It returns true, if at least one expression is true; else, it returns false. If all values are false, it returns the last value.

Syntax

```
result = a || b;
```

Example

const a = true; const b = false;

console.log(a || b);

console.log(b || false);

Output

true

false

Logical NOT Operator

The logical NOT operator returns true if the given expression is false and returns false if the expression is true. It reverses the boolean result of the operand and converts it into a boolean type.

Syntax

```
result = !value; // Can have a single argument
```

Example

```
const a = true;

console.log(!a);

console.log(!false);
```

Output

```
false

true
```

What are logical operators used for?

JavaScript logical operators combine two or more conditions or expressions, returning a single boolean value. There are three logical operators- AND, OR, NOT, which help create more complex conditions for a program.

JavaScript Comments

Writing **comments in JavaScript** enhances code maintainability, readability, and collaboration. They are like small notes within the codebase to explain its logic and functionality or provide context.

This chapter focuses on JavaScript comments, explaining each type of **JS comment** in detail through examples, best practices for writing efficient comments, and more.

What are Comments in JavaScript?

JavaScript comments are an effective and meaningful way to explain code and deliver a message. It allows us to add extra information about the code, be it a suggestion or warning, to help the end user interpret the code easily. We can also temporarily use comments to describe despicable parts of a code. The JavaScript compiler ignores comments while running the code.

Types of Comments in JavaScript

There are two types of **comments in JS**:

1. Single-line Comment
2. Multi-line Comment

Single-line Comments in JavaScript

A JavaScript single-line comment is represented by two forward slashes (//), which can be used before and after the statement.

Syntax

```
// your comment here
```

When to use single-line comments:

- To drop quick reminders
- To explain what a specific piece of code does
- To turn off code lines for a bit to test things

Example

// This is a single-line comment

let x = 5;

Multi-line Comments in JavaScript

A JavaScript multiline comment is used to add a single or multiple lines to the source code. It is a more convenient way to comment and is represented by a single forward slash with an asterisk.

Syntax

```
/*

This is a multi-line comment

It can span multiple lines

*/
```

When to use multi-line comments:

- To stop a piece of code from running at once
- To provide the context of a code and what it is meant to do
- To add notes about copyright, who wrote the code, and tasks to do
- To explain tricky parts of code

Example

```
/* This is a multi-line comment.

It can span multiple lines.

Use it to explain complex logic or disable multiple lines of code. */

let y = 10;
```

JavaScript Comments to Prevent Execution

We also use // and /*...*/ to change the execution of the code in JavaScript using comments. They can prevent code execution and are suitable for testing the code.

Example

```
/* let z = 15;

console.log(z); */
```

Advantages of JavaScript Comments

Below, we have explained the benefits of commenting in JavaScript:

- Comments add clarity to the code, making it easier for developers to read and understand its functionality and purpose. They elaborate the code so it acts as a guide, especially when they see the code after a long time.
- Well-commented code highlights potential issues, helps identify bugs, and outlines reasons behind specific solutions, making it easier to debug and maintain.
- We often add code for a specific function but may need to disable it after some time. In such cases, JavaScript comments help us avoid unnecessary code and its execution.
- When working with a team, comments help collaborate by enabling each member to comprehend the code written by others so they can work together seamlessly.

Best Practices for Commenting in JavaScript

Comments in JavaScript are critical, and you must adhere to a few rules and guidelines when writing them.

Explain Intent

While adding comments to code, focus on explaining the 'why' more than the 'how.' It helps others understand why you wrote the code in a certain way.

Avoid Over-commenting

Comments are beneficial, but excessive commenting can result in clutter. So, maintain the balance, writing comments where needed and where they add value without stating the obvious.

Comment Complexity

Make sure to explain the logic behind comments for intricate algorithms or unconventional solutions.

Update Comments Regularly

With code evolving, you must write accurate comments that align with the code changes. If comments don't match the code, it can be confusing. So, update the comments if you change the code. Delete comments that are not needed to keep it clean.

Use Descriptive Comments

Explain the purpose of function, complex logic, and variables through descriptive comments so other developers can understand the intent of the code.

Write Effective Comments

Comments help others understand what's happening in the code. Explain the why through comments so people can know why you wrote the code in a specific way.

JavaScript Type Conversion

JavaScript is a loosely typed programming language that implicitly converts a value most of the time. However, we might need to explicitly convert data types in certain situations. Although JS offers various methods to convert one data type to another, there are two most common conversion methods- Implicit and Explicit.

This chapter will discuss type conversion and its methods through various examples. So, let's get started:

What is Type Conversion in JavaScript?

In JavaScript, type conversion refers to an algorithm that manipulates the data type of a function. It includes automatic or explicit conversion from one **data type in JavaScript** to another. JavaScript can sometimes automatically perform these conversions, ensuring that the function gets valid inputs based on the operator.

Type conversion is a crucial concept of JavaScript as it enables effective comparisons and operations.

Types of type Conversion in JavaScript

Type conversion is possible in two ways:

1. Implicit Conversion
2. Explicit Conversion

1. Implicit Type Conversion

When JavaScript automatically performs type conversion, it is known as implicit type conversion. It converts the value of one data type into another. For example, if we use + with string and number operands, it will convert the number to a string, concatenating it with the string,

Implicit Conversion to String

Type conversion to string means automatic conversion of a non-string value to a string type when used in a context where a string is required. Developers don't need to give explicit instructions as it happens behind the scenes.

Example

let num = 10;

let result = "The number is " + num;

console.log(result);

Output:

The number is 10

Implicit Conversion to Number

When a non-numeric value converts to a number automatically in JavaScript when it is used in numeric contexts, it is referred to as an implicit conversion to a number. For example, arithmetic operations.

When we use a string containing digits with arithmetic operators except for +, it automatically converts the operand to numbers to perform the operation.

Example

```
let str = "5";

let sum = str - 2;

console.log(sum);
```

Output:

3

Null Conversion to Number

We may implicitly use null (represents an empty value) in a numeric context, such as arithmetic operations. In such cases, it is automatically converted to 0 to ensure that the operation with null behaves predictably within numeric contexts.

Example

```
let value = null;

let total = value + 5;

console.log(total);
```

Output:

5

Implicit Conversion to Boolean

When we use the !! operator with a variable, it automatically converts its value to a boolean value.

Example

```
let value = "Hello";

let isTruthy = !!value;

console.log(isTruthy);
```

Output: true

Non-numeric String Results to NaN

In JavaScript, converting a non-numeric string to numbers using Number() results in NaN, which represents 'Not-a-Number'. For example, Number("false"), Number("Hi"), and Number("") will result in NaN as they lack numeric value.

Example

```
let nonNumeric = "abc";

let result = Number(nonNumeric);

console.log(result);
```

Output:

NaN

Undefined with Number, Boolean, or Null

When we use undefined in numeric, boolean, or null operation, it will always give the NaN as the output, indicating it is not a number but an undefined value.

Example

```
let value;

console.log(Number(value));

console.log(Boolean(value));

console.log(value + null);
```

Output:

NaN

false

NaN

Implicitly Converting Numbers to Dates

To create data objects from numeric values, we must instantiate new data using the required timestamp.

Example

```
let timestamp = 1609459200000;

let date = new Date(timestamp);

console.log(date);
```

Output:

Fri Jan 01 2021 00:00:00 GMT+0000 (UTC)

Converting Dates to Numbers

Converting data objects to numbers often represented as milliseconds involves using the getTime() method.

Example

```
let date = new Date();

let timestamp = date.getTime();

console.log(timestamp);
```

Output: e.g., 1700982345678

2. Explicit Type Conversion

Sometimes, developers may need to manually convert one data type of a variable to another using built-in or constructor functions. This is called explicit type conversion. It allows the conversion of data types according to your needs.

Explicit Conversion to String

It involves converting one data type to a string using the String() constructor or toString(). We can convert boolean, numbers, or other data types to another.

Example

```
let num = 123;

let bool = true;

console.log(String(num));

console.log(bool.toString());
```

Output:

"123"

"true"

Explicit Conversion to Number

We may explicitly convert a value to a numeric value using the Number() function. We can also use + to convert a string to a number.

Example

```
let str = "42";

let bool = false;

console.log(Number(str));

console.log(+str);

console.log(Number(bool));
```

Output:

42

42 (Unary plus operator)

0

Explicit Conversion to Boolean

It refers to explicitly converting a value to a boolean type using the Boolean() function, ensuring that the value is interpreted as either true or false and facilitating boolean operations.

Example

```
let value1 = 0;

let value2 = "Hello";

console.log(Boolean(value1));

console.log(Boolean(value2));
```

Output:

false

true

JavaScript Type Conversion Table

In the below table, we have given the original values and their resultant value when we convert the original value to string, number, and boolean.

In the table below are some of the most common JavaScript values conversion to Number, String, and Boolean :point_down:

Value	**String conversion**	**Number conversion**	**Boolean conversion**
0	"0"	0	false
1	"1"	1	true
"1"	"1"	1	true
"0"	"0"	0	true
"One"	"One"	NaN	true
""	""	0	false
NaN	“NaN”	NaN	false
true	"true"	1	true
false	"false"	0	false
null	"null"	0	false
undefined	"undefined"	NaN	false
[44]	"44"	44	true
[22, 44]	"[22, 44]"	NaN	true
['fun','enjoy']	'fun,enjoy'	NaN	true

['fun']	'fun'	NaN	true
Infinity	'Infinity'	Infinity	true
-Infinity	'-Infinity'	-Infinity	true

Rules for JS Type Conversion

You must follow the rules for type conversion to avoid any unexpected issues and results in your code. So, here are the key points to remember:

- When we subtract one string from another, JavaScript tries to convert both of them to numbers.
- Operations with binary + will convert numbers to strings when there is a string and number concatenation.
- Changing the type of an object may cause data loss.
- Logical operations convert operands into boolean values.

JavaScript Scope

You might have faced a situation where your code doesn't work as expected. You define a variable in one part of the code, but it may not work properly when used elsewhere. This is because variables always have a scope, and you can't use them outside of that scope.

In this chapter of the JavaScript tutorial, we will explore the variable **scope in JavaScript** and how to use it efficiently when writing code.

What is JavaScript Scope?

JavaScript scope refers to the area where a variable exists and is accessible. It defines the current context of the code, which determines where we can access certain functions and variables.

So, when we define a variable in JS at a specific part of the code, it affects when we access it elsewhere. Hence, when we define a variable within a function, it is accessible only within that function. JavaScript allows us to layer the scope on the system so the child scope can access the parent scope, but not vice versa.

Different Types of Scopes in JavaScript

There are different types of scope in JavaScript, and we will discuss each of them in depth:

1. Global Scope

2. Local Scope
3. Function Scope
4. Block Scope
5. Lexical Scope

1. Global Scope

Variables declared outside the function or block have a global scope. We can access global variables from anywhere in the program. This is the default scope for variables in JavaScript and can be defined using the let, const, or var keyword.

When to Use Global Scope

- We use global scope for variables that must be accessible from anywhere outside the codebase.
- For example, utility libraries or functions with standard functionality across different modules or components.
- Configuration variables used throughout the application.
- Make sure to use it carefully to avoid potential conflicts with other parts of the code.

Example

```
let globalVar = "I am a global variable";


function showGlobalVar() {
   console.log(globalVar);
}


showGlobalVar();


console.log(globalVar);
```

Output:

```
I am a global variable
I am a global variable
```

Local Scope

A variable declared inside a function and available only in the current code block has a local scope. We can access these variables only within the function in which they are defined. If we try to access them from another function, it will throw an error. As we can't access these variables from outside the function, we can have a variable of the same name in another function.

When to Use Local Scope

- For variables that can be accessed only within the function, they are declared in.
- Usually declared within functions, conditional statements, loops, or other code blocks.

Example

```
function localScopeExample() {
   let localVar = "I am a local variable";
   console.log(localVar);
}


localScopeExample();
console.log(localVar);
```

Output:

I am a local variable

Error: localVar is not defined

Function Scope

Function scope is similar to local scope. Here, variables defined within a function are available inside that function and can't be accessed from outside that function. With the creation of a new function, JS has a new scope. Also, variables defined in a function are not available in the global scope. Inside a function, variables defined using var, let, and const work similarly. Functions use curly braces to store the body.

When to Use Function Scope

- For variables needed only within a specific function.

- For example, helper functions that are relevant only to a specific execution content of a function.
- Temporary variables used for calculations within a function.
- Variables with function scope are not accessible outside the function they are declared in, so it promotes encapsulation and avoids any unnecessary side effects.

Example

```
function functionScopeExample() {
   var funcVar = "I am scoped to this function";
   console.log(funcVar);
}
functionScopeExample();
console.log(funcVar);
```

Output:

I am scoped to this function

Error: funcVar is not defined

Block Scope

Prior to ES6 in 2015, there were only two scopes in JavaScript, Global and Local. With the introduction of const and let keywords, there were new additions, and block scope was one of them.

Variables declared inside a specific block (represented by {}) are not accessible from outside the block. It doesn't work with the var keyword, so you can use only let or const keywords with it.

When to USe Block Scope

- For variables with scope limited to a specific block of the code, such as for loops, while loops, and if statements.
- For example, temporary variables used for conditionally executing code blocks.
- Loop iterators relevant only within a loop.
- Variables with block scope are accessible only within the block they are declared in, which provides more control over variable lifetime and visibility.

Example

```
if (true) {
   let blockVar = "I am block scoped";
   const anotherBlockVar = "I am also block scoped";
   console.log(blockVar);
   console.log(anotherBlockVar);
}

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

Output

I am block scoped

I am also block scoped

Error: blockVar is not defined

Error: anotherBlockVar is not defined

Lexical Scope

It is a bit of a complex concept. A variable is determined by its place or position in the code. When a variable is declared within a function, it can be accessed only within that block or nested block. That variable is said to have lexical scope, which is also known as compile-time scope or static scope.

When to Use Lexical Scope

- It is inherent in JS and is automatically applied according to the placement of a code within the source file.
- For example, variables declared in the outer scope are accessible to nested functions. This allows maintaining access to variables even after the outer function returns.
- It allows powerful programming patterns, such as closures, which encapsulate state and create reusable code.

Example

```
function outerFunction() {
   let outerVar = "I am from the outer function";

   function innerFunction() {
      console.log(outerVar);
   }

   innerFunction();
}

outerFunction();
```

Output:

I am from the outer function

I am from the outer function

JavaScript if-else Statement

In JavaScript, you may need to choose one path out of multiple options while writing a program. For such scenarios, you must use conditional statements so programs can make correct decisions and take the right actions.

This chapter focuses on the **if-else statement in JavaScript**, a conditional statement used to perform actions according to different conditions.

What is the if-else statement in JavaScript?

The JavaScript if-else statement executes a block of code when a certain condition is true. If the condition is false, it executes the code within the else block. We use **if-else in JavaScript** to control the program's flow of execution depending on different conditions.

if-else Statements in JavaScript

JavaScript supports four types of if-else statements:

1. JavaScript if-statement

2. JavaScript if-else statement
3. JavaScript if-else-if ladder statement
4. JavaScript nested-if statement

In the following section, we will discuss each **JavaScript if-else** statement in detail, its syntax, flowchart, and examples. Let's get started.

1. JavaScript if Statement

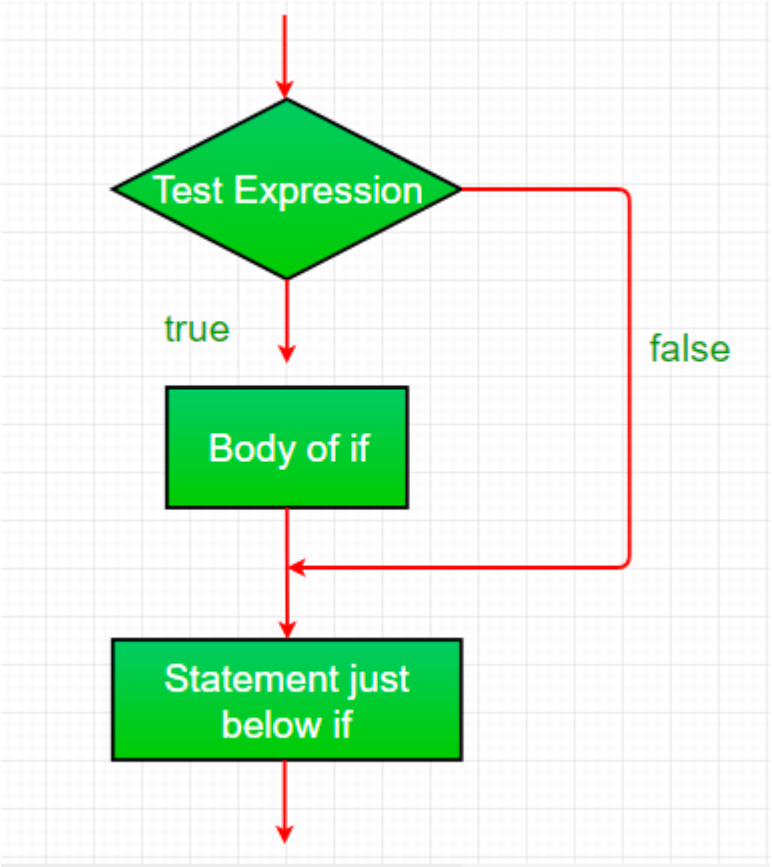
This conditional statement is used to decide whether to execute a certain block of code or not based on a specific condition. This fundamental control statement enables JavaScript to make decisions and execute statements conditionally. If the given condition is true, the statement is executed; otherwise, it is not.

Syntax

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, the condition is evaluated. If the resulting value is true, the statement under it is executed. If the resulting value is false, the code is skipped, and no statement is executed. If we don't provide curly braces after the if condition, the default if statement considers the immediate expression to be within the block. We mostly use comparison operators to make decisions.

Flowchart of the if Statement



Example

```
let age = 18;

if (age >= 18) {

    console.log("You are eligible to vote.");

}
```

Output:

You are eligible to vote.

JavaScript if-else statement

The if-else statement is another form of control statement that enables JavaScript to execute the code in a more controlled manner. The if statement tells the program to execute a block of code if the given condition is true, and if the condition is false, it won't execute the statement.

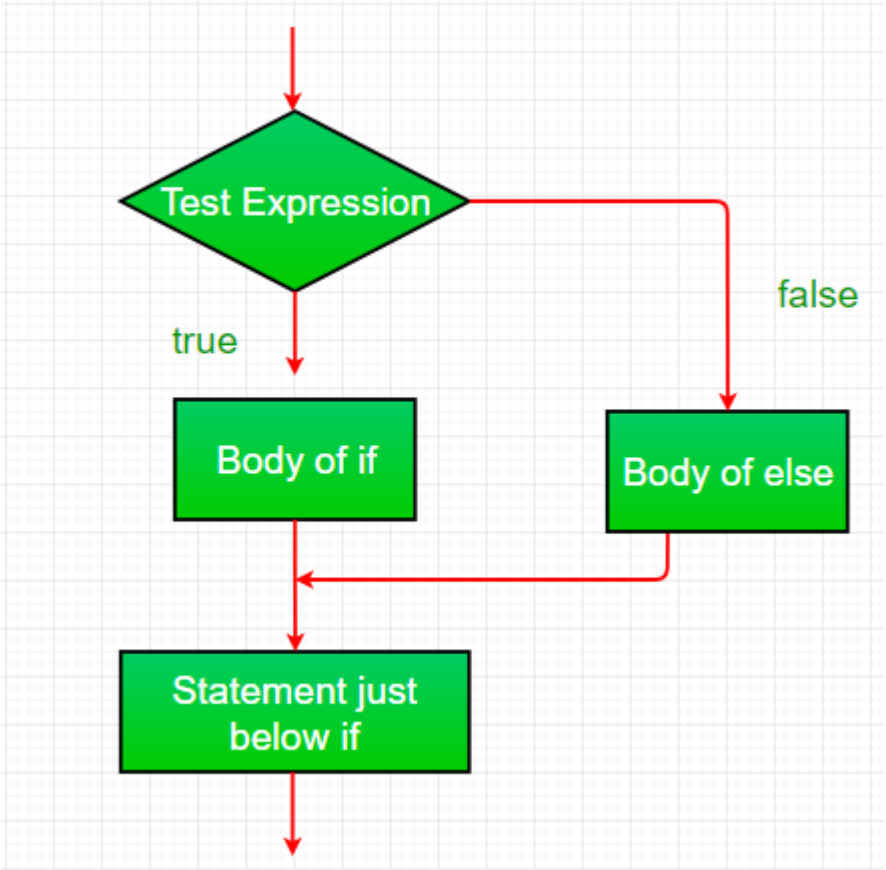
However, what if we want another action if the condition is false? Hence, we use **if-else in JS** to execute the block of code even if the condition is false.

Syntax

```
if (condition) {
    // statement to
    // execute if condition is true
}
else {
    // statement to
    // execute if condition is false
}
```

Here, the if-else statement evaluates the condition and executes the code. If the resulting value is true, the given code inside the if block is executed, and the else block is skipped. If the condition is false, the code inside the else block is executed.

Flowchart of the if-else Statement



Example

```
let age = 16;


if (age >= 18) {
   console.log("You are eligible to vote.");
} else {
   console.log("You are not eligible to vote.");
}
```

Output:

You are not eligible to vote.

JavaScript if-else-if ladder statement

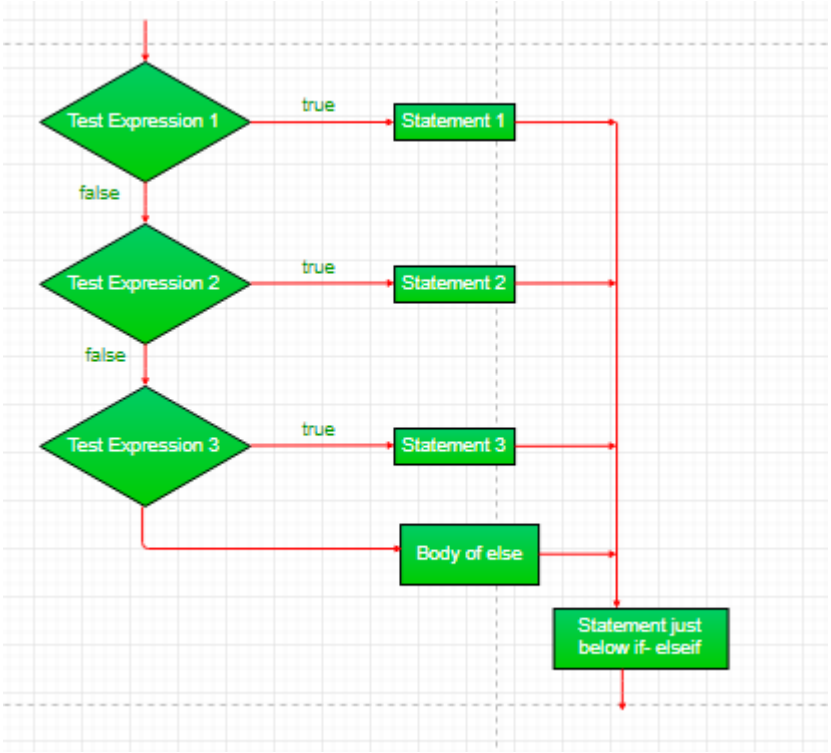
The if-else-if statement in JavaScript, or if-else ladder, is an advanced if-else statement that allows JavaScript to make decisions based on several conditions. The statements are executed from top to bottom. When the condition controlling the if evaluates to be true, the statement related to that block is also executed, and the rest of the blocks are bypassed. If no condition is true, the final else statement is executed.

Syntax

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    Statement;
```

Here, we use **else-if in JavaScript** to execute the code. If the condition is true, the body of if is executed, and the rest of the blocks are skipped. If the condition in the else-if statement is true, the given statement is executed. And if no condition is met, the code block in else is executed.

Flowchart of the if-else-if Statement



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");
}
```

Output:

Grade: A

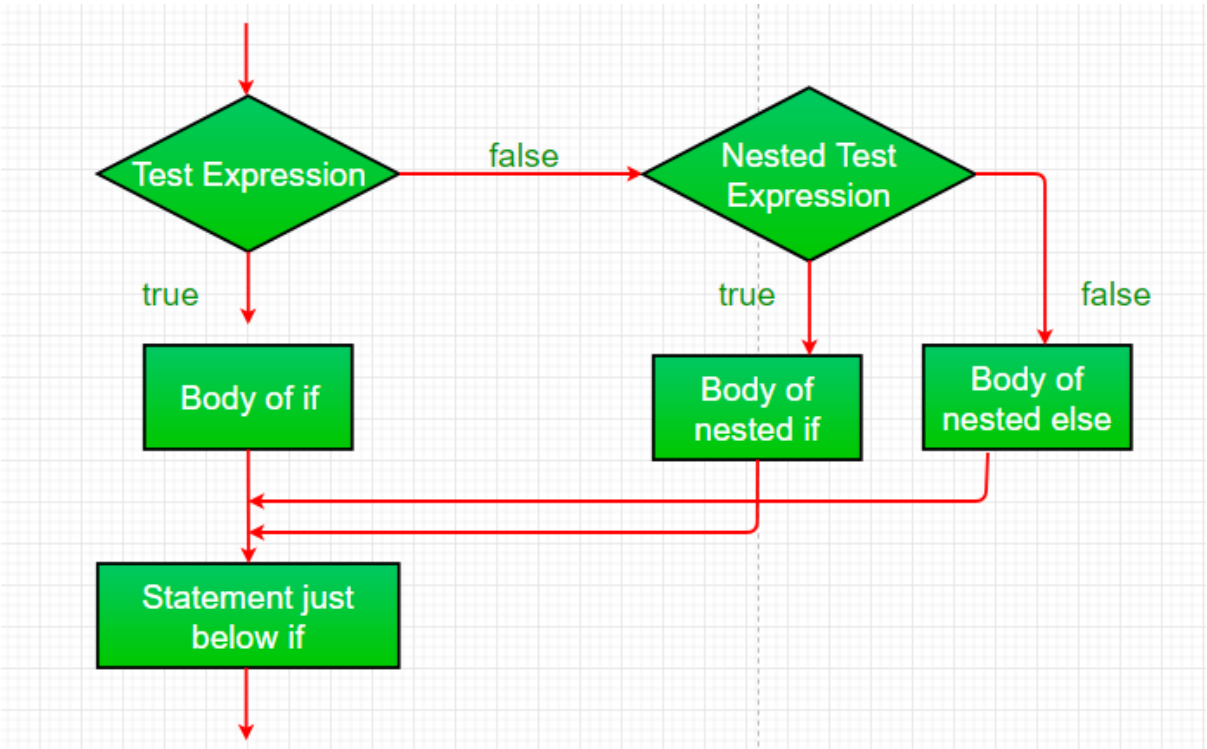
JavaScript nested-if statement

In JavaScript, we use the nested if statements within if statements, placing one statement inside another. A nested if is another type of an if statement that is the target of another if or else.

Syntax

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```

Flowchart of the Nested-if Statement



Example

```
let number = 15;


if (number > 0) {
    if (number % 2 === 0) {
        console.log("The number is positive and even.");
    } else {
        console.log("The number is positive and odd.");
    }
} else {
    console.log("The number is not positive.");
}
```

Output:

The number is positive and odd.

JavaScript Ternary Operator

In any programming language, we use different ways to handle conditional situations. The if statement is the most common method. The **ternary operator in JavaScript**, also known as the conditional operator, is a shortcut to replace simple if-else statements. It is based on a condition and allows you to decide between two values depending on whether the given condition is true or false.

The JS ternary operator improves code readability and simplicity and enables concise representation of conditional blocks with expressions. In this tutorial, you will learn about **JS ternary operators** in detail, their role in streamlining code, their benefits, examples, and more.

What is a Ternary Operator in JavaScript?

A **JavaScript ternary operator** offers an improved and better approach to expressing conditional statements. It evaluates a condition and executes a code block depending on the given condition. It is like a shorthand way of writing an if-else statement. It assigns a value to a variable based on the given condition.

A ternary operator evaluates a condition and executes a block of code based on it. It operates on three operands- a condition, a value to be returned if the condition is true, and the value to be returned if the condition is false. The condition is followed by the question mark symbol (?), and expressions are separated by the colon (:).

How Does the Ternary Operator Work in JS?

Now, let's understand how a **JavaScript ternary operator works**. It includes three operands:

Condition- A statement that returns either a true or false value.

Value if True- The value to return if the condition is true.

Value if False- The value to return if the condition is false.

Syntax

```
condition ? trueExpression : falseExpression
```

Parameters

Condition- It is an expression to be evaluated. It returns a boolean value.

trueExpression- An expression to execute if the condition evaluates to be true (a value that is equal to or can be converted to true).

falseExpression- An expression to execute if the condition is false (a value that can be converted to false).

Characteristics of a JavaScript Ternary Operator

It is the only operator to have three operands- the condition, value if true, and value if false.

- The condition is evaluated to give the result that is either true or false.
- The true value comes between “?” and “:”. It is executed if the condition is true.
- The false value comes after “:” and is executed if the condition is false.

Why Should You Use a Ternary Operator in JavaScript?

You may wonder **why we use a ternary operator in JavaScript** to write conditional statements when we already have if-else statements. Here are the main reasons to use a ternary operator:

Avoid Repetition

With a ternary operator, we can avoid repeating the same block of code for different branches of a condition.

Inline Usage

Ternary operators in JS are helpful for assigning values to variables inline or assessing a condition and returning a value without the complete if statement.

Concise

We can express conditional logic in a more concise and readable manner, especially when we need to mention a simple condition.

Functional Style

We prefer using a ternary operator over the traditional if statement for functional programming because it aligns with the functional paradigm of avoiding mutable states.

Improved Readability

Ternary conditions make the code more readable as they keep the logic inline, whereas an if statement breaks it into multiple lines.

Ternary Operator Instead of if-else

We use a ternary operator in JavaScript to replace the traditional if statements.

Example - 1

```
let age = 18;

let canVote = age >= 18 ? "Yes, eligible to vote" : "No, not eligible to vote";

console.log(canVote);
```

Output:

Yes, eligible to vote

Example - 2

```
let isLoggedIn = true;

let message = isLoggedIn ? "Welcome back!" : "Please log in.";

console.log(message);
```

Output:

Welcome back!

Nested Ternary Operators

Similar to nested if-else statements, we can nest one ternary operator within another. This is known as nested ternary operators.

Example

```
let marks = 85;


let grade = marks >= 90

   ? "A+"

   : marks >= 80

      ? "A"

      : marks >= 70
```

```
        ? "B"
        : "F";

console.log(grade);
```

Output:

A

Multiple Ternary Operators

We can use a ternary operator to improve code maintainability and readability. However, when we use multiple ternary operators, we must be careful.

Example 1

```
let num = 0;

let result = num > 0
   ? "Positive"
   : num < 0
      ? "Negative"
      : "Zero";

console.log(result);
```

Output:

Zero

Example - 2

```
function checkEvenOdd(number) {
   return number % 2 === 0 ? "Even" : "Odd";
}
```

```
console.log(checkEvenOdd(10));
console.log(checkEvenOdd(7));
```

Output:

Even

Odd

Example - 3

```
let isWeekend = true;

let message = `Today is ${isWeekend ? "a weekend" : "a weekday"}.`;
console.log(message);
```

Output:

Today is a weekend.

Example - 4

```
let temperature = 30;

let weather = temperature > 35
    ? "Hot"
    : temperature > 20
        ? "Warm"
        : "Cold";

console.log(weather);
```

Output:

Warm

Example - 5

```
const getDiscount = (age) => age >= 60 ? "Senior Citizen Discount" : "No Discount";

console.log(getDiscount(65));

console.log(getDiscount(30));
```

Output:

Senior Citizen Discount

No Discount

Advantage of Ternary Operator in JavaScript

Here are the benefits of using the ternary operator in JavaScript:

Simplified Logic

It simplifies the decision-making logic, so we can maintain and understand it easily.

Clarity

When used effectively, a ternary operator can bring code clarity by expressing the condition and output in a single line.

Efficiency

It makes the code more efficient by evaluating conditional statements clearly and quickly.

JavaScript Loops

A loop is a powerful programming tool used to repeat a set of instructions. It continues to iterate until a given condition is met. This specific condition is known as the stopping condition. Similarly, **loops in JS** are used to iterate a piece of code.

This chapter of our JavaScript tutorial delves deeper into **JS loops**, different types of loops, and more.

What is a Loop in JavaScript?

JavaScript loops are efficient ways to handle repetitive tasks. They are used to iterate a piece of code as long as the given condition is met. Loops make the code compact and are used in arrays. Loops are also known as iterative statements. Loops automate tasks and streamline the code.

Types of Loops in JavaScript

Here are the different types of loops in JS that programmers work with:

- for Loop
- while Loop
- do-while Loop
- Nested for Loop
- for-in Loop
- for-of Loop
- Labeled Statement
- Break Statement
- Continue Statement
- Infinite Loop (Loop Error)

JavaScript for Loop

For loops in JavaScript are a concise way of writing the loop structure. It iterates the elements for a specific number of times and is used only when the number of iterations is known. The for loop executes the code block until the given condition is true.

The for loop in JS contains initialization, condition, and increment/decrement in a single line, which makes the code shorter and easy to debug.

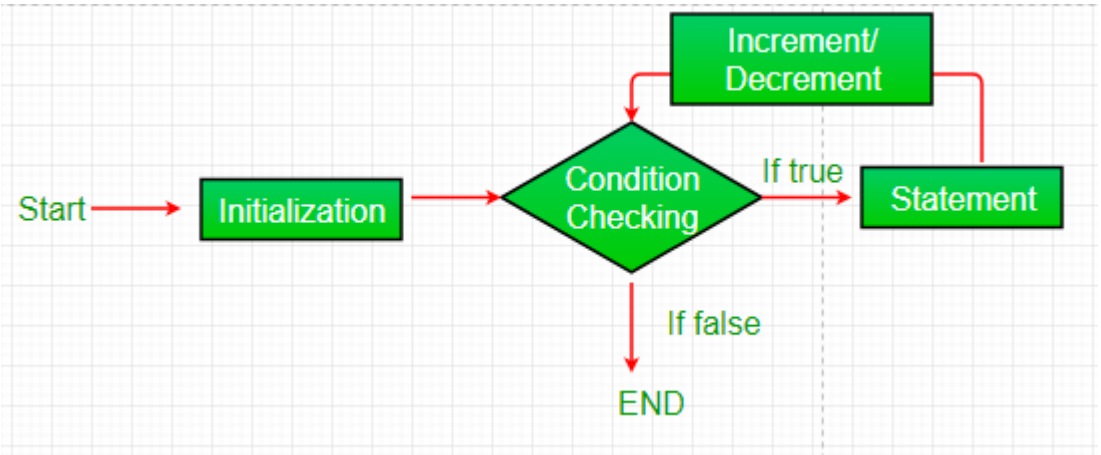
The three important aspects in a for loop are separated by semicolons and provide the following critical information:

- The initialization states where to begin the loop by declaring the iterator variable.
- The condition, or stopping condition, decides when to stop looping (once the condition evaluates to be false).
- The iteration statement updates the iterator every time the loop is completed.

JavaScript for loop Syntax

```
for (initialization; testing condition; increment/decrement) {
    statement(s)
}
```

Flowchart



Initialization Condition- It initializes the variable and marks the beginning of the for loop. We can use a declared variable or declare one.

Test Condition- Tests the exit condition of the for loop and returns a boolean value. This Entry Control Loop checks the condition prior to the execution of the **loop statements in JavaScript**.

Statement Execution- When the condition is evaluated to be true, the statements in the body of the loop are executed.

Increment/Decrement- It updates the variable for the next iteration.

Loop Termination- Once the condition is evaluated to be false, the loop stops, and its lifecycle ends.

Example

```
for (let i = 0; i < 5; i++) {
   console.log("Iteration number: " + i);
}
```

Output

Iteration number: 0

Iteration number: 1

Iteration number: 2

Iteration number: 3

Iteration number: 4

JavaScript while Loop

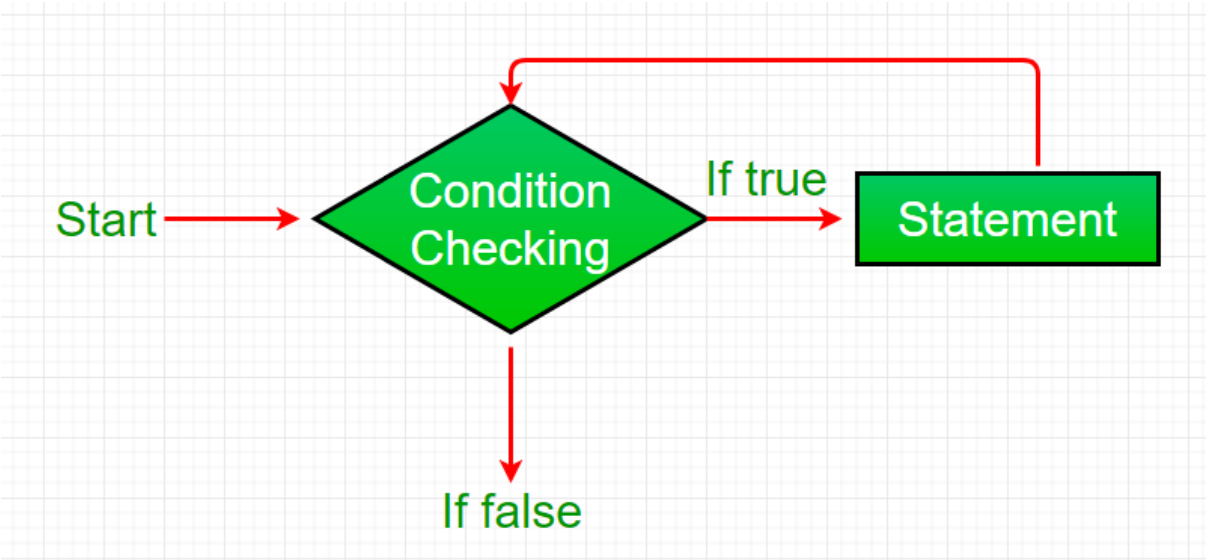
The while loop in JavaScript is a control flow statement used to execute the code repetitively based on the boolean condition. It is like a repeating if statement. The while loop iterates the elements an infinite number of times and is used when we don't know the number of iterations. In the while loop, the condition is mentioned before the loop, which is checked first, and if it is true, the statement within the curly braces is executed. The variable in the while loop is altered to determine when to stop the loop.

JavaScript while loop Syntax

```
while (boolean condition) {
   loop statements...
```

```
}
```

Flowchart



- First, the while loop checks the condition and if it is true, it executes the statements within the loop body; else, it executes the first statement following the loop. Hence, it is also known as the Entry Control Loop.
- When the condition is true, it executes the loop body.
- The statements include an updated value for the variable processed for the next iteration.
- If the condition is false, the loop stops, making the end of its lifecycle.

Example

```
let i = 0;

while (i < 5) {

   console.log("While loop iteration: " + i);

   i++;

}
```

Output

While loop iteration: 0

While loop iteration: 1

While loop iteration: 2

While loop iteration: 3

While loop iteration: 4

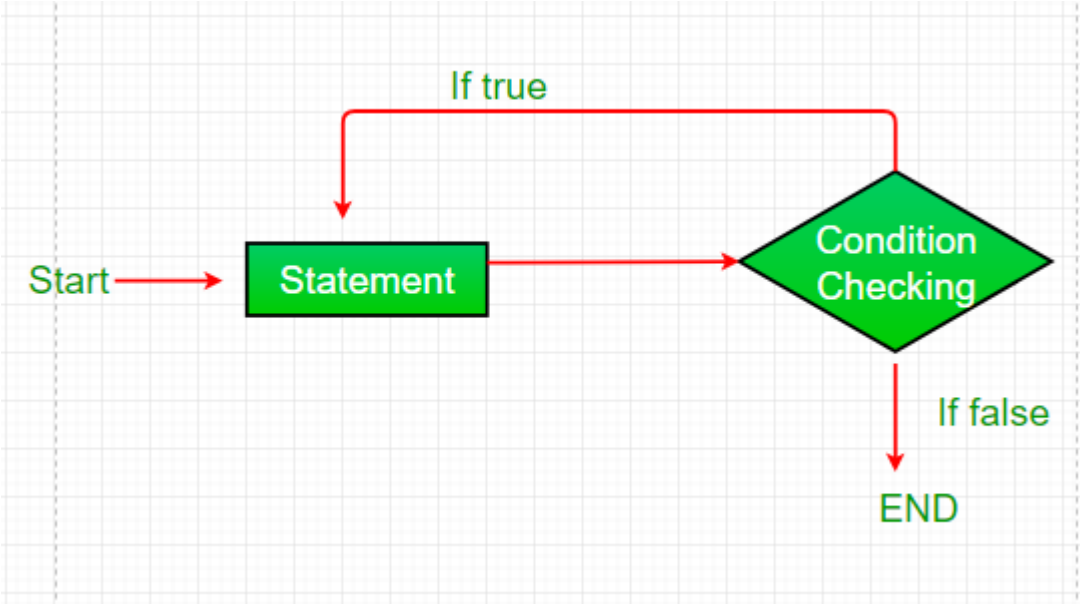
JavaScript do-while loop

The do-while loop in JavaScript is similar to the while loop. However, the only difference is that the do-while loop evaluates the condition after executing the statement and executes the loop at least once, even if the condition is false. It executes the code block, checks the condition, and repeats the loop until the condition remains true. It iterates the elements infinitely. As the test condition is checked at the end of the loop, the body is executed unconditionally for the first time and ends the loop when the condition is false. We use it when we need to execute the loop body at least once.

JavaScript do-while loop Syntax

```
do {
    Statements...
}
while (condition);
```

Flowchart



- First, the do-while loop executes the statement without checking the condition.
- After executing the statement once and updating the variable value, it checks the condition. If the condition is evaluated to be true, it starts the next iteration of the loop.
- If the condition is false, the loop stops, marking the end of its lifecycle.
- The do-while loop executes the statement at least once before evaluating any condition, making it an example of the Exit Control Loop.

Example

```
let i = 0;

do {

   console.log("Do-while iteration: " + i);

   i++;

} while (i < 5);
```

Do-while iteration: 0

Do-while iteration: 1

Do-while iteration: 2

Do-while iteration: 3

Do-while iteration: 4

Nested for Loops

A for loop inside another is known as a nested for loop. For every outer loop cycle, the inner loop completes the entire iteration sequence.

Example

```
for (let i = 1; i <= 3; i++) {
   for (let j = 1; j <= 3; j++) {
      console.log(`Outer loop: ${i}, Inner loop: ${j}`);
   }
}
```

Output

Outer loop: 1, Inner loop: 1

Outer loop: 1, Inner loop: 2

Outer loop: 1, Inner loop: 3

Outer loop: 2, Inner loop: 1

Outer loop: 2, Inner loop: 2

Outer loop: 2, Inner loop: 3

Outer loop: 3, Inner loop: 1

Outer loop: 3, Inner loop: 2

Outer loop: 3, Inner loop: 3

JavaScript for-in loop

The for-in loop in JS is used to iterate over the properties of an object. It iterates only over string type keys of an object with their enumerable property set to ‘true’. It allows access to the values by index-reference.

JavaScript for-in loop Syntax

```
for(let variable_name in object_name) {
    // Statement
}
```

Example

```
const obj = { name: "John", age: 30, city: "New York" };
for (let key in obj) {
   console.log(`${key}: ${obj[key]}`);
}
```

Output

name: John

age: 30

city: New York

JavaScript for-of Loop

The for-of loop in JavaScript is used to iterate the iterable objects, such as objects, sets, arrays, and maps. It directly iterates the value of the object instead of keys, allowing direct access to the items. Also, it has a more concise syntax than the for loop. A few examples of iterable objects include a string of characters, an array of elements, and a map of key-value pairs.

JavaScript for-of loop Syntax

```
for(let variable_name of  object_name) {
    // Statement
}
```

Example

```
const arr = ["a", "b", "c"];
for (let value of arr) {
   console.log(value);
}
```

Output

```
a
b
c
```

JavaScript Labeled Statement

The labeled statement in JavaScript doesn't have a goto keyword, so you can use continue with the label statement. You can also use the break keyword to terminate the loop. You can use break without defining a label but it will stop only the parent loop. If you want to terminate the outer loop using the break keyword, you must define the label.

JavaScript labeled Statement Syntax

```
Label:
   statement (loop or block of code)
```

Example

```
outerLoop: for (let i = 1; i <= 3; i++) {
   for (let j = 1; j <= 3; j++) {
      if (i === 2 && j === 2) {
         break outerLoop;
      }
      console.log(`i: ${i}, j: ${j}`);
   }
}
```

Output

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
```

JavaScript break Statement

The break statement in JavaScript terminates the loop execution or switch statement when the condition is evaluated to be true.

JavaScript break Syntax

```
break;
```

Example

```
for (let i = 0; i < 10; i++) {
```

```
    if (i === 5) {
        break;
    }
    console.log(i);
}
```

Output

0

1

2

3

4

JavaScript continue Statement

The continue statement in JavaScript is used to break the loop iteration and continue with the next iteration. It breaks the iteration only when the given condition is met. Unlike break that jumps out of the loop completely, continue only jumps over the current iteration and moves on to the next statement.

JavaScript continue Syntax

```
continue;
```

Example

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue;
    }
```

```
    console.log(i);
}
```

Output

```
1
3
5
7
9
```

JavaScript Infinite Loop (Loop Error)

One common mistake you may encounter while implementing a loop is that it may not exist, meaning the loop runs infinitely. This happens when the condition fails due to any reason.

Example

```
let i = 0;
while (i < 5) {
    console.log("This is an infinite loop.");
}
```

Output

```
This is an infinite loop.
This is an infinite loop.
...
```

Explanation

1. **The value of `i` is never updated, so the condition `i < 5` is always true.**

2. **The while loop checks the condition i < 5.**
3. **Inside the loop, `i` is not incremented or changed.**
4. **As a result, the condition remains `true` forever, and the loop never exits.**

//

JavaScript while and do...while Loop

In programming, we may need to perform a specific task over and over again. In such cases, we use loop statements that reduce the number of code lines by automating the repetitive tasks. Loops are one of the most useful and crucial features of any programming language.

JavaScript supports all essential types of loops required to make programming easier and hassle-free. **While and do…while loop in JavaScript** are similar to conditional statements, which refer to code blocks that execute if the given condition is true.

In this chapter, we will focus on the **while loop in JavaScript**, including do…while and infinite while loops, through their syntaxes, flowcharts, and examples.

Types of Loops in JavaScript

There are primarily two types of loops:

Entry-controlled Loops- These loops check the condition before entering the body of the loop and then execute the statements. For and while loops are entry-controlled loops.

Exit-controlled Loops- These loops enter the body of the code, execute the loop statements, and then test the condition at the end after completing the iteration. Hence, they execute the loop body at least once, no matter if the condition is true or false. The do-while loop is the exit-controlled loop.

JavaScript while Loop

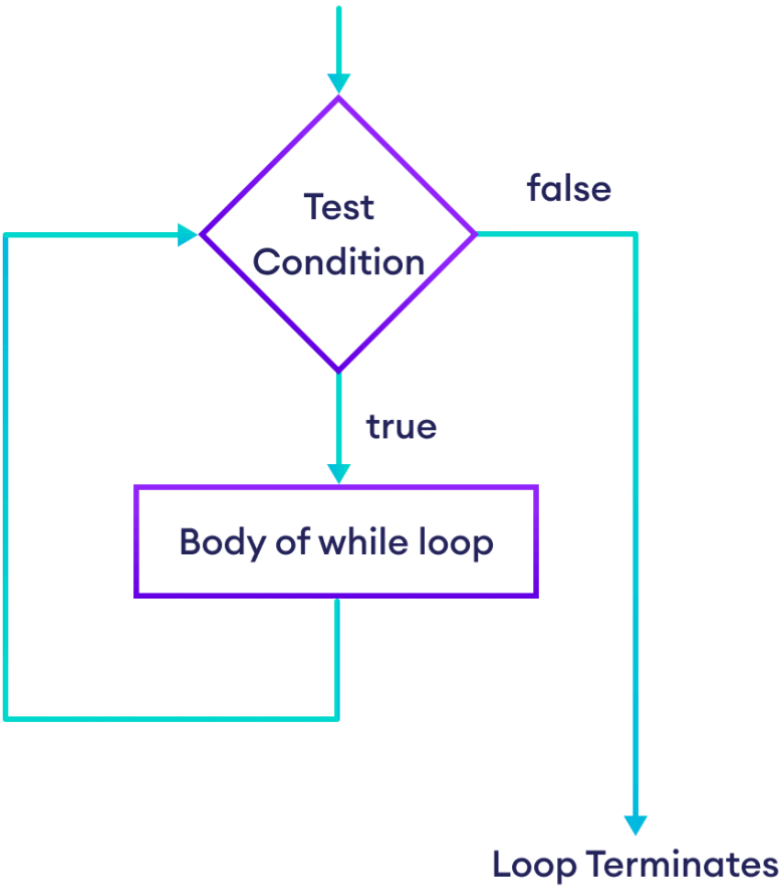
In the **while loop in JS,** the code block is repeatedly executed as long as the condition remains true. It evaluates the given condition before every iteration and continues running until the condition is true. If the condition evaluates to be false, the loop terminates. Hence, it enables dynamic and repeated operations according to the changing conditions.

JavaScript while Loop Syntax

```
while (condition) {
    Code block to be executed
}
```

- First, the JavaScript while loop tests the condition given inside ().
- If the condition is true, the code within curly braces {} is executed.
- The condition is evaluated again, and the process continues until the condition remains true.
- If the condition becomes false, the loop terminates.

Flowchart of JavaScript while Loop



JavaScript while Loop Example

```
let number = 12345;

let sum = 0;

console.log(`Initial number: ${number}`);


while (number > 0) {

    let digit = number % 10;

    sum += digit;
```

```
    number = Math.floor(number / 10);
    console.log(`Extracted digit: ${digit}, Remaining number: ${number}`);
}

console.log(`The sum of the digits is: ${sum}`);
```

Output

Initial number: 12345

Extracted digit: 5, Remaining number: 1234

Extracted digit: 4, Remaining number: 123

Extracted digit: 3, Remaining number: 12

Extracted digit: 2, Remaining number: 1

Extracted digit: 1, Remaining number: 0

The sum of the digits is: 15

JavaScript do…while Loop

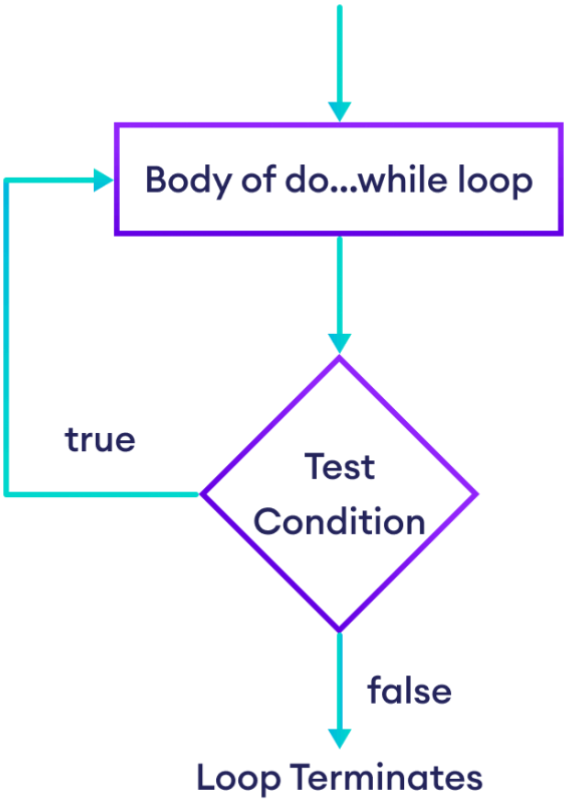
The **do-while loop in JavaScript** is similar to the while loop, but in this control structure, the code is executed at least once, no matter whether the condition is true or false. After this, it executes the code as long as the condition remains true.

JavaScript do…while Loop Syntax

```
do {
        // code block to be executed
 } while (condition);
```

- The do…while in JavaScript executes the code within {}.
- After this, it evaluates the condition given inside ().
- If the condition is true, it executes the code inside () again.
- It repeats the process as long as the condition evaluates to be true.
- If the condition becomes false, the loop stops.

Flowchart of JavaScript do...while Loop



JavaScript do…while Loop Example

```
const correctPassword = "jsrocks";
let userPassword;
```

```
do {
   userPassword = prompt("Enter your password:");
   if (userPassword !== correctPassword) {
      console.log("Incorrect password. Please try again.");
   }
} while (userPassword !== correctPassword);

console.log("Access granted. Welcome!");
```

Output

```
Enter your password: hello123
Incorrect password. Please try again.
Enter your password: jsrocks
Access granted. Welcome!
```

Infinite while Loop in JavaScript

As the name suggests, an infinite while loop is the one that runs forever because the given condition is always true. In an infinite loop, the condition of the while statement is set to true. It is important to be aware of the infinite loop to avoid them. If you accidentally create an infinite loop and are unaware of it, the loop can crash the browser or computer.

Example

```
while (true) {

   console.log("This is an infinite loop!");

}
```

Output

This is an infinite loop!

This is an infinite loop!

This is an infinite loop!

... (keeps running indefinitely)

Difference Between while and do…while loop in JavaScript

In the **JS while loop**, the condition is checked first before executing the code. On the other hand, in the do…while loop, the code block is executed at least once before checking the condition. Both loops share a few more differences that we have discussed in the following table:

Parameter	**while loop**	**Do…while loop**
Block of Code	The block control condition is available at the start of the loop.	The block control condition is available at the end of the loop.
Type of Loop	It is an entry-controlled loop.	It is an exit-controlled loop
Number of Iterations	The number of iterations depends on the condition specified in the while block.	The number of iterations is at least one, irrespective of the condition mentioned in the do…while block.

JavaScript Break Statement

The **break statement in JavaScript** is used to terminate loops and statements immediately once the break keyword is encountered.

This tutorial will explore the break statement, its uses, and examples.

What is a Break Statement in JavaScript?

The **JavaScript break statement** terminates the loop or the switch case statement. When used with a switch, it breaks the code and stops code execution. When used within a loop, it breaks the loop, and the control flow jumps out of it but continues executing the code after the loop.

We can also use the break statement within the labeled statement to jump a labeled statement. It controls the flow of execution in a JS code.

Syntax

```
break;
```

Using Labels

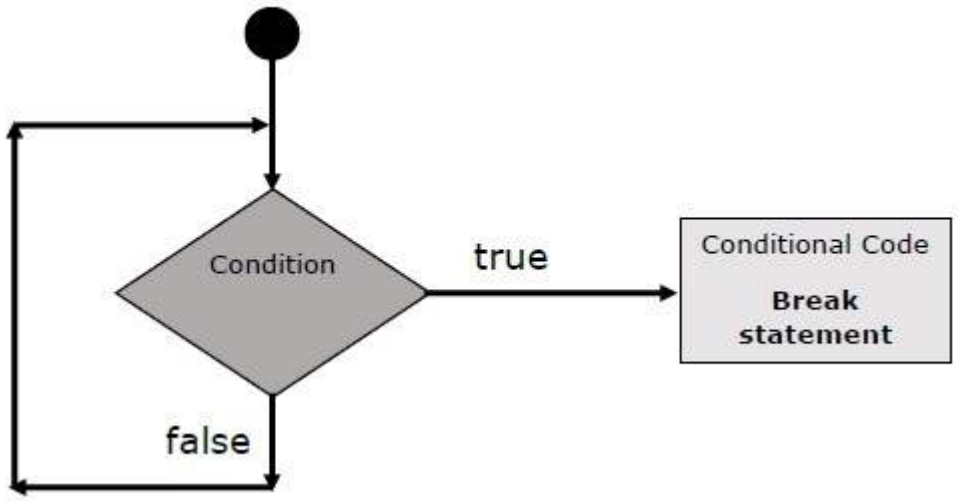
The break statement also uses a label reference to exit a code block. We can use a loop or switch with the break in the absence of a label.

Syntax

```
break labelName;
```

Flowchart of the Break Statement in JS

Here is the flowchart of the **JS break statement**:



Working of the JavaScript Break Statement

The following image demonstrates how the JavaScript break statement works in for and while loops.

```
for (initialization; condition; update) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
// code after for loop
```

```
while (condition) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
// code after while loop
```

JS Break Statement Examples

Example 1 (JavaScript break statement with for loop)

```
for (let i = 0; i < 5; i++) {
 if (i === 3) {
 break;
 }
 console.log(i);
```

```
}
```

Output:

0

1

2

Example 2 (JavaScript break with while Loop)

```
let i = 0;
while (i < 5) {
 if (i === 3) {
 Break;
}
console.log(i); i++;
 }
```

Output:

0

1

2

Example 3 (JavaScript break with a nested loop)

```
for (let i = 0; i < 3; i++) {
 for (let j = 0; j < 3; j++) {
 if (j === 1) {
break;
}
```

```
console.log(`i: ${i}, j: ${j}`);
} }
```

Output:

i: 0, j: 0

i: 1, j: 0

i: 2, j: 0

Explanation:

We can also use the break statement to jump out of any loop in nested loops. When we use break in JS within two nested loops, it terminates the inner loop. When we use break with the parent loop, it terminates all iterations of the nested loop. However, when we use break with the nested loop, it terminates only the nested loop.

Example 4 (JavaScript break with labels)

```
outerLoop: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      break outerLoop;
    }
    console.log(`i: ${i}, j: ${j}`);
  }
}
```

Output:

i: 0, j: 0

i: 0, j: 1

i: 0, j: 2

i: 1, j: 0

Explanation:

Using labeled break with nested loops terminates the outer loop.

Example 5 (JavaScript break in a switch statement)

```
let fruit = "apple";


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

Output:

It's an apple!

Explanation:

We can use break within a switch statement to terminate a case. The switch case statement executes one block of code out of multiple blocks based on the conditional expression. The break statement matches one or more cases with the value of the conditional expression and terminates the switch case statement.

JavaScript switch case Statement

In the previous chapters, we learned to use if-else statements to perform a multiway branch. However, using if-else is not always the best method, especially when branches depend on the value of a single variable. Hence, we use the **JavaScript switch statement**, which handles such situations more effectively than if-else statements.

The switch…case statement in JavaScript executes different code blocks based on the value of the given expression. Let's explore the switch statement more, including its syntax, flowchart, and examples.

What is a switch case in JavaScript?

The **JavaScript switch case statement** is a conditional statement used to execute an expression and blocks of code based on the conditions and value of an expression. It evaluates the expression, and if it matches the value of the case label, the associated code block is executed. If no case label matches the value of the expression, the block of code associated with the default label is executed. In other words, the switch case statement performs different actions depending on various conditions present in the program.

The **switch case in JS** is an alternative to the long if-else statements that improves the readability and maintainability of the code while handling multiple conditional branches.

JavaScript Switch Statement Syntax

The **JS switch case** statement gives an expression to evaluate and execute code blocks based on the value of the expression. The interpreter checks every case against each value of the expression to find the match. If nothing matches, it uses the default condition.

```
switch (expression) {
   case value1:
      // code block 1;
      break;
   case value2:
       // code block 2;
      break;
  ...
   default:
     // default code block;
}
```

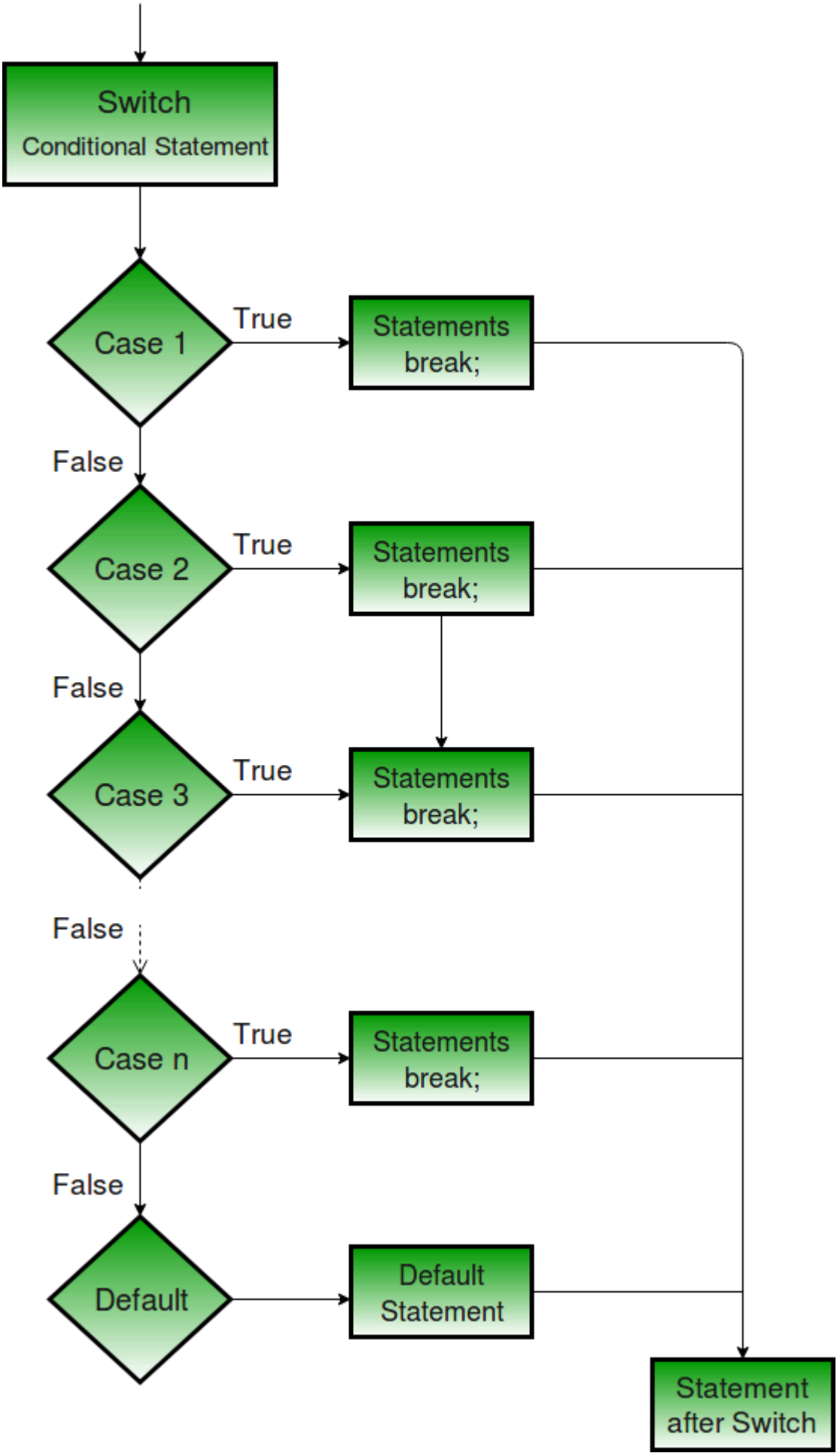
expression- Value to compare.

case value1, case value2- Possible values of the expression.

break- Indicates the end of the specific case and terminates the switch statement. If break is omitted, the interpreter will continue to execute the statement in the next case.

default- It represents the default expression or code to run if no match is found. When no case matches the expression of the switch case statement, the default code block is executed.

Flowchart of Switch Statement


Switch
Conditional Statement
Case 1
True
Statements
break;
False
Case 2
True
Statements
break;
False
Case 3
True
Statements
break;
False
Case n
True
Statements
break;
False
Default
Default
Statement
Statement
after Switch

How Does a Switch Statement Work in JavaScript?

Evaluation

The expression within the switch statement is evaluated once.

Comparison

The expression value is compared with every case label using the strict equality operator (===).

Execution

Once the match is found, the associated code block following the matching case label is executed. If no match is found, the default case or the next statement after the switch block is executed.

Break Statement

After the code block is executed, the break statement terminates the switch statement, which stops the execution from falling through to subsequent cases. If break is not present, execution continues to the next case.

Default Case

It is optional, and the programmer can decide whether to use it or not. When no match is found, the code block under the default case is executed.

Switch Statement Example

```
let fruit = "apple";


switch (fruit) {
  case "banana":
    console.log("Banana is yellow.");
    break;
  case "apple":
    console.log("Apple is red.");
    break;
  case "grape":
    console.log("Grapes are purple.");
```

```
    break;
  default:
    console.log("Unknown fruit.");
}
```

Output:

Apple is red.

break Keyword

We use the break keyword in JS to terminate the execution of the loop or switch statement.

Example

```
let number = 2;

switch (number) {
  case 1:
    console.log("One");
  case 2:
    console.log("Two");
  case 3:
    console.log("Three");
    break;
  default:
    console.log("Default case");
}
```

Output:

Two

Three

default Keyword

If no case expression matches the value, we use the default keyword within a switch statement in JavaScript as a fallback option. It works similarly to the else statement, offering a default action when no specific case matches.

Example

```
let weather = "sunny";


switch (weather) {
  case "rainy":
    console.log("Take an umbrella!");
    break;
  case "cloudy":
    console.log("It might rain later.");
    break;
  default:
    console.log("Enjoy your day!");
}
```

Output:

Enjoy your day!

Position of the default case does not matter

No matter where the default case is placed, it is executed only if none of the case conditions are met. Whether you place it in the start, end, or middle, it won't change the core logic.

Example

```
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("Grade: F");
}
```

Output:

Grade: B

Common Code Blocks

In JavaScript, a few cases require us to use the same code for multiple switch cases.

Example

```
let day = "Saturday";

switch (day) {
  case "Saturday":
```

```
  case "Sunday":
    console.log("It's the weekend!");
    break;
  case "Monday":
    console.log("Start of the workweek.");
    break;
  default:
    console.log("It's a regular weekday.");
}
```

Output:

It's the weekend!

Difference Between if-else and switch Statements

Here is a detailed comparison between the if-else statement and the switch statement:

Parameter	**if-else**	**switch**
Purpose	It checks the single condition and executes different code blocks based on whether they are true or false.	It checks the single expression with multiple cases and executes the block depending on the matching case.
Readability	If we compare it with the switch condition when a complex condition is to be checked, if-else is more powerful and flexible.	If compared to the if-else statement when multiple statements are to be checked, the switch statement is easier to read.
Syntax	The if keyword is followed by a condition inside parentheses and an optional else clause.	The switch keyword is followed by an expression in parentheses and a series of case statements.
Efficiency	Less efficient than the switch	More efficient if more cases are to

	statement because of multiple cases.	check the statement.
Cases	Two possible code blocks- one for the true condition and another one for false.	Multiple cases and each case has a different value to match the expression.

We can conclude that both if-else and switch statements have pros and cons. You must choose either of them based on your specific programming requirements.

JavaScript Continue Statement

JavaScript offers complete control to handle the loop statements. However, sometimes we must skip some code of the loop and move to the next iteration. Hence, we use the **continue statement in JavaScript.** It skips the current iteration of a loop and continues the next iteration. The continue statement is used in conjunction with the if statement to evaluate a condition, and if it is met, it skips the iteration.

What is a Continue Statement in JavaScript?

The **JavaScript continue** statement is used to skip the iteration of the loop and continue the next iteration. It breaks the iteration only when the given condition is met. We can use it with the while loop, for loop, and do-while loop. When used with the while loop, the continue statement jumps back to the condition. When we use continue in JS, the program's flow is immediately moved to conditional expression. If the condition is true, it starts the next iteration; else, the control exits the loop.

The continue statement tells the JavaScript interpreter to start the next iteration immediately and skip the rest of the code block. When it encounters the continue statement, the flow of the program moves to the loop check expression and starts the next iteration if the condition is evaluated to be true.

One major difference between break and continue is that the former terminates the loop completely, whereas the latter breaks one statement and iterates to the next one.

JS Continue Statement Syntax

```
continue;
```

Using the label reference (the label is optional)

```
continue[label];
```

The **JS continue statement** has an optional label to skip to the next iteration of the labeled loop statement rather than looping through the current loop. The continue statement behaves differently based on the type of loop being executed:

- In the for loop, it evaluates the increment expression and then checks if the next iteration should occur based on the given condition.
- In the while loop, the loop is executed only if the given condition is true.

Examples of JavaScript Continue Statements

JavaScript continue with for loop

In the for loop, the continue statement skips the iterations.

Example

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue;
  }
  console.log(i);
}
```

Output:

1

2

4

5

JavaScript continue with while loop

We use continue with a while loop as shown in the example.

Example

```
let i = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
  console.log(i);
}
```

Output:

1

2

4

5

JavaScript Continue with the nested loop

When we use continue with the nested loop, it skips the iteration of the parent loop or child loop. It affects the innermost loop when used with nested loops.

Example

```
for (let i = 1; i <= 2; i++) {
  for (let j = 1; j <= 3; j++) {
    if (j === 2) {
      continue;
```

```
    }
    console.log(`i = ${i}, j = ${j}`);
  }
}
```

Output:

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
```

JavaScript continue with labels

We can skip iterations of the outer loop by using the labeled continue statement in JS.

Example

```
outerLoop: for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    if (i === 2 && j === 2) {
      continue outerLoop;
    }
    console.log(`i = ${i}, j = ${j}`);
  }
}
```

Output:

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
```

i = 2, j = 1

i = 3, j = 1

i = 3, j = 2

i = 3, j = 3

JavaScript Functions

Like other programming languages, JavaScript supports different features required to write modular code using functions. These functions are reusable blocks of code containing instructions to perform a specific task. We define a function once and run it whenever we need it to perform a task in the program.

Once we write a function, we can use it again and again. As functions are the foundation of JavaScript programming, it's important for every beginner and developer to understand them in detail. So, let's explore **JS functions** through examples and more.

What are JavaScript Functions?

A function in JS is an independent block of reusable code that can be called anywhere in the program to perform a specific task. With functions, we don't need to write the same code again and again and can write modular code. Moreover, functions enable us to break a big program into multiple small and manageable functions.

Function expression is a method to store functions in variables.

Key points to note:

- Functions are the fundamental building blocks in JS programming.
- They are composed of a sequence of statements known as a function body.
- Each JavaScript function is a function object.
- To use a function, we first define it in the scope and then call it.
- A function can take one or more inputs and return output. However, taking input and returning output are optional.

How to Define a Function in JavaScript?

To use **functions in JS,** we first define or create them. The most common method to define a function is to use the function keyword, followed by the unique function name, a list of parameters within parentheses, and a body of the function or statement block within curly braces {}.

Defining a function is also called function declaration or function statement. We can use JS functions in the same way as variables for assignments or calculations.

Here are the rules for creating functions in JavaScript:

- The name of the user-defined function must be unique.
- Each function should start with the keyword function.
- The list of parameters must be enclosed within parentheses and separated by commas.
- Statements list composing the body of the function must be enclosed within curly braces {}.

JavaScript Functions Syntax

The following is the syntax for creating or declaring a function in JS:

```
function functionName(Parameter1, Parameter2, ...)
{
    // Function body
}
```

Example

```
function greet() {
  console.log(`Hello, Aman}!`);
}
```

greet();

Output:

Hello, Aman!

Why use functions in JavaScript?

- Functions break complex programs into small manageable chunks.
- We can use them multiple times, which reduces redundancy.
- They can call themselves to solve a problem recursively.
- They hide implementation details, which help manage complexity.

Function Invocation

We can execute a function code by calling it:

- We can explicitly call a function from JS code.
- It can be triggered by an event.
- It can be executed automatically.

Function Expression

A function expression is the same as a function declaration, including its syntax. However, the difference between the two is the function name, which we can omit in function expressions to create an anonymous function. We use the function expression to define a function as an expression. Also, it can be stored in a variable assignment.

Syntax

```
let Wscube= function(paramA, paramB) {
    // Set of statements
```

```
}
```

Example

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

console.log(add(5, 3));
```

Output:

8

Calling a Function

Defining a function is not enough to execute it. When we define a function, we simply assign it a name and specify what it will do once it is called. We must call a function so it can perform the specified task with the indicated parameters.

So, after defining a function, we need to call it to use it. To invoke a function, we need to write the name of that function followed by the value of parameters enclosed within parentheses and a semicolon at the end.

Syntax

```
functionName( Value1, Value2, ..);
```

Example

```
function add(a, b) {
  console.log(a + b);
}

add(5, 3);
add(10, 20);
```

Output:

8

30

Functions as Variable Values

We can use functions in the same way as variables. That is why functions in JS are known to be first-class and can be passed as arguments to other functions.

Example

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

console.log(greet('Aman'));
```

Output:

Hello, Aman!

Arrow Function

The arrow function is easy to implement, making it a commonly used and efficient way to create a function in JavaScript. It has a simpler and shorter syntax than a function expression and doesn't have its arguments, new.target, or super. It is a compact version of a regular function expression but is restricted as it can't be used in all situations. Arrow functions are always anonymous.

Syntax

```
let function_name = (argument1, argument2 ,..) => expression
```

Example

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

console.log(multiply(4, 3));

Output:

12

Function Parameters

While calling a function, we can pass different parameters. We have seen and read about the parameters above but haven't dived deeper into them. Parameters are additional information we pass to a function and can be manipulated. We pass parameters to a function within parentheses after the function name and use commas to separate them. There can be any number of parameters, but a function in JavaScript can't have a single parameter.

Example

```
function calculateSum(a, b) {
  return a + b;
}
let result = calculateSum(5, 10);
```

console.log("The sum is:", result);

Output:

The sum is: 15

Explanation:

Function Definition: The function `calculateSum` takes two parameters: a and b.

Parameters as Variables: a and b act as variables inside the function.

Return Statement: The function returns the sum of a and b.

Function Call: The function is called with arguments 5 and `10`, and the return value is stored in `result`.

Return Statement

We may want to return some values from a function after performing a few operations. Hence, we use the return statement, which is optional and mostly the last statement in JavaScript functions. For example, we can have two numbers in a function and want it to return the result after calculating them.

The return statement ends the execution of the function and mentions the value to be returned to the calling function. The code written after the return statement is unreachable and throws a warning.

Syntax

```
return value;
```

Here, the syntax begins with the keyword return followed by the value that will be returned. We can also use an expression instead of a value. If we omit the value, undefined is returned.

Example

```
function multiplyNumbers(a, b) {
  return a * b;
  console.log("This line will not execute because it comes after return");
}
let result = multiplyNumbers(4, 5);


console.log("The product is:", result);
```

Output:

The product is: 20

Function Hoisting

Hosting in JavaScript is the process that involves declarations of variables, functions, and classes moving to the top of the scope before code execution. With hoisting, we can safely use functions in the code before declaring them.

Example

```
sayHello();


function sayHello() {
  console.log('Hello, World!');
}
```

Output:

Hello, World!

Function Scope

Scope means the visibility of functions, variables, and objects within a part of the code during the runtime. Scope determines the visibility of variables and resources in a specific area of the code. Variables defined within a function can’t be accessed from anywhere

outside that function because they are defined in the scope of that function. However, a function can access all variables and functions defined within the scope.

Moreover, a function defined within another function can access variables defined in its parent function and other variables that the parent function can access. The scope of the function can be its body (which declares local variables), or it can contain the name of the function (a class or namespace).

Local Scope- A variable defined within a function is said to have a local scope. A function in JS can access a variable present within its scope or global scope but can't access a variable defined in another function.

Example

```
function localScopeExample() {
  let localVar = "I am a local variable";
  console.log(localVar);
}

localScopeExample();

console.log(localVar);
```

Output:

Error: localVar is not defined

Global Scope- A variable defined outside a function is said to have a global scope. The outermost scope in a script is the global scope.

Example

```
let globalVar = "I am a global variable";

function globalScopeExample() {
  console.log(globalVar);
}
globalScopeExample();
```

```
console.log(globalVar);
```

Output:

I am a global variable

I am a global variable

Lexical Scope- In nested functions, the innermost function can access the variables and resources of their parent scope.

Example

```
function outerFunction() {
  let outerVar = "I am from the outer function";
  function innerFunction() {
    console.log(outerVar);
  }

  innerFunction();
}
outerFunction();
```

Output:

I am from the outer function

Scope and the Function Stack

Recursion

Recursion is when a function can call or refer itself to solve a problem. We can also say that a function that calls itself is a recursive function. A function can refer to itself in three ways:

- Name of the function.
- In-scope variable referring to the function.
- arguments.callee- It contains the currently executing function.

Example

```
function countdown(n) {
  if (n <= 0) {
    console.log("Done!");
    return;
  }
  console.log(n);
  countdown(n - 1);
}


countdown(5);
```

Output:

```
5
4
3
2
1
Done!
```

Nested Functions and Closures

A function within another function is a nested function. The nested or inner function is private to its containing or outer function. The inner function can access variables or other resources of its parent function.

Example

```
function outerFunction() {
  let outerVar = "I am from the outer function";


  function innerFunction() {
    console.log(outerVar);
  }
```

```
  innerFunction();
}
outerFunction();
```

Output:

I am from the outer function

Closures

Closure is an expression with access to variables and parameters of the outer function. The inner function preserves the outer function value even when the outer function has returned. A closure has free variables with an environment that binds those variables.

A nested function is a closure, i.e., it can inherit the variables and arguments of its containing function.

Scope Chain- When we use a variable, the JavaScript engine will find the value of the variable in its current scope. If it can't find the variable, it will seek the outer scope and continue the process until it finds the variable or reaches the global scope. If it can't find the variable until the end, it will declare the variable implicitly or throw an error.

Key points to note:

- The inner function can be accessed from the outer function statements.
- It forms a closure, so the inner function can use the variables of the outer function.

Example

```
function outer() {
  let outerVar = "I am from the outer function";

  function inner() {
    console.log(outerVar);
  }

  return inner;
}
```

const closureFunction = outer();

closureFunction();

Output:

I am from the outer function

A closure looks like this:

- An outer function
- An inner function
- A return statement referencing the inner function

```
function outer(){
  function inner(){
 }
 return inner;
}
let demo = outer();
demo();
```

We can use a closure to place one function within another and execute the inner function later, preserving the variables of its parent function.

Example

```
function counter() {
  let count = 0;


  return function () {
    count++;
```

```
    console.log(`Count: ${count}`);
  };
}

const increment = counter();
increment();
increment();
increment();
```

Output:

Count: 1

Count: 2

Count: 3

JavaScript Anonymous Functions

JavaScript functions are first-class citizens, so we can treat them as values and pass them around like other variables. One primary concept related to this is anonymous functions.

Anonymous means something that is unknown, that doesn't have a name or identity. **JS anonymous functions** don't have any names. When we create these functions, they are declared without any identifier. Arrow functions are anonymous functions with a short syntax.

Normal functions have names associated with them, which is the main difference between them and anonymous functions.

In this chapter, we will explore **anonymous functions in JavaScript**, their role, how they work, their implementation, their uses, and more. Let's start by understanding what an anonymous function is.

What is an Anonymous Function in JavaScript?

In JavaScript, we declare a function by using the function keyword followed by its name. However, an **anonymous function in JavaScript** doesn't have a name or identity, so we can't call them like other functions in JS. These functions are created for specific tasks and

are used as arguments for other functions or assigned to variables. We use anonymous functions when we don't need to reuse the function outside its immediate context.

Syntax

Here is the syntax for declaring an anonymous function using the normal declaration:

```
function() {
    // Function Body
 }
```

We can also use the arrow function technique to declare an anonymous function, as shown below:

```
( () => {
    // Function Body...
} )();
```

Examples 1: Anonymous Function Assigned to a Variable

```
const greet = function() {
  console.log("Hello, this is an anonymous function!");
};

greet();
```

Output:

Hello, this is an anonymous function!

Example 2: Anonymous Function as a Callback

```
setTimeout(function() {
  console.log("This is an anonymous function executed after 2 seconds!");
}, 2000);
```

Output:

This is an anonymous function executed after 2 seconds!

Example 3: Anonymous Arrow Function

```
const sum = (a, b) => {
  return a + b;
};

console.log(sum(5, 3));
```

Output:

8

Implementation of JavaScript Anonymous Functions

Here is an example showing how to use an anonymous function in JS and how it is different from normal functions.

Examples 1: Anonymous Function Used as a Callback for Array's `forEach`

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

numbers.forEach(function(number) {
```

```
  console.log(number * 2);
});
```

Output:

2

4

6

8

10

Example 2: Anonymous Function with Arrow Function Syntax for Sorting

```
const fruits = ["banana", "apple", "orange", "mango"];
fruits.sort((a, b) => a.localeCompare(b));
console.log(fruits);
```

Output:

["apple", "banana", "mango", "orange"]

Uses of Anonymous Functions in JavaScript

We can use **anonymous functions in JS** for multiple purposes. Some of them are:

Using Anonymous Functions as Arguments in Other Functions

As functions can be passed as an argument to other functions or can be returned by other functions, they are called first-class citizens. One of the best uses of anonymous functions is they can be passed as arguments to other functions.

Example

```
setTimeout(function() {
  console.log("This message is displayed after 2 seconds.");
}, 2000);
```

Output:

This message is displayed after 2 seconds.

Invoke Function Execution

When we need to execute a function immediately after declaring it to avoid going through the hassle of calling it, we use anonymous functions as they execute right after when declared.

In JS, an IIFE (Immediately Invoked Function Expression) is a function that is invoked immediately. We can also say that it runs as soon as it is defined. It allows us to execute functions right after they are created.

Syntax

```
(function () {
  statements
})();
```

The design pattern of IIFE is also called Self-executing Anonymous Function, containing the following major parts as shown in the image below:

Example

```
(function() {
  console.log("This anonymous function is invoked immediately!");
```

```
})();
```

Output:

This anonymous function is invoked immediately!

Arrow functions

With ES6, JavaScript introduced a shorter way of declaring an anonymous function. It is called the arrow function. They are compact and allow us to write cleaner and shorter code. In an arrow function, everything is the same except that it doesn't require the function keyword. We define the function through a single parenthesis, followed by =>and the function body.

However, arrow functions have a few limitations, making them unsuitable in some cases.

Arrow Functions Syntax

In case of one parameter

```
param => expression
```

Here, we write the parameter name without using parentheses. In the case of a simple expression, we can skip the return keyword and curly braces.

In case of multiple parameters

```
(param1, params) => expression
```

We write parameters and the code statement. In simple expressions, we can omit the return keyword.

In case of 0 parameters

```
() => expression
```

In case of multiple parameters and multi-line code statement

```
(param1, paramN) => {
    //line 1
    //line 2
    return value
}
```

Here, we write parameters enclosed in (), followed by =>. The code statements are within curly braces {}. Similar to other JavaScript functions, we can optionally return the result. For multi-line code statements where you need to return a value from the function, you must use the return keyword but can skip it if there is a single-line code statement.

Example

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

console.log(greet());

Output:

Hello, World!

JavaScript Function Expression

The function expression is used to define a JavaScript function in an expression. A JS function is defined using a function expression or function declaration, but we can omit the function name in the function expression. Hence, we can create an anonymous function in

JavaScript. We can store **JavaScript function expressions** in a variable, which can be used to invoke the function expression.

What is Function Expression in JavaScript?

The **JS function expression** allows us to define a function within an expression. Also, we can use it to create an anonymous function, i.e., a function without a name, which differentiates it from a function declaration.

We can also use a function expression as an IIFE (Immediately Invoked Function Expression), which executes immediately after it is defined. We store it in a variable and can access it using variableName. As ES6 has introduced the Arrow Function, we can easily declare function expression.

Syntax for Function Declaration

```
function functionName(x, y) { statements... return (z) };
```

Syntax for Anonymous Function Expression

```
let variableName = function(x, y) { statements... return (z) };
```

Syntax for Named Function Expression

```
let variableName = function functionName(x, y)
{ statements... return (z) };
```

We can’t use the function name to invoke the function; we must use only the variable.

Syntax for Arrow Function

```
let variableName = (x, y) => { statements... return (z) };
```

We must define a function expression before calling or using it as a parameter. Also, an arrow function must have a return statement.

Examples

```
// Anonymous Function Expression
let add = function(x, y) {
    return x + y;
};
console.log(add(5, 3));

// Named Function Expression
let subtract = function subtractNumbers(x, y) {
    return x - y;
};
console.log(subtract(10, 4));

// Arrow Function
let multiply = (x, y) => x * y;
console.log(multiply(4, 6));
```

Output:
8
6
24

Example - (Function Declaration)

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

console.log(greet('Harun'));
```

Output

Hello, Harun!

Example - Function Expression (anonymous)

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

console.log(add(5, 3));
```

Output

8

Example - Function Expression (named)

```
const multiply = function multiplyNumbers(a, b) {
   return a * b;
};

console.log(multiply(4, 2));
```

Output

8

Example - Arrow Function

```
const divide = (a, b) => a / b;
```

```
console.log(divide(10, 2));
```

Output

5

JavaScript Hoisting

Hoisting in JavaScript is the behavior where the interpreter moves declarations of variables, functions, and classes to the top of their scope before code execution. Hence, variables can be accessed before the declaration, which promotes flexible coding practices and mitigates the chances of undefined errors.

What is Hoisting in JavaScript?

JavaScript hoisting plays a crucial part in how functions and variables are executed within their respective scopes. It is the default behavior where function and variable declarations are moved to the top of their respective scopes during the compilation phase. This allows us to use variables, functions, and classes before they are declared. Also, irrespective of where the declarations appear, they can be accessed throughout the scope.

Due to hoisting, developers can call functions and reference variables before they are defined, making the code more flexible.

Features of Hoisting in JavaScript

- Allows developers to call functions before they are declared.
- Declarations are hoisted and not initializations.
- When assigned a value, undeclared variables are created implicitly as global variables.
- Variable and function declarations are done before code execution.
- In JavaScript, memory is allocated for all variables before execution.

Sequence of Variable Declaration

Here is the sequence of declaring and initializing a variable:

Declaration –> Initialisation/Assignment –> Usage

Variable Lifecycle

```
let a;              // Declaration
a = 246;            // Assignment
console.log(a);  // Usage
```

In JavaScript, we can declare and initialize variables simultaneously.

```
let a = 246;
```

Please Note- In the background, JavaScript first declares the variable and then initializes it. So, you must know that the variable declaration occurs before code execution. However, undeclared variables in JavaScript don't exist until the code assigning them is executed. Hence, when we assign a value to an undeclared variable, it implicitly creates the variable as a global variable. Therefore, all undeclared variables are global variables.

JavaScript Function Hoisting

Function declarations are placed in the memory at the compile time. Hence, it allows a declared function to be called before it is defined. We use a function in the code before declaring it officially, which makes the sequence of function declarations less restrictive.

Example

```
greet();


function greet() {
   console.log("Hello, World!");
```

}

Output:

Hello, World!

JavaScript Variable Hoisting

We have been discussing **Variable hoisting in JavaScript** throughout the chapter. It is the process where the variable declaration is moved to the top of the code. The hoisting behavior varies based on whether a variable is declared using let, var, or const.

Hoisting with 'var'

Hoisting with var is a bit different. When we use the var keyword to declare a variable, the interpreter hoists its declarations.

Example

console.log(x);

var x = 5;

console.log(x);

Output:

Undefined

5

JavaScript Hoisting with 'let' and 'const'

When a variable is declared with the let or const keyword, it is hoisted to the top of its current scope. However, it doesn’t have a default value when hoisted. A variable declared with let is block-scoped and not function-scoped, so there are no issues in hoisting.

Example

console.log(b);

let b = 20;

console.log(b);

Output:

ReferenceError: Cannot access 'b' before initialization

20

When hoisted with let, the assignment occurs after initialization, but with const, the assignment and initialization occur simultaneously because we can't change the value once assigned. If we assign an undefined value and try to assign another value, it will violate the rules of hoisting.

Example

```
console.log(c);

const c = 30;

console.log(c);
```

Output:

ReferenceError: Cannot access 'c' before initialization

30

Initializations Are Not Hoisted

Although JavaScript moves declarations of variables at the top of their respective scopes before code execution, the initialization part remains at its original place in the code.

Example

```
console.log(a);

var a = 50;

console.log(a);
```

Output:

undefined

50

To Hoist or Not to Hoist?

ECMAScript specification, which explains the standards of JavaScript language, doesn't explicitly define hoisting. It is more of a JS behavior than a feature. Therefore, you must focus on its characteristics while working with JavaScript rather than using it as a tool or design pattern in the apps you write. In a few code bases, you may find functions used in business logic at the top of the script and its declarations at the bottom. This is all due to hoisting.

JavaScript Arrow Function

With ES6, JavaScript introduced the Arrow function, which offers a more readable and concise way to write function expressions. They have a simple syntax containing => and reducing boilerplate. An **arrow function in JavaScript** makes the code more structured and is quite useful when we work within objects or need to handle callbacks.

We will explore the JavaScript arrow function more, including its syntax, benefits, drawbacks, examples, and more.

What is an Arrow Function in JavaScript?

A **JavaScript arrow function** is an anonymous function, i.e., a function without a name and not bound by an identifier. It has a more concise syntax and is assigned to variables, which makes it reusable. Arrow functions don't return any value and can be declared without using the function keyword. They are also known as lambda functions in other programming languages.

We can't use arrow functions as constructors, and the context within them is statically or lexically defined. They don't include prototype property and can't be used with the new keyword.

JavaScript Arrow Function Syntax

```
let myFunction = (arg1, arg2, ...argN) => {
    statement(s)
}
```

myFunction is the name of the function.

=> is the lambda notation.

Arg1, arg2,...argN are the arguments.

statement(s) is the body of the function.

In case the body has a single expression or statement, we can also write the arrow function using the following syntax:

```
let myFunction = (arg1, arg2, ...argN) => expression
```

Things to Avoid with Arrow Functions

- We can't use the **JS arrow function** as a constructor.
- We must not use it to create methods within objects.

JavaScript Arrow Function Examples

1. Simple JavaScript Function to Print a Message

Example

```
const myFunction = () => {
        console.log('Welcome to WsCube Tech')
};

myFunction();
```

Output:
Welcome to WsCube Tech

2. Arrow Function with a Single Parameter

Example

```
const square = x => x * x;

console.log(square(5));
```

Output:

25

You can omit the parentheses if the arrow function has a single parameter.

3. Arrow Function with Multiple Parameters

Example

```
const add = (a, b) => a + b;

console.log(add(3, 7));
```

Output:

10

If the arrow function has multiple parameters, it will simplify writing concise function expressions, making it useful for functions that require multiple arguments.

4. Arrow Function without Parameters

Example

```
const greet = () => 'Hello, World!';

console.log(greet());
```

Output:

Hello, World!

If the arrow function doesn't take any argument, we can define it using empty parentheses. We use it for functions that don't need arguments.

5. Arrow Function with Default Parameters

Example

```
const greetUser = (name = 'Guest') => `Hello, ${name}!`;


console.log(greetUser());

console.log(greetUser(Aman));
```

Output:

Hello, Guest!

Hello, Aman!

JavaScript arrow functions support default parameters, which allow predefined values when no argument is passed. This makes defining the function more concise and flexible.

6. Return Object Literals

Example

```
const getUser = (name, age) => ({ name, age });


console.log(getUser('Krishna', 25));
```

Output:

{ name: 'Krishna', age: 25 }

Returning object literals within a function is concise. Using ()=> ({ key: value }) will return an object {key: value}, which will immediately create and return an object.

Async Arrow Functions

We can make arrow functions asynchronous by using the sync keyword before the list of parameters.

Example

```
const fetchData = async () => {
const response = await fetch('Api url');
```

```
    const data = await response.json();
    console.log(data);
};

fetchData();
```

this Keyword with Arrow Function

We use the this keyword inside a regular function, referring to the function where it is called. However, we don't use it in the arrow function. If we call this, it refers to the parent scope.

Example

```
const person = {
    name: 'Harun',
    greet: function() {
        setTimeout(() => {
            console.log(`Hello, my name is ${this.name}`);
        }, 1000);
    }
};

person.greet();
```

Output:
Hello, my name is Harun

Arrow Function as an Expression

After creating an arrow function, we can also use it as an expression.

Example

```
const isEven = num => num % 2 === 0;

console.log(isEven(4));
console.log(isEven(7));
```

Output:
true
false

Multiline Arrow Functions

If there are multiple statements in a function body, we write them within curly braces {}.

Example

```
const multiplyAndLog = (a, b) => {
   const result = a * b;
   console.log(`Result: ${result}`);
   return result;
};

console.log(multiplyAndLog(3, 4));
```

Output:
Result: 12
12

They are Better for Short Functions

We can omit the curly braces in the arrow function to create a single-line function. Also, in the case of a single parameter, we can remove parentheses containing the function parameters. We use empty parentheses between the assignment and the arrow function syntax when there are no parameters. Also, they are better options when we don't need to name the function.

Example

```
// Regular Function Callback

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

const doubled = numbers.map(function(number) {

   return number * 2;

});


// Arrow Function Callback
```

```
const doubledArrow = numbers.map(number => number * 2);


console.log(doubled);

console.log(doubledArrow);
```

Output:

[2, 4, 6, 8, 10]

[2, 4, 6, 8, 10]

JavaScript Recursion

This chapter will focus on **recursion in JS**, exploring its fundamentals, implementation, and more through examples.

So, let's get started by understanding the meaning of JavaScript recursion with examples.

What is Recursion in JS?

JS recursion function is a programming technique where a function calls itself repeatedly within the function body to solve a problem. The recursion function executes until it reaches the Base Case. This powerful technique is used for problems whose solutions depend on smaller versions of the same problem.

Key Points to Remember

- In simple words, the recursion function calls itself again and again. It's similar to running on a racing track repeatedly, and with every time, the lap gets smaller.
- Recursion is a function that calls itself repeatedly and stops on its own. However, you must know that a function doesn't stop itself. To make the function stop, you must provide a Base Case, which tells the function it's time to stop.
- Once you give the Base Case condition, implement the recursion part in which you call the function according to the required result.
- Recursion is also like a loop in which each user-passed parameter function is called again and again, producing the output as per the requirement.

- **JS recursion examples** include Palindrome Check, Longest Common Subsequence, etc.

Syntax for JavaScript Recursion

```
function recursion() {
    if (condition) {       //desired command
        return;            // return from the current loop
    }
    // Function code
    // recursive step codes
}
recursion();
// This is the simplest syntax format a recursive function can take
```

Example

```
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
}


console.log(factorial(5));
```

Output:

120

Three Parts of a JavaScript Recursive Function

The **JavaScript Recursion** works the same way as other programming languages. Syntax includes three crucial elements when writing the recursive function(). If any of the three elements are missing, the recursive function will not work as expected. We have explained below:

1. The Function Declaration
2. The Base Case
3. The Recursive Call command

The Function Declaration

This is the part where the **JavaScript recursion function** is defined or declared. It is similar to how a new function is declared in JavaScript. However, the difference between a recursive function and a regular function is the base condition and recursive call.

Syntax

```
function recursion() {     //The declared function takes arguments

    // Function and recursive step codes

}
```

Example

```
function sumOfNaturalNumbers(n) {

   if (n === 0) {

      return 0;

   }

   return n + sumOfNaturalNumbers(n - 1);

}


console.log(sumOfNaturalNumbers(5));
```

Output:

15

Explanation of the Three Parts:

- **The Function Declaration:**

function sumOfNaturalNumbers(n) { ... }

- **The Base Case:**

```
if (n === 0) {
  return 0;
}
```

- **The Recursive Call:**

return n + sumOfNaturalNumbers(n - 1);

The Base Condition

While using a **JavaScript recursive function**, the base case allows the function to know when to stop calling itself. Once the base condition is met, the recursion terminates. It is another critical component of the recursive function. When we divide a big issue into smaller chunks, it is the smallest aspect of a problem to be solved. It is similar to the command because if the set condition is true, it can stop the recursion process.

Syntax

```
function recursiveFunc() {
  if(base condition) {
    // stops recursion if condition is met
  }
  // else recursion continues
  recurse();
}
```

Example

function factorial(n) {

// Base Case: If n is 0 or 1, return 1 to stop recursion

```
    if (n <= 1) {
        return 1;
    }

    // Recursive Call: Multiply n by the factorial of (n-1)
    return n * factorial(n - 1);
}
console.log(factorial(5));
```

Output:

120

Explanation:

- **Base Condition:**

  ```
  if (n <= 1) {
      return 1;
  }
  ```

- **Recursive Call:**

```
return n * factorial(n - 1);
```

Why do we need a base case command?

If there is no base condition, you will encounter an infinite recursion, where the function will continue calling itself without stopping. Moreover, without the base command, the function will exceed the maximum call stack, and you will run into an error.

The Recursive Call

The recursive call handles the function calling itself repeatedly. It is responsible for triggering a new recursive call. The recursive call command is the actual command addressing the main issue to be solved.

```
doSomething(n-1)
```

When a function calls itself, a new parameter (n - 1) is passed to the function. With each iteration of a recursive call, the parameter differs from the previous call. The function will call itself until the new parameter meets the base condition.

```
function recursion() {    // The Function Declaration
    if (true) {           // The base case
        return;
    }
    // Function code
    // and recursive step codes
    return ... ;
}
//The Recursive Call command
recursion()
```

Example

```
function countdown(n) {
    // Base Case: If n reaches 0, stop the recursion
    if (n <= 0) {
        console.log("Done!");
        return;
    }

    // Recursive Call: Print the current number and call countdown(n-1)
    console.log(n);
    countdown(n - 1);
}

countdown(5);
```

Output:

5,

4,

3,

2,

1,

Done!

Using the if...else Statement to Stop Infinite Recursion

You can prevent infinite recursion by incorporating the if…else statement within the recursive function to establish the base case where the function stops calling itself repeatedly. The base case condition halts the recursive calls to terminate the recursion.

Example

```
function factorial(n) {
   // Base Case: When n is 1 or less, stop recursion
   if (n <= 1) {
      return 1;
   } else {
      // Recursive Call: Multiply n by factorial of (n - 1)
      return n * factorial(n - 1);
   }
}

console.log(factorial(5));
```

Output:

120

Recursion vs Loops

Recursion and loops are control flow mechanisms that work similarly in JavaScript. For each recursive function you write, there is an alternative solution with a loop. However, the two have a few distinct features.

JavaScript recursion includes a function calling itself again and again until it reaches the base condition. This elegant method is used to solve problems where solutions depend on small parts of the same issue.

On the other hand, loops iterate over a code block until a specific condition is met, which provides a straightforward approach for repetitive tasks.

Recursion is concise and expressive for a particular problem, whereas loops are efficient for performance and memory usage. So, choose the right option depending on your requirements and the constraints of the issue.

Examples -1: How to find the factorial with a loop

```
function factorialLoop(n) {
   let result = 1;
   for (let i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}

console.log(factorialLoop(5));
```

Output:

120

Examples - 2: How to find the factorial with recursion

```
function factorialRecursion(n) {
```

```
    if (n <= 1) {
        return 1;
    } else {
        return n * factorialRecursion(n - 1);
    }
}

console.log(factorialRecursion(5));
```

Output:

120

JavaScript CallBack Function

JavaScript can handle asynchronous programming, allowing programmers to do multiple tasks at once without blocking the main execution thread. One of the crucial aspects to achieving this is **callback functions in JavaScript,** which help developers execute specific code once an asynchronous task is completed.

Let's learn more about the **JS callback function,** why use it, and some real-world examples.

What is a Callback in JavaScript?

A callback in JavaScript is a function passed as an argument to another function. It is executed once the main function is completed. It is called at the end of a task to deliver results or perform an action. It allows JS to perform various tasks, such as making HTTP requests, reading files, waiting for user input without blocking program execution, etc., enhancing user experience.

We pass the **JavaScript callback** to the main function, and once the main function is completed, it invokes the callback to move to the next steps.

Why use the JavaScript Callback Function?

As JavaScript runs in a single-threaded environment, it can execute only one command at a time. **JS callbacks** help manage the outcomes of asynchronous operations, which take time to complete. In the case of synchronous tasks, the program halts until they are finished, which would make the user experience sluggish.

However, callbacks ensure that the code runs smoothly without waiting for the tasks to complete. You can continue running the program while the tasks are running in the background. Once the task is done, the callback function handles the result. It is crucial for responsive programs and improved user experience.

Key Points to Remember About Callbacks

Anonymous Functions

These functions are not named and are used as callbacks.

Asynchronous Programming

We use callback in JavaScript to handle the output of asynchronous tasks without blocking the execution of the remaining program. The program continues to run smoothly, and the callback function is executed once the operation is finished.

Higher-order Functions

They take one or more functions as arguments or return a function as the output.

Closure

This function has access to variables in the outer scope even when the outer function has returned. Hence, the callback function can access variables and information from the main function even after the main function completes the execution.

Non-blocking

In non-blocking programming the program doesn't stop but waits for the operation to complete before continuing the execution. This improves the performance and responsiveness of applications.

Real-Life Examples of JS Callback

To handle the form submission

During form submission, data processing takes time and so does the process of sending it to the server. For the synchronous form submission process, a user has to wait for the data processing and send it to the server before submitting the form. However, callbacks enable the user to handle the form submission task asynchronously. Hence, they can keep on interacting with the form while the data is processed and sent in the background.

To load images on a website

When you open a website, loading images can be time-consuming, especially for large-sized images. When images are loaded synchronously, the website freezes and loads every image and then continues. JS callbacks allow you to load images asynchronously so the website can run smoothly and load images in the background.

Example: Basic Callback Function

```
function submitForm(data, callback) {
    console.log("Form submission started...");
    setTimeout(() => {
        console.log("Form data processed:", data);
        callback("Form submitted successfully!");
    }, 2000);
}

function handleSubmit() {
    const formData = { name: "John Doe", email: "john@example.com" };
    submitForm(formData, (message) => {
        console.log(message);
    });
}

handleSubmit();
```

Output:

```
Form submission started...
Form data processed: { name: "John Doe", email: "john@example.com" }
Form submitted successfully!
```

Example: Callback with Array.forEach() Method

```
function loadImage(url, callback) {
    console.log(`Loading image from ${url}...`);
```

```
    setTimeout(() => {
        console.log(`Image loaded: ${url}`);
        callback();
    }, 1500);
}

function displayImages() {
    const imageUrls = [
        "image1.jpg",
        "image2.jpg",
        "image3.jpg"
    ];

    imageUrls.forEach((url) => {
        loadImage(url, () => {
            console.log(`Callback executed for ${url}`);
        });
    });
}

displayImages();
```

Output:

```
Loading image from image1.jpg...
Loading image from image2.jpg...
Loading image from image3.jpg...
Image loaded: image1.jpg
Callback executed for image1.jpg
```

Image loaded: image2.jpg

Callback executed for image2.jpg

Image loaded: image3.jpg

Callback executed for image3.jpg

JavaScript Objects

Objects in JavaScript refer to a data type that builds the foundation of modern JavaScript. It is composed of a collection of names or key-value pairs. Primitive data types, like boolean, number, symbol, string, null, or undefined, hold a single value, but objects can hold multiple values. Hence, they are used to group related data and operations, which makes the code more maintainable and organized.

This chapter will dive deeper into **JS objects**, exploring different operations performed on objects through examples.

What are JavaScript Objects?

In JavaScript, objects refer to a collection of related data and functions, which are known as properties and methods, respectively. Properties are composed of names or keys and values, represented by key: value pairs that store data, whereas methods are functions associated with the object, manipulating its properties.

JavaScript Objects Syntax

```
new Object(value)

Object(value)

let object_name = {

    key1: value1,

     key2: value2,

    ...,

    keyN: valueN
```

```
}
```

key1: value1- First key-value pair

key2: value2- Second key-value pair

keyN: valueN- Nth key-value pair

Please Note- Every key-value pair has a colon between them and is separated by a comma. We can call an object() with or without new; both methods will create a new object.

How to Create JavaScript Objects?

There are three ways to create objects in JS, which are:

1. Using object literal
2. Using constructor function
3. Using Object.create()

Using object literal

You can create an object using **JavaScript object literals**, which involves defining an object with key-value pairs inside curly braces {}. It is a simple and concise method that allows you to create objects in no time with method and properties while improving code readability.

Example

```
let person = {
   name: "Aman",
   age: 25,
   greet: function() {
      console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
   }
};


console.log(person.name);
console.log(person.age);
person.greet();
```

Output:

Aman

25

Hello, my name is Aman and I am 25 years old.

Using constructor function

This is the easiest method to create an object in JavaScript that involves a constructor (a function) and the new keyword. With the constructor function, we can create multiple objects.

Example

```
function Car(make, model, year) {
   this.make = make;
   this.model = model;
   this.year = year;
   this.getDetails = function() {
      return `${this.year} ${this.make} ${this.model}`;
   };
}

let car1 = new Car('Toyota', 'Camry', 2022);
let car2 = new Car('Honda', 'Civic', 2023);

console.log(car1.getDetails());
console.log(car2.getDetails());
```

Output:

2022 Toyota Camry

2023 Honda Civic

Using Object.create()

This method uses the existing object as the prototype to create new objects in JavaScript. In this method, the new object inherits the properties and methods of the prototype object, which enables inheritance-like behavior. We use Object.create() to create objects with shared behavior with unique properties.

Example

```
let animal = {
   type: "Mammal",
   sound: function() {
      console.log(`${this.name} makes a sound.`);
   }
};

let dog = Object.create(animal);
dog.name = "Dog";
dog.sound();
```

Output:

Dog makes a sound.

JavaScript Object Example

```
let person = {
   name: "Arun",
   age: 30,
   isStudent: false
};

console.log(person.name);
console.log(person.age);
console.log(person.isStudent);
```

Output:
Arun
30
false

Objects as Reference Types

JavaScript objects are reference types, so when we assign them to variables, the variables hold a pointer or reference to the location where the object is stored rather than the actual object.

JavaScript Object Properties

Properties are key-value pairs of an object. The key is a symbol or a string, and the value can be any data type, including functions, strings, arrays, booleans, numbers, or other objects.

Example

```
let user = {
    name: "Arun",
    age: 25,
    isAdmin: true,
    hobbies: ["Reading", "Gaming"],
    address: {
        city: "New Delhi",
        zip: 10001
    },
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};


console.log(user.name);
console.log(user.age);
console.log(user.isAdmin);
```

```
console.log(user.hobbies[0]);
console.log(user.address.city);
user.greet();
```

Output:

Arun

25

True

Reading

New Delhi

Hello, my name is Arun

Inherited Properties

Objects in JS can acquire properties of their prototypes, known as inherited properties. We use the hasOwnProperty method to check if an object has its own properties.

Example

```
let animal = {
   eats: true,
   walk: function() {
      console.log("Animal walks");
   }
};
let dog = Object.create(animal);
dog.bark = function() {
   console.log("Dog barks");
};
```

```
console.log(dog.eats);

dog.walk();


console.log(dog.hasOwnProperty('eats'));

console.log(dog.hasOwnProperty('bark'));
```

Output:

true

Animal walks

false

true

Property Attributes

Every property in JavaScript has attributes that represent its behavior.

writable- We can change the value of the property if it is true.

configurable- If false, any attempt to change the property to be an access or property, delete the property, or change its attribute will fail.

value- It refers to the value of the property

enumerable- If true, we can iterate the property over by for-in enumeration; else, the property is non-enumerable.

Example

```
let user = {};


Object.defineProperty(user, 'name', {

   value: 'Arun',

   writable: false,

   enumerable: true,

   configurable: false

});
```

```
console.log(user.name);

user.name = 'Bob';
console.log(user.name);
delete user.name;
console.log(user.name);

for (let key in user) {
   console.log(key);
}


Object.defineProperty(user, 'name', {
   value: 'Charlie'
});
```

Output:

Arun

Arun

Arun

name

TypeError: Cannot redefine property: name

Accessing Object Members

There are two ways to access object members (properties or methods):

1. Dot notation

Syntax

```
(objectName.memberName);
```

2. Bracket notation

   Syntax

```
objectName["memberName"]
```

Example

```
let user = {
   name: "Arun",
   age: 25,
   "favorite hobby": "Reading"
};

console.log(user.name);
console.log(user.age);
console.log(user["name"]);
console.log(user["age"]);
console.log(user["favorite hobby"]);

user.city = "New Delhi";
user["country"] = "India";

console.log(user.city);
```

```
console.log(user["country"]);
```

Output:

Arun

25

Arun

25

Reading

New Delhi

India

The dot notation and bracket notation methods are used regularly, but the former is faster and more readable. However, it also has a few limitations. The bracket notation method allows access to property names stored in a variable and is used when the property of an object contains a special character.

JavaScript Object Operations

JavaScript allows us to perform various operations on object properties, such as modifying, adding, and deleting. Let's take a closer look at these operations.

Modifying Object Properties

Modifying object properties involves assigning a new value to the existing key. We can use dot notation to modify properties in an object.

Example

```
let user = {
   name: "Arun",
   age: 25,
   city: "New Delhi"
};
console.log(user.name);
console.log(user.city);
```

```
user.name = "Varun";

user.city = "Lucknow";

console.log(user.name);

console.log(user.city);
```

Output:

Arun

New Delhi

Varun

Lucknow

Adding Object Properties

To add a new object property to an object, we assign a new value to the property and use the assignment operator. We can also create a brand new object or use the dot notation when we want to access properties that the object has not added before.

Example

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

user.city = "New Delhi";

user.country = "India";

user["hobby"] = "Reading";

user["isStudent"] = false;

let dynamicKey = "email";
```

```
user[dynamicKey] = "Arun@example.com";


console.log(user);
```

Output:

```
{
   name: "Arun",
   age: 25,
   city: "New Delhi",
   country: "India",
   hobby: "Reading",
   isStudent: false,
   email: "Arun@example.com"
}
```

Deleting Object Properties

We use the delete operator to delete or remove a property of an object. It is a simple process to discard the properties you want. Please note that you can’t delete inherited or non-configurable properties.

Example

```
let user = {
   name: "Arun",
   age: 25,
   city: "New Delhi"
};
delete user.city;


console.log(user);
```

Output:

{ name: "Arun", age: 25 }

Iterating Over All Keys of an Object

In JavaScript, there is a built-in type of the for loop specifically designed to iterate over all enumerable properties of an object. It is called the for…in loop that iterates over the own properties of an object, along with enumerable properties that it has inherited.

Example

```
let user = {
   name: "Arun",
   age: 25,
   city: "New Delhi"
};
for (let key in user) {
   console.log(`${key}: ${user[key]}`);
}
```

Output:

name: Arun

age: 25

city: New Delhi

Dynamically Set Properties

JavaScript lets you enable a dynamic feature by simply using square brackets and objects. This method is used when we want to define a new property, especially when the property name is not known.

Example

```
let user = {};
```

let dynamicKey = "email";

user[dynamicKey] = "Arun@example.com";

console.log(user);

Output:

{ email: "alice@example.com" }

JavaScript Object Methods

Methods refer to a function, which is the value of an object property. Hence, it is a task that the object performs. We can access object methods in JavaScript using functions that are stored as property values. We can call objects without using ().

We can say that a property is a noun, whereas a method is a verb. For example, a name is a noun that can be associated with an object and is a property. Similarly, talk() is a verb that can be used as a method function definition.

In object methods, this represents the owner object. We can also include additional information with the object method.

Syntax

```
objectName.methodName()
```

We can divide a function into different property values that can be combined and returned together. It returns functions or methods stored as object properties.

Example- Storing property values and accessing without using ()

```
let person = {
    name: "Arun",
    greet: function() {
        return "Hello, " + this.name;
    }
};
```

```
console.log(person.greet());
```

Output:

Hello, Arun

Example- Using function definition as the property value

```
let person = {
    name: "Varun",
    introduce: function() {
        return `Hi, my name is ${this.name}.`;
    }
};
console.log(person.introduce());
```

Output:

Hi, my name is Varun.

Example- Using function definition as a property value and including additional details.

```
let person = {
    name: "Charlie",
    age: 30,
    details: function(greeting) {
        return `${greeting}, my name is ${this.name} and I am ${this.age} years old.`;
    }
};
console.log(person.details("Hello"));
```

Output:

Hello, my name is Charlie and I am 30 years old.

JavaScript Constructor Function

In JavaScript programming, creating and initializing objects is one of the basic tasks. We can achieve this through either constructor functions or class constructors. Both these methods serve the same purpose but adopt different approaches.

This chapter discusses the **constructor function in JavaScript** in detail through examples.

What is a Constructor Function in JavaScript?

The **JavaScript constructor function** is a special function used to create and instantiate a new function object. You can use it especially when you need to create multiple objects of the same kind.

When used with the new operator, it allows developers to define functions in no time and passes the function body as a string. This method offers greater flexibility in situations that require functions to be created based on runtime conditions or user input. As the **JS constructor** has a global scope, we can access them from anywhere in the code.

JavaScript Constructor Function Syntax

You can create a constructor in JavaScript the same way you create a regular function. Add the function keyword followed by arguments.

```
var variable = new Function(arg1, arg2..., Function Body)
```

Parameters

arg*- An optional parameter. You can add any number of string arguments.

Function Body- An argument containing the JS statements with the function definition.

Example

```
function Person(name, age) {

  this.name = name;
```

```
  this.age = age;


  this.greet = function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  };
}


const person = new Person('Arun', 25);


person.greet();
```

Output:

Hello, my name is Arun and I am 25 years old.

Types of Constructors in JavaScript

There are two types of constructors in JavaScript:

Built-in Constructors

These are readily available and predefined constructors bundled with the execution environment. They help create instances of native objects. Users simply need to invoke them to complete their tasks. JavaScript provides various built-in constructors, including Date(), Array(), and Object().

User-defined Constructors

They are custom constructors defined by a programmer to be used throughout the applications. You can use them to define your own object types with specific methods and properties. By convention, all constructors in JS are sentence-cased, so users can know that this function can be invoked using the new keyword.

What Happens When a Constructor is Called in JavaScript?

To use a constructor, we must call it. So, let’s see what happens in the process.

- When we use the new keyword, an empty object is created.

- this refers to the newly created object, becoming the current instance.
- The constructor is called, within which the properties and values are bound to the new object by using this.arg = arg.
- The object is returned.

Create Multiple Objects Using JS Constructor Function

We use object literals to create a single object. However, constructor functions are used to create multiple objects.

Example

```
function Product(name, price, category) {
  this.name = name;
  this.price = price;
  this.category = category;
  this.displayDetails = function() {
        console.log(`Product: ${this.name}, Price: $${this.price}, Category: ${this.category}`);
  };
}

const product1 = new Product('Laptop', 1200, 'Electronics');
const product2 = new Product('Shoes', 80, 'Fashion');
const product3 = new Product('Watch', 150, 'Accessories');
```

```
product1.displayDetails();

product2.displayDetails();

product3.displayDetails();
```

Output:

Product: Laptop, Price: $1200, Category: Electronics

Product: Shoes, Price: $80, Category: Fashion

Product: Watch, Price: $150, Category: Accessories

Each Object Created from a Constructor function is Unique

The objects we create using constructor functions are unique. Therefore, you can add a new property to a specific object that is not accessible to other objects. However, when you create an object using an object literal, any object variable made from that particular object will be like a clone of the original one. Hence, when you make changes in one object, they will reflect in others.

Example :

```
function Car(brand, model) {

  this.brand = brand;

  this.model = model;


  this.displayDetails = function() {

    console.log(`Car: ${this.brand} ${this.model}`);

  };

}


const car1 = new Car('Toyota', 'Corolla');

const car2 = new Car('Honda', 'Civic');
```

```
car1.color = 'Red';

car1.displayDetails();
console.log(car1.color);

car2.displayDetails();
console.log(car2.color);
```

Output:

Car: Toyota Corolla

Red

Car: Honda Civic

Undefined

Explanation:

1. **Constructor Function (`Car`)**
   - **`Car` constructor function initializes each object with `brand` and `model` properties.**
2. **Unique Objects:**
   - **`car1` and `car2` are separate instances.**
   - **Adding a `color` property to `car1` does not affect `car2`.**
3. **Property Access:**
   - **`car1` has a `color` property because it was explicitly added.**
   - **`car2` does not have a `color` property.**

JavaScript Built-in Constructors

In JavaScript, there are various built-in constructors used to create different types of objects. Some of them are:

Constructor	Description

Object()	Creates a new object with properties and methods.
Number()	Creates a new number object to handle data and operations.
String()	Creates a new string object to manipulate and represent textual data.
Array()	Creates a new array object.
Boolean()	Creates a new boolean object that represents true or false values for logical operations.

Example- 1: Using Object Constructor

```
const obj = new Object();
obj.name = 'Alice';
obj.age = 25;
console.log(obj);
```

Output:

```
{ name: 'Alice', age: 25 }
```

Example- 2: Using Number Constructor

```
const num = new Number(42);
console.log(num);
console.log(typeof num);
```

Output:

```
[Number: 42]
object
```

Example- 3: Using String Constructor

```
const str = new String('Hello, World!');
console.log(str);
console.log(typeof str);
```

Output:

```
[String: 'Hello, World!']
object
```

Example- 4: Using Array Constructor

```
const arr = new Array(1, 2, 3, 4, 5);
console.log(arr);
console.log(Array.isArray(arr));
```

Output:

```
[1, 2, 3, 4, 5]
true
```

Example- 5: Using Boolean Constructor

```
const bool = new Boolean(true);
console.log(bool);
console.log(typeof bool);
```

Output:

```
[Boolean: true]
object
```

Browser Compatibility

Although constructors are supported in almost all modern browsers, the compatibility of a few features might differ. Here are a few issues you may face:

Classic Static Methods- Certain old browsers may not support static methods in classes. Static methods are called on the class itself instead of instances of the class.

Arrow Functions- They were introduced in ES6 and showcase different behavior regarding the this keyword. In constructor functions, they may not behave as expected, especially when it's about handling the this context.

Class Syntax- A few old browsers may not completely support the features introduced in ES6. However, these features are compatible with modern browsers.

Example

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, I'm a ${this.name}`);
  }



  static info() {
    console.log('Animals are living organisms.');
  }
}



const animal = new Animal('Dog');
animal.greet();


Animal.info();
```

```
function Person(name) {
  this.name = name;
  this.sayHello = () => {
    console.log(`Hello, my name is ${this.name}`);
  };
}

const person = new Person('Arun');
person.sayHello();
```

Output:

Hello, I'm a Dog

Animals are living organisms

Hello, my name is Arun

Explanation

1. **Static Method Issue:**
   - Static methods (Animal.info()) are called directly on the class itself (Animal) and not on the instance (animal).
   - Older browsers like IE11 do not support static methods.
2. **Arrow Functions in Constructors:**
   - Arrow functions inherit this from their parent scope.
   - In the example, sayHello properly accesses this.name because the arrow function retains the this of the enclosing scope.
3. **Class Syntax:**
   - The class syntax makes constructor functions cleaner and more modern but requires transpilation (e.g., Babel) for older browsers.

Map and Set in JavaScript

Developers often correct data structures to manipulate data easily when required. It not only saves ample time but also makes the code easier to understand. They commonly use objects and arrays data structures to store data collections. The former is used to store key-value pairs, while the latter stores indexed lists.

However, ECMAScript 2015 introduced two new iterable objects- Maps and Sets. The two types of objects are used to store data in an ordered manner. They store distinct types of data within the same object. A Map guarantees an ordered collection of key-value pairs, whereas a Set is a collection of unique values.

This chapter will explore **JavaScript Map and Set**, their properties and methods, examples, and more.

What is a Map in JavaScript?

Maps in JavaScript are a collection of key-value pairs containing any data type and of any length. Keys uniquely identify an element, and values contain data associated with it. Maps and objects are similar, but maps can store any data type as a key, whereas objects store symbols, strings, and numbers as keys. Also, maps are iterable and include various methods. A **Map in JS** allows duplicate values and guarantees order.

Syntax

```
const myMap = new Map();
```

Example

const myMap = new Map();

myMap.set('name', 'Kanha');

myMap.set('age', 30);

console.log(myMap);

Output:

Map { 'name' => 'Kanha', 'age' => 30 }

Properties and Methods of Map

JavaScript maps include various methods and properties used to add, retrieve, delete, and search elements in a map. Let's talk about these properties and methods, their uses, and examples in the next section.

Add Values to the Map with the set() Method

Use the set() method to add values to a map. The first argument is the key, and the second is the value.

Example

Search a Key in the Map with the has() Method

Use the has() method to check if the map contains a specific key or element. It returns a boolean value.

Example

```
const myMap = new Map();

myMap.set('name', 'Arun');


console.log(myMap.has('name'));

console.log(myMap.has('age'));
```

Output:

true

false

Retrieve a Value from the Map with the get() Method

The get() method is used to retrieve a value by key from the map object. Pass the key as the argument.

Example

```
const myMap = new Map();

myMap.set('name', 'Arun');


console.log(myMap.get('name'));
```

Output:

Arun

Remove a Specific Key with the delete() Method

Use the delete() method to remove a single element from the map. Pass the key as the argument, and the method will return a boolean true value if the item exists and is deleted. If it can't find the match, it will return false.

Example

```
const myMap = new Map();

myMap.set('name', 'Aman');


console.log(myMap.delete('name'));

console.log(myMap.has('name'));
```

Output:

true

false

Remove All Elements with the clear() Method

Clear all elements of key-value pairs using the clear() method.

Example

```
const myMap = new Map();

myMap.set('name', 'Arun');

myMap.clear();


console.log(myMap.size);
```

Output:

0

Check the Number of Elements with the size Method

You can use the size method to see how many key-value pairs exist in a map. It will allow you access to the map.

Example

```
const myMap = new Map();

myMap.set('name', 'Arun');

myMap.set('age', 25);


console.log(myMap.size);
```

Output:

2

Return All Keys and Values with the entries() Method

The entries() method returns an array of key-value pairs.

Example

```
const myMap = new Map();
myMap.set('name', 'Arun');
myMap.set('age', 25);

for (const [key, value] of myMap.entries()) {
  console.log(key, value);
}
```

Output:

name Arun

age 25

Return All Keys with the keys() Method

The keys() method returns all the keys of a map and runs a for or for-of loop.

Example

```
const myMap = new Map();
myMap.set('name', 'Arun');
myMap.set('age', 25);

for (const key of myMap.keys()) {
  console.log(key);
}
```

Output:

name

age

Return All Values with the values() Method

The values() method returns all the values in the map.

Example

```
const myMap = new Map();

myMap.set('name', 'Sultan');

myMap.set('age', 25);


for (const value of myMap.values()) {

  console.log(value);

}
```

Output:

Sultan

25

Iterate Over a Map with the forEach() Method

Similar to an array, the built-in forEach() method in Map is used to iterate over a map. It iterates through the value, key, and map, but in an array, it iterates through the index, item, and array.

Example

```
const myMap = new Map();

myMap.set('name', 'Sultan');

myMap.set('age', 25);


myMap.forEach((value, key) => {
```

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

Output:

name: Sultan

age: 25

Here is a table showing methods and properties of Maps:

Properties/Methods	**Description**	**Returns**
set(key, value)	Adds a key/value pair to a Map	Map Object
has(key)	Checks if a specific key is present in a Map.	Boolean
get(key)	Returns a value by key and returns undefined if the key doesn't exist in the Map.	value
delete(key)	Removes a specific element from a Map by the key.	Boolean
clear()	Removes all elements from a Map.	N/A
size	Returns the number of items in a Map	Number

entries()	Returns an iterable for all keys and values in a Map as [key, value].	MapIterator object
keys()	Returns all keys in a Map.	MapIterator object
values()	Returns all values in a Map.	MapIterator object
forEach()	Iterates over the Map in insertion order.	N/A

What is a Set in JavaScript?

A set is a data structure containing a collection of unique values. Hence, no duplicate values are allowed in sets. **Sets in JS** are more similar to an array than objects because they are a list of values rather than key-value pairs. However, sets can't replace arrays but can supplement them and provide additional support. The values stored in Sets can be of any data type and length.

Developers use sets to remove duplicates from any data structures. Also, a set doesn't guarantee ordered elements, and you can access elements through index.

Syntax

```
const ids = new set();
```

Example

const mySet = new Set();

mySet.add(1);

```
mySet.add(2);

console.log(mySet);
```

Output:

Set { 1, 2 }

Properties and Methods of a Set

Similar to Maps, Sets also have various methods and properties used to add, retrieve, delete, and clear elements. So, have a look at the following properties and methods of Sets, including their uses and examples.

Add Elements to a Set with the add() Method

Use the add() method to add values to a set.

Example

```
const mySet = new Set();
mySet.add(1);

console.log(mySet.has(1));
console.log(mySet.has(2));
```

Output:

true

false

Check if a Specific Value Exists in a Set with the has() Method

Use the has() method to check if a set contains a specific value.

Example

```
const mySet = new Set();
mySet.add(1);
mySet.add(2);

console.log(mySet.size);
```

Output:

2

Check the Size of a Set with the size() Method

The size() method is used to find the size of a set. This method returns the number of elements in a set.

Example

```
const mySet = new Set();
mySet.add(1);

console.log(mySet.delete(1));
console.log(mySet.has(1));
```

Output:

true

false

Remove a Specific Value from a Set with the delete() Method

Delete a single element from a set using the delete() method.

Example

```
const mySet = new Set();
```

```
mySet.add(1);
mySet.add(2);
mySet.clear();

console.log(mySet.size);
```

Output:

0

Clear the Entire Set with the clear() Method

Clear all elements in a set using the clear() method.

Example

```
const mySet = new Set();
mySet.add(1);
mySet.add(2);
mySet.clear();

console.log(mySet.size);
```

Output:

0

Return All Values in a Set Twice with the entries() Method

The entries() method retrieves all elements in a set twice.

Example

```
const mySet = new Set();
```

```
mySet.add(1);
mySet.add(2);

for (const entry of mySet.entries()) {
  console.log(entry);
}
```

Output:

[1, 1]

[2, 2]

Return All Values with the values() Method

The values() method retrieves all values in a set.

Example

```
const mySet = new Set();
mySet.add(1);
mySet.add(2);

for (const value of mySet.values()) {
  console.log(value);
}
```

Output:

1

2

Iterate Over a Set with the forEach() Method

A Set has a built-in forEach() method used to iterate over a set. As there are no keys in sets, the first and second parameters of the method return the same value, so it is not useful outside Map. You can also use for…of to iterate over a set.

Example

```
const mySet = new Set();

mySet.add(1);

mySet.add(2);


mySet.forEach((value) => {

  console.log(value);

});
```

Output:

1

2

Here is a table showing methods and properties of Sets:

Properties/Methods	Description	Returns
add(value)	Adds a new value to a Set.	Set Object

has(value)	Checks for the presence of a specific value in a Set.	Boolean
size	Returns the element count of a Set.	Number
delete(value)	Removes the specified value from a Set.	Boolean
clear()	Removes all values from a Set.	N/A
entries()	Returns all values in a Set as [value, value]	SetIterator object
values()	Returns all elements in a Set (same as keys())	SetIterator object
forEach()	Iterates over the entire Set in the insertion order.	N/A

When to Use Maps in JavaScript?

Maps and objects are similar as they both hold key-value pairs. However, Maps can be treated like an upgraded version of an object that can have any data type as the key value. You can easily add or remove keys, which can be useful when you want to change data frequently. Maps offer a few more advantages over objects, which we have discussed below:

- They can contain any data type as a key to a value, whereas objects contain only strings.
- Maps have a size property, whereas objects don’t have a built-in method to retrieve the size.

- Maps retain the insertion order, but objects don't guarantee order.
- Maps are directly iterable, which is not the case with objects.
- Maps offer effective methods to manipulate data, such as entries() to get all key-value pairs, has() to check if the map contains a specific key, and values() to retrieve all values.

When to Use Sets in JavaScript?

Sets are a useful addition to JavaScript, especially when it comes to working with duplicate values. They are like an alternative version of arrays. Here are a few reasons to use sets in JavaScript:

- As sets ignore duplicate values, you can use them to remove duplicates from an array and convert the set back to an array.
- With duplicate values removed from data, the size of the data structure is also reduced.
- Due to the implementation using hash tables, sets offer faster membership tests and lookups.
- You can easily combine a set with other sets to perform different operations and find intersections, unions, and differences between two sets of data.
- You can use them to compare multiple set objects using various methods, which are not available in arrays.

Difference Between Map and Set in JavaScript

Hitherto, you have learned almost everything about Maps and Sets in JavaScript. It's time to see the difference between the two:

Map	**Set**
Refers to a collection of key-value pairs.	Refers to a collection of unique elements.
We can access values using keys.	We use built-in methods to access values.

Used when we want the data to be identified by a key.	Used when we want to store unique values.
Two-dimensional.	One-dimensional.

JavaScript Arrays

An **array in JavaScript** is implemented as an object containing an ordered collection of elements or values. Hence, you can access these elements through their position. Arrays can store different data types, and there is no fixed size of an array, which means you can add as many elements as you want to an array.

This chapter focuses on how we use **arrays for JavaScript** programming, different operations we can perform on arrays, and array methods.

What are Arrays in JavaScript?

A **JavaScript array** is a data structure that stores multiple values in a single variable. It can store different data types and supports dynamic resizing. We deal with array objects in the same way we deal with other data types. However, arrays allow us to access their elements through their respective index, which starts from 0, and perform various operations on them.

Why do we Use Arrays in JavaScript?

In JavaScript, arrays allow us to organize related data in a group within a single variable. For example, if you want to store 100 employee names, you can do so easily using arrays and store as many values as you want in a single variable. You can also access it by referring to the corresponding index number.

Declaring an Array in JavaScript

We can declare an array in JavaScript in the following two ways:

1. Using Array Literal

We can create an array through array literal, which involves using square brackets []. The square bracket is a literal notation used to create and initialize an array. We separate each element using a comma. It is a commonly used and concise method known for its simplicity.

Syntax

```
let arrayName = [value1, value2, ...];
```

Example

let fruits = ['apple', 'banana', 'cherry'];

console.log(fruits);

Output:

`['apple', 'banana', 'cherry']`

2. Using an Array Constructor (new keyword)

An array constructor is an array creation method that involves invoking the array constructor function. This method allows dynamic initialization, which we can use to create arrays with specified lengths and elements. In this method, we create an instance of an array by passing arguments in a constructor. Hence, we don’t need to provide value explicitly.

Syntax

```
let arrayName = new Array();
```

Example

```
let colors = new Array('red', 'green', 'blue');

console.log(colors);
```

Output:

```
['red', 'green', 'blue']
```

Basic JavaScript Array Operations

We can perform different operations on each element of a JavaScript array. Let's discuss these operations in detail:

Accessing Array Elements

Every element in an array is assigned a number known as an index, which represents its position in an array. By default, the first index is zero. To access a specific element, you need to use square brackets [] containing the index value of that element.

For example,

```
let numbers = [10, 30, 40, 60, 80];
```

The index of each element of the array is:

Indexes →	0	1	2	3	4
	10	30	40	60	80

Example

```
let numbers = [10, 20, 30];

console.log(numbers[1]);
```

Output:

```
20
```

Accessing the First Element of an Array

As the indexing starts from zero, you can access the first element of an array using its index number.

Example

```
let animals = ['dog', 'cat', 'rabbit'];

console.log(animals[0]);
```

Output:

```
'dog'
```

Accessing the Last Element of an Array

As the arrays use zero-based indexing, you can access the last element using [array.length – 1].

Example

```
let cities = ['London', 'Paris', 'Tokyo'];

console.log(cities[cities.length - 1]);
```

Output:

```
'Tokyo'
```

Array Length Property

The length property is used to retrieve the length of an array so you can see the number of elements an array has.

Example

```
let cars = ['Tesla', 'BMW', 'Audi'];

console.log(cars.length);
```

Output:

3

Modifying Array Elements

We can modify or change the elements in an array by assigning a new value to their corresponding index.

Example

```
let names = ['Alice', 'Bob', 'Charlie'];

names[1] = 'David';

console.log(names);
```

Output:

```
['Alice', 'David', 'Charlie']
```

Adding Elements to an Array

We can add elements to a JS array using the following two built-in methods:

push() method-

It adds an element to the end of the array.

Example

```
let books = ['Math', 'Science'];

books.push('History');

console.log(books);
```

['Math', 'Science', 'History']

unshift() method- It adds an element to the beginning of the array.

Example

```
let languages = ['Python', 'Java'];

languages.unshift('JavaScript');

console.log(languages);
```

Output:

```
['JavaScript', 'Python', 'Java']
```

Removing Elements from an Array

We can remove elements from an array using different methods explained below:

pop() method- It removes the last element from an array.

Example

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

fruits.pop();

console.log(fruits);
```

Output:

```
['Apple', 'Banana']
```

shift() method- It removes the first element from an array.

Example

```
let tools = ['Hammer', 'Screwdriver', 'Wrench'];

tools.shift();

console.log(tools);
```

Output:

```
['Screwdriver', 'Wrench']
```

splice() method- It removes or replaces elements from the middle of an array.

Example

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

numbers.splice(2, 1);

console.log(numbers);
```

Output:

```
[1, 2, 4, 5]
```

Increasing and Decreasing the Array Length

We use the JavaScript length property to increase or decrease a **JS array length**.

Example -1

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

arr.length = 5;

console.log(arr);
```

Output:

```
[1, 2, 3, <2 empty items>]
```

Example - 2

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

arr.length = 2;

console.log(arr);
```

Output:

```
[1, 2]
```

Iterating Through Elements in an Array

You can loop over array elements to access multiple elements at once. For this, you can use the forEach method, which calls the function once for every element of an array.

Example

```
let scores = [10, 20, 30];

scores.forEach(score => console.log(score));
```

Output:

```
10

20

30
```

Converting an Array to a String

For conversion of an array to a string, use the built-in method called toString(). Separate the elements using commas. You can also use the join() method to pass a specific string separator as the argument.

Example -1

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

console.log(arr.toString());
```

Output:

```
'1,2,3'
```

Example -2

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

console.log(arr.join('-'));
```

Output:

```
'1-2-3'
```

Concatenating Arrays

To combine two or more arrays, you can use the concat() method, as it returns a new array with joined array elements.

Example

```
let arr1 = [1, 2];

let arr2 = [3, 4];
```

```
let combined = arr1.concat(arr2);

console.log(combined);
```

Output:

```
[1, 2, 3, 4]
```

Checking the Type of an Array

JavaScript allows you to check the type of an array using the typeof operator, which returns “object” for arrays.

Example

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

console.log(typeof arr);
```

Output:

```
'object'
```

Checking if a Variable is an Array

Use the Array.isArray() method to check if a variable is an array. It tests whether the argument is an array or not and returns true if you pass an array and false for anything else.

Example

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

console.log(Array.isArray(arr));
```

Output:

```
true
```

Comparing Two Arrays

In JavaScript, arrays are treated as objects. When you compare two arrays, it will look at the address of the memory location that stores the array, i.e., the reference, rather than the actual values. Therefore, when elements are the same but stored in different memory

locations, they return false information. If both variables refer to the same array object, it returns true.

Example

```
let arr1 = [1, 2];

let arr2 = [1, 2];

console.log(arr1 === arr2);
```

Output:

```
false
```

Copying an Array

You can copy an array using the slice() method, which returns a shallow copy, so the values in the copy are references to the original array. If an array contains primitive values, such as booleans, numbers, or strings, the shallow copy won't lead to a problem but can be an issue if the copy contains an array of objects.

Example

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

let copy = arr.slice();

console.log(copy);
```

Output:

```
[1, 2, 3]
```

Searching an Array

You can search for an array using the following three methods. Choose the one based on the result you want to achieve.

- Find the value that aligns with certain criteria in an array
- Find if an element exists in the array
- Find the index position of an array element.

Finding If an Element Exists in an Array

To know if a specific element exists in an array, use the includes() method, which returns true when the element is found; else, it returns false.

Example

let arr = [1, 2, 3];

console.log(arr.includes(2));

Output:

`true`

Finding the Index Position of an Element in an Array

To find the index position of an element, use the indexOf() method.

Example

let arr = [1, 2, 3];

console.log(arr.indexOf(2));

Output:
1

Finding Elements that Meet Certain Criteria in an Array

Use the built-in **JS array filter()** method to find elements that meet certain criteria. This method is used for array objects and helps filter an array.

Syntax

```
arrayObject.filter(callback, thisContext);
```

Parameters

callback- It is the necessary parameter that is executed for every array value.

thisContext- It is optional and refers to the value of the this keyword inside the callback. You need it only to define the callback function that accepts three arguments:

- The index of the element starting from 0.
- The array object to call filter().
- The currentElement processed into the method.

Example

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

let even = arr.filter(num => num % 2 === 0);

console.log(even);
```

Output:

```
[2, 4]
```

Difference Between JavaScript Arrays and Objects

Let's understand the difference between JavaScript arrays and objects:

Feature	**JavaScript Array**	**JavaScript Object**
Uses	To store sequences, lists, and ordered data.	To store key-values pairs and attributes.
Iteration	Iteration occurs using for or forEach method.	Uses for…in, Object.keys(), or Object.entries() for iteration.

When to Use	When you need numeric indexing and order.	When the relationship between key-value matters and you need named keys.
Order	It is an ordered collection.	It is an unordered collection.
Size Flexibility	It can grow or shrink dynamically.	It allows to add or remove key-value pairs.
Index Type	Numeric indexes (0, 1, 2, …)	Named keys (strings or symbols)
Accessing Elements	Can be accessed using index.	Can be accessed using the key.

JavaScript Array Methods

Array methods in JavaScript are built-in functions used to manipulate and traverse arrays. They offer necessary functionalities, including adding, removing, or modifying elements. You can also use them to sort elements to enhance code readability. Here are a few commonly used JavaScript array methods:

Method	**Description**
concat()	Merges two or more arrays and returns a new merged array object.
sort()	Sorts the elements alphabetically and in ascending

	order and returns the elements.
slice()	Returns a new array with a copy of the part of the original array, depending on the start and end index given as arguments.
splice()	Adds or removes elements from the given array.
pop()	Removes the last element of the array.
push()	Adds an element to the end of an array.
toString()	Converts the elements of a specified array into a string without changing the original array.
unshift()	Adds elements in the beginning of the array and increases the index of each element by one.
shift()	Removes the first element of an array and shifts all elements, reducing the index of each element by one.
reduce()	Reduces the array to a single value and executes the given function for each value (from left to right) to get a single output value.
reverse()	Reverses the order of elements of a given array and returns a reference to the same array with reversed order.
values()	Creates a new iterator object with values for each index in the array.
map()	Calls a specific function on each element of an array present in the parent array and returns a new array.
some()	Finds if any element of the array meets the condition checked by the argument function.
flat()	Flattens or merges the array and reduces all the nesting present in it.
join()	Concatenates all the elements of an array and separates each element using the specified separator to return them as a string.

JavaScript Number

A **number in JavaScript** is used to represent numeric values. It can be a floating-point or integer. The number object follows the IEEE standard to represent floating-point numbers.

Here is a chapter that will help you understand JavaScript Numbers and related concepts in detail through examples.

What is a Number in JavaScript?

The **JS number** is a primitive data type that contains different properties and methods to enable you to work with numbers. Unlike other programming languages, JavaScript doesn't require you to declare different numeric types like float, int, etc. As the browser automatically converts number literals to instances of the number class, you don't need to be concerned about the number object.

The **JS number format** to store this data type is the double-precision 64-bit binary format defined by the IEEE 754 standard, which stores numbers in 64 bits:

- 0-51 bits store the fraction value
- 52-62 bits store the exponent
- 63rd bit stores the sign

We can define a number object using the Number constructor and convert other data types to numbers using the Number() function. JS doesn't distinguish between integer and floating-point.

Key Points to Remember

- If constants are preceded by 0x, they are considered hexadecimal.
- By adding a string and a number, we get a string concatenation.
- By default, the numbers in JS are of base 10. However, when we use toString(), we can get the result in the required base ranging from base 2 to base 36.
- Numbers are primitive values that can be defined as objects by using the new keyword.
- JavaScript also has BigInt numbers, which are integers with arbitrary length.

JavaScript Number Syntax

The syntax to create a JS number object is:

```
const val = new Number(number);
```

If you provide a non-numeric argument in place of ‘number’, the argument can’t be converted to a numeric value, and the code returns NaN, i.e., Not-a-Number.

You can also assign a number to a variable:

Example

const val = new Number(42);

console.log(val);

Output

[Number: 42]

JavaScript NaN

Not a Number (NaN) is a special value returned when the mathematical operation performed on the input can’t give a numeric result. If we perform an arithmetic operation (except addition) on numbers and strings, it will return NaN.

We use the typeof operator for the NaN value, which gives a number as the result. To find if a value is a number, use the built-in isNaN() function.

Example

console.log(Number('abc'));

console.log(isNaN('abc'));

Output

NaN

true

JavaScript Numeric Types

Numbers are crucial in JavaScript, so you need to gain a fair understanding of their behavior to be a successful programmer. Here are the different aspects of **JS number types**:

Numeric Type

In JavaScript, there is only one numeric type- the double-precision 64-bit binary format IEEE 754. It means that there is no difference between floating-point and integer numbers. It uses the unified approach for numeric values. The number precision is 53 bits, which accurately represents the integer values between -2^53 + 1 to 2^53 – 1.

Example

let num = 42;

let float = 42.5;

console.log(typeof num);

console.log(typeof float);

Output

number

number

Floating-point Precision

The binary representation restrictions don't guarantee 100% accuracy for floating point arithmetic.

Example

console.log(0.1 + 0.2);

Output:

0.30000000000000004

Scientific Notation

We can use scientific notation to write extra-large or extra-small numbers.

Example

let sci = 5e3; // 5 * 10^3

console.log(sci);

Output

5000

Numeric Strings

JavaScript can automatically convert a numeric string to a number for most operations.

Example

```
let strNum = "123";

console.log(Number(strNum));
```

Output

123

Integer Precision

Integers are numbers without a period or exponent notation. They are accurate up to 15 digits.

Example

```
console.log(9999999999999999);
```

Output

10000000000000000

Adding Number and String

The + operator is used to add or concatenate numbers.

Example

```
console.log(10 + "20");
```

Output

"1020"

JavaScript Number Literals

Here are different types of number literals you can use in JavaScript:

Binary Numbers

These numbers begin from 0b or 0B, followed by 0s and 1s.

Example

let bin = 0b1010;

console.log(bin);

Output

10

Octal Numbers

Octal numbers start with 0, followed by a number smaller than 8.

Example

let oct = 0o12;

console.log(oct);

Output

10

Hexadecimal Numbers

Hexadecimal notation starts with 0x or 0X followed by any of the following digits: 0123456789ABCDEF.

Example

```
let hex = 0xFF;

console.log(hex);
```

Output

255

Decimal Numbers

Unlike other programming languages, JavaScript doesn't have different types of numbers. There is only one type that can hold both decimal and non-decimal values.

Example

```
let dec = 10.5;

console.log(dec);
```

Output

10.5

Number Coercion in JavaScript

Coercion in JavaScript is the automatic or implicit conversion of one data type to another. When we apply different operators to values, JS performs type coercion to complete the operation. Have a look at the following coercion examples:

Boolean to Number

In JavaScript, boolean values can be converted to numbers- 1 for true and 0 for false.

Example

```
console.log(Number(true));

console.log(Number(false));
```

Output

1

0

Null to 0

When used in arithmetic operations, the null value is converted to 0.

Example

console.log(Number(null));

Output

0

String to Number

For arithmetic operations, string values are converted to numbers. If the string can't be parsed as a valid number, it returns NaN.

Example

console.log(Number("42"));

Output

42

Undefined to NaN

In case of performing an operation on an undefined value, JavaScript returns NaN.

Example

console.log(Number(undefined));

Output

NaN

BigInts and Symbols

If we try to convert symbol values to numbers, it will lead to TypeError.

Example

```
try {
   console.log(Number(Symbol('symbol')));
} catch (e) {
   console.log(e.message);
}
```

Output

Cannot convert a Symbol value to a number

Integer Conversion

We use integers for operations involving spring indexes, arrays, or date/time. After coercion, if the number is greater than 0, it is returned the same, but if the number is -0 or NaN, it returns 0. The output value is always integer.

Example

```
console.log(Number.parseInt('42.9'));
```

Output

42

Fixed-width Number Conversion

A few JavaScript functions involve binary encoding of integers, such as typedArray objects and bitwise operators. The latter always converts the operands to 32-bit integers.

Example

```
console.log(5 | 3);
```

Output

7 (Bitwise OR converts to 32-bit integers)

JavaScript Precision Problems

JS numbers, especially decimals, may not always be stored accurately, which can cause precision issues. This is because JavaScript stores numeric values in the binary form, representing decimal digits, and we can't represent these digits exactly in the binary format. Hence, we use two ways to solve this problem:

- Use the toFixed() method.
- Convert floating point to integers.

Example

let result = (0.1 + 0.2).toFixed(2);

console.log(result);

Output

"0.30"

JavaScript Number Constructor

A number constructor in JavaScript is called when we create an object using the new keyword.

Constructor	**Description**
Number()	Returns the numeric value for any type of variable.

JavaScript Number Properties

A property is a member of an object associating a key and a value. JavaScript has two types of Number properties:

Static Properties- They have the same value for the entire class**.** Examples include MIN_VALUE, MIN_VALUE, EPSILON, NaN, NEGATIVE_INFINITY, and

POSITIVE_INFINITY.

Instance Properties- They have a new copy for each new instance of the class. For example, constructor.

Property	Description
EPSILON	Represents the difference between 1 and the smallest floating-point numeric value, which is greater than 1.
MIN_VALUE	Returns the smallest value a JavaScript number can have.
MAX_VALUE	Returns the maximum value a JavaScript number can have.
prototype	Assigns new properties and methods to the Number object in the current document. It is a static property of the Number object.
MAX_SAFE_INTEGER	Returns the maximum safe integer.
MIN_SAFE_INTEGER	Returns the minimum safe integer.
POSITIVE_INFINITY	Returns the largest positive infinity, overflow value. This value is greater than MAX_VALUE

NEGATIVE_INFINITY	Returns the largest negative infinity, overflow value. It is less than MIN_VALUE.
NaN	Returns a value that is Not-a-Number.
constructor	Returns the number constructor function that created this instance of the object. By default, it is the Number object.

JavaScript Number Methods

The Number Methods are actions performed on objects. These are of two types:

Static Method- They are called using the Number class. For example, isFinite(), isInteger(), isNaN(), and parseInt().

Instance Method- They are called on the instance of a number. For example, toFixed(), toPrecision(), valueOf(), and toExponential().

Method	Description
isNaN()	Finds whether the passed value is NaN or is of Number type.
isFinite()	Finds whether the passed value is a finite number or not.
isInteger()	Checks if the passed value is an integer. Returns true if it is; else, returns false.

isSafeInteger()	Checks whether the passed value is a safe integer.
parseFloat()	Accepts the string and converts the given string to a floating-point number.
parseInt()	Converts the given string to an integer value.
toExponential()	Returns a string value representing a number in exponential notation.
toFixed()	Returns a string representing a number in fixed-point notation, i.e., exact digits after decimal.
toPrecision()	Returns a string for a number to the specified precision.
toString()	Converts the number into a string and returns it.
valueOf()	Returns the number's primitive value.
toLocaleString()	Converts a number into a string and returns it in a local language format.

JavaScript String

A **string in JavaScript** is a sequence of characters that contains 0 or more characters. It allows you to work with a series of characters and wraps the string data type with helper methods.

So, let's learn about the string, operations performed on the string, and more.

What is a String in JavaScript?

The **JavaScript String** is a primitive data type representing a sequence of characters, especially textual data. Similar to Python and Java, JS strings are immutable. JavaScript doesn't have a character data type. Hence, we use a single-character string whenever we need a character. We can use a single quote or double quote to declare a string. However, make sure to be consistent throughout the code.

Features of the String in JavaScript

The JavaScript String showcases some of the most amazing features, which include:

- Strings are case-sensitive. Hence, the lowercase and uppercase values are treated differently. For example, "bye" and "BYE" are treated as different values.
- They are immutable, so you can't change the characters of a string. However, JavaScript allows you to assign a new value to the string variable.

Creating a JavaScript String

In JavaScript, we can declare or create a string using various ways:

Using Single Quotes

Example

```
let singleQuote = 'Hello, World!';

console.log(singleQuote);
```

Output

```
Hello, World!
```

Using Double Quotes

Whether you use single quotes or double quotes, it's practically the same. So, choose according to you but make sure to maintain the consistency throughout the code.

Example

let doubleQuote = "Hello, JavaScript!";

console.log(doubleQuote);

Output

Hello, JavaScript!

Creating an Empty String

If you want to create an empty string, assign single or double quotes without any characters in between.

Example

let emptyString = '';

console.log(emptyString);

Output

(empty output)

Using Template Literals (Using Backticks)

You can also create strings using Template Literals, which lets you embed expressions within backticks (`). This method dynamically creates strings, which makes it more versatile and readable. It is used when you want to add expressions or variables into a string, which you can do by wrapping variables with $.

Example

let name = "Sultan";

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

console.log(greeting);

Output

Hello, Sultan!

Using String Constructor

It is a less common method to directly create a string. The syntax for the string creation using the string constructor is:

```
var val = new String(value);
```

Here, value is a series of characters encoded properly.

Example

let constructedString = new String("Constructed String");

console.log(constructedString);

Output

[String: 'Constructed String']

Basic Operations on JavaScript Strings

JavaScript allows you to perform various operations on strings that we will discuss in this section:

1. Access String Characters

You can access characters of a string in two ways:

- The charAt() Method

To access an individual character in a string, use the bracket notation and charAt() method.

Example

let str = "JavaScript";

console.log(str.charAt(2));

Output

v

- Indexes

Treat the string as an array and access a character through the specific index.

Example

let str = "JavaScript";

console.log(str[2]);

Output

v

2. Find the Length of the String

You can find the string length using the built-in length property.

Example

console.log(str.length);

Output

10

3. Escape Characters

Escape characters are used to add characters that are impossible or challenging to represent directly in a string. You can use escape characters to add a new line, double quotes, and backslash. Below, we have shared different ways to use the escape characters.

Code	**Character**
\n	New Line

\r	Carriage Return
\\	Backslash
\f	Form feed
\"	Double Quote
\b	Backspace
\v	Vertical Tab
\t	Horizontal Tab

Example

let escapeExample = "First Line\nSecond Line\tTabbed";

console.log(escapeExample);

Output

First Line

Second Line Tabbed

4. Insert a Quote Inside Another Quote

Although you can write a quote inside another quote, it shouldn't match the surrounding quotes.

Example

```
let quote = 'She said, "Hello!"';

console.log(quote);
```

Output

```
She said, "Hello!"
```

5. Concatenate the String

To combine two or more strings, use the + operator.

Example

```
let part1 = "Hello";

let part2 = "World";

let result = part1 + " " + part2;

console.log(result);
```

Output

```
Hello World
```

6. Create Multiline Strings

In JavaScript, you can use various methods to create multiline strings. We have explained them below:

a) Multiline in code and output

You can use two techniques to print multiple lines with a single string variable:

- Newline Escape Character \n

Example

```
let multiLine = "Line 1\nLine 2\nLine 3";

console.log(multiLine);
```

Output

Line 1

Line 2

Line 3

- Template Literals

Example

```
let multiLineTemplate = `Line 1
Line 2
Line 3`;
console.log(multiLineTemplate);
```

Output

Line 1

Line 2

Line 3

b) Multiline in code but not in output

Use the + or \ operator to break a long string into multiline. When you separate a long string into multiple lines, the code becomes cleaner. However, the multiple strings will be shown as a single line in the output. Using the backslash may not be supported in all browsers, so in most cases, it's better to use the string addition.

Example

```
let longString = "This is a very long string that " +
                "spans multiple lines for better readability.";
console.log(longString);
```

Output

This is a very long string that spans multiple lines for better readability.

7. Search a String in JavaScript

Use the indexOf() method to find the first index of a substring in a string.

Example

```
let searchString = "Learn JavaScript";
console.log(searchString.indexOf("JavaScript"));
```

Output

6

8. Convert to String

The String() method is used to convert different data types to strings.

Example

```
let num = 123;
let strNum = String(num);
console.log(strNum);
```

Output

"123"

9. Trim Whitespace from a String

Remove leading and trailing whitespaces from a string by using the trim() method.

Example

```
let untrimmed = "   Hello World   ";
console.log(untrimmed.trim());
```

Output

"Hello World"

10. Find the Substring of a String

To extract a part of a string, use the substring() method.

Example

```
let sub = "JavaScript";

console.log(sub.substring(0, 4));
```

Output

Java

11. Compare a String

JavaScript provides built-in methods to compare strings. You can use the equality operator or localeCompare() method for this purpose.

Example -1

```
console.log("apple" === "apple");

console.log("apple" === "Apple");
```

Output

true

false

Example - 2

```
console.log("apple".localeCompare("banana"));
```

Output

-1

12. Replace a String

Replace occurrences of a substring within another by using the replace() method.

Example

let sentence = "I love JavaScript";

let newSentence = sentence.replace("JavaScript", "Python");

console.log(newSentence);

Output

I love Python

13. Convert String to Uppercase and Lowercase

USe the toUpperCase() and toLowerCase() methods to convert a string's characters to uppercase and lowercase, respectively.

Example - 1

let lower = "hello";

console.log(lower.toUpperCase());

Output

HELLO

Example - 2

let upper = "WORLD";

console.log(upper.toLowerCase());

Output

world

JavaScript String Methods

When we use string methods, JavaScript strings are automatically converted to string objects. This is called auto-boxing. Here are methods that you can call on strings:

Methods	Description
at()	Finds the character at the given index.
concat()	Combines two or more strings.
charAt()	Returns that character at the given string index.
charCodeAt()	Returns a Unicode value of the character at the given index. It accepts one argument.
anchor()	Creates an anchor element that we can use as a hypertext target.
indexOf()	Returns the index of the first occurrence of the character value in the given string.
match()	Finds a matching string with a regular expression.
matchAll()	Returns iterators matching the reference string with a regular expression.

startsWith()	Checks if the given string begins with the specific characters of the string.
endsWith()	Checks if the specified string ends with the characters of the specified string.
includes()	Checks if a string contains the given substring. Returns true if the substring is found; else, returns false.
codePointAt()	Returns a non-negative integer value.
lastIndexOf()	Returns the index of the last occurrence of the character value in the given string.
localeCompare()	Compares two elements and returns a positive number.
repeat()	Creates a new string with a given number of copies.
replace()	Replaces a part of the specified string with another one or a regular expression while keeping the original string unchanged.

replaceAll()	Replaces all the matches of a string with a string or RegEx and keeps the original string unchanged.
padEnd()	Pads a string with another string at the end to the given length.
padStart()	Pads a string with another string at the beginning of the given length.
search()	Searches for the given regular expressions in the string.
slice()	Returns a part of the given string based on the index number at the beginning and end as parameters.
split()	Splits the string into substrings based on the specified separator provided as the parameter.
substr()	Extracts a part of the original string and returns a new one. The number of characters depends on the specified index.
substring()	Returns a part of the string from the start to the end index.

toString()	Converts values to a string and returns it.
normalize()	Returns the Unicode normalization of the given string.
toUpperCase()	Converts the characters of an entire string to uppercase. It takes one parameter stringVariable string to be converted to uppercase.
toLowerCase()	Converts all the characters of an entire string to lowercase.
toLocaleLowerCase ()	Converts a string to lowercase based on the host's current locale.
toLocaleUpperCase ()	Returns the string value converted to uppercase based on the host's current locale.
valueOf()	Returns the primitive value of the string.
String.raw(str, …sub)	Returns the raw string format of template literals.the string doesn't process escape characters.

trim()	Removes the white spaces from both ends of the string and returns a new string. It doesn't accept any parameters.
trimEnd()	Removes whitespace from the end of a string without changing the value of the string and the whitespace before the string.
trimStart()	Removes whitespace from the beginning of the given string, without changing the value of the string, including the whitespace after the string.
fromCharCode(n1, n2, …, nX)	Creates a string from the given UTF-16 code units and returns a string, not a string object.
fromCodePoint(a1, a2, a3, ….)	Returns a string or an element using the given code point values (ASCII value).
toWellFormed()	Returns where all lone surrogates of the string are replaced with the Unicode replacement character.
isWellFormed()	Checks if the string contains a lone surrogate.

JavaScript Symbol

The **Symbol in JavaScript** is a type of primitive data introduced in ES6. Developers use the symbol to represent a unique value that can be used as a key or identifier in objects. It is also used to create private methods and properties in classes.

Let's dive deeper into a JavaScript symbol, understanding its benefits and more through examples.

What is a Symbol in JavaScript?

The **JavaScript symbol** is a new primitive data type created using the Symbol() function. It represents a unique identifier and is used in different ways. We can use it to create object properties to assign a unique identifier to an object.

Symbols are immutable and unique, so they are useful for creating property keys that are unique across different objects. They can also be used to create non-enumerable properties in objects. They are used to create unique constants, making them a useful choice for creating APIs.

While creating a symbol, we can use an optional string parameter to describe the symbol. This helps debug and identify the symbol.

Key Points to Remember

- The Symbol() function returns a unique value.
- Symbols define unique object properties.
- They can't be typecasted to primitive data types.
- The type of a symbol can't be changed.
- The value of a symbol can be used as an identifier for object properties.
- They are immutable, so their value can't be updated or modified.

JavaScript Symbol Syntax

Here is a syntax to create a symbol:

```
const sym = Symbol([description]);
```

Parameters

description- It is an optional parameter used to give a description of the symbol.

Creating Symbol in JavaScript

As explained above, we use the **JavaScript Symbol() function** to create a symbol. We can pass an optional string as a parameter.

Example

```
const sym = Symbol('uniqueIdentifier');

console.log(sym);
```

Output:

```
Symbol(uniqueIdentifier)
```

Accessing Symbol Description

We use the . operator to access the description of a symbol.

Example

```
const sym = Symbol('descriptionExample');

console.log(sym.description);
```

Output:

```
descriptionExample
```

Adding Symbol as an Object Key

To add symbols as a key in an object, we use square brackets [].

Example

```
const symKey = Symbol('id');
```

```
const obj = {
  [symKey]: 123
};
console.log(obj[symKey]);
```

Output:

123

Please Note- The for…in loop doesn't iterate over Symbol properties.

Example

```
const symKey = Symbol('hidden');
const obj = {
  visible: 'I am visible',
  [symKey]: 'I am hidden'
};

for (let key in obj) {
  console.log(key);
}
console.log(obj[symKey]);
```

Output:

visible (Symbol property is skipped)

I am hidden

JavaScript Symbol Constructor

The constructor is called when we create an object.

Constructor	Description
Symbol()	Creates a symbol object

JavaScript Symbol Properties

The property in JavaScript refers to a member of an object associating a key with a value.

Static Properties

They have the same value for the entire class.

Properties	Description
asyncIterator	Sets the default object as async iterable
hasInstance	Determine if a constructor object identifies an object as its instance
isConcatSpreadable	When the Array.prototype.concat() method is used, the object should be flattened to its array elements

iterator	Returns the default iterator and makes an element easier to use in the for..of loop
match	Matches a regular expression to a string
matchAll	Returns an iterator that yields a regular expression that matches the String
replace	Replaces matched substring of a string
search	Returns the index in a string that matches a regular expression
species	Creates a derived object using the function-valued property
split	Splits a string at the indices that match a regular expression
toPrimitive	Converts an object to a primitive data value
toStringTag	Creates the default string description of an object

Instance Properties

They have a new copy for each new instance of a class.

Properties	Description
constructor	Returns the string constructor function for an object
description	Returns an optional description for the given symbol object

JavaScript Symbol Methods

Methods in the JavaScript Symbol are actions performed on objects. There are two types of JavaScript Symbol methods:

Static Methods

They are called using the symbol class.

Methods	Description
for()	Searches for the given symbol

keyFor()	Retrieves a shared symbol from the global symbol registry.

Instance Methods

They are called on the instance of a symbol.

Methods	**Description**
toString()	Convert a symbol object into the string
valueOf()	Returns the primitive data value of the symbol object
toSource()	Returns a string that contains the source of the Symbol object

Benefits of Using Symbols in JS

Now, let's take a look at the benefits of using **symbols in JS** for real-time development.

- As a symbol is unique, it helps avoid overwriting similar properties.
- In JavaScript, if you add a symbol as a key and traverse object properties using the for…in loop, it won't traverse through the symbol keys.

- Each symbol is unique in JavaScript, so you can use it as a key to avoid accidental collisions between keys that have the same name.
- It is used when you use the same instance of the object in two different codes and add the same properties.
- The symbol can also be useful in defining private properties in JavaScript classes.

JavaScript Proxies

JavaScript Proxy is an object used to wrap an object and customize its different operations, including reading, validating, etc. Developers use proxies to add custom behavior to an object.

Let's delve deeper and understand more about JavaScript Proxy, its uses, and examples.

What is a Proxy in JavaScript?

Proxies in JavaScript are objects that intercept other objects and redefine their fundamental operations. We use a proxy to hide the details of an object from any unauthorized access. Proxy in JS allows us to add custom behavior to an object and implement features, such as caching, logging, and securities.

There are two parts of a proxy- its target and handler. The former is an object in JS on which the proxy is applied, whereas the latter includes the function required to intercept an operation on it.

We can create a proxy object using the Proxy() constructor in JavaScript. The constructor accepts two parameters – the target object and the handler object. It returns a new proxy object for the target object.

Syntax to Create a Proxy in JS

We can create JavaScript proxies using the Proxy() constructor that accepts two parameters- target object and handler object.

```
const prox = new Proxy(tar, handle)
```

Parameters

new Proxy()- It is the constructor

tar- The object on which the proxy is applied.

handle- The object to define the intercept condition.

It returns a new proxy object.

JavaScript Proxy Example

JavaScript Proxy Handlers

We use the proxy handlers to override the default behavior of an object. There are various proxy handlers in JavaScript, and we have explained a few in detail below.

JavaScript get() Proxy Handler

The get() proxy handler is used to access and change the properties of a target object.

Syntax

Here is a syntax for using the get() method with the proxy object.

```
get(target, property, receiver)
```

Parameters

target- A target object.

property- Whose value is to be accessed.

receiver- The proxy object.

Example

JavaScript set() Proxy Handler

The set() proxy handler is used to set the value of an object.

Syntax

Here is the syntax to use the set() method:

```
set(target, property, value, receiver)
```

Parameters

target- The target object.

property- The property to change its value.

value- The updated value.

receiver- The proxy object.

Example

JavaScript apply() Proxy Handler

The apply() proxy handler is the trap to change the default behavior of the function call.

Syntax

Here is the syntax to use the apply() proxy handler:

```
apply(target, thisArg, arguments)
```

Parameters

target- The target function to execute.

thisAg- The context whose values must be accessed using the this keyword.

arguments- An array of arguments to be passed to the function.

Uses of Proxy in JavaScript

Now that you have understood JS proxies and proxy handlers, let's take a look at the top uses of the **JavScript proxy**:

Default Values

In case of an undefined property of an object or a value in an array, you can provide default values.

Validation

Example

```
const handler = {
  get: (target, property) => property in target ? target[property] : 'Default Value'
};
```

```
const obj = { name: 'Alice' };
const proxy = new Proxy(obj, handler);

console.log(proxy.name);
console.log(proxy.age);
```

Output:

Alice

Default Value

You can check the key value and take action depending on the value. Proxies are used for validation.

Example

```
const handler = {
  set: (target, property, value) => {
    if (property === 'age' && value < 0) {
      throw new Error('Age cannot be negative');
    }
    target[property] = value;
    return true;
  }
};

const obj = {};
const proxy = new Proxy(obj, handler);

proxy.age = 25;
console.log(proxy.age);

proxy.age = -5;
```

Output:

25

Throws Error: Age cannot be negative

Call A Function

You can call another function when a specific condition is fulfilled.

Example

```
const handler = {
  apply: (target, thisArg, argumentsList) => {
    console.log('Function is being called');
    return target(...argumentsList);
  }
};

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

const proxy = new Proxy(greet, handler);
console.log(proxy('Harun'));
```

Output:

Function is being called \n Hello, Harun!

DOM Manipulation

You can also use proxies to manipulate DOM nodes.

Example

```
const handler = {
  set: (target, property, value) => {
    if (property === 'textContent') {
      console.log(`Text changed to: ${value}`);
    }
    target[property] = value;
    return true;
  }
};

const element = document.createElement('p');
const proxy = new Proxy(element, handler);

proxy.textContent = 'Hello, Proxy!';
```

Output:

Text changed to: Hello, Proxy!

Read Only Object

When you don't want to allow others to make changes to an object, use a proxy to make it only readable.

Example

```
const handler = {
  set: () => {
    throw new Error('This object is read-only');
  }
};
```

```
const obj = { name: 'Sultan' };

const proxy = new Proxy(obj, handler);


console.log(proxy.name);


proxy.name = 'Bob';
```

Output:

Sultan

Throws Error: This object is read-only

Pros and Cons of Using JavaScript Proxies

There is no denying that Proxies in JavaScript offer many benefits and uses. However, they also have a few restrictions. So, here are the advantages and disadvantages of the proxy in JS:

Pros

- They enable DOM manipulation.
- As you have more control over how objects interact, it provides higher flexibility and more power over the implementation of fundamental object operations.
- You can allow a read-only view for data within objects to prevent unauthorized access or changes of data.
- With proxies, you can redefine the get method to generate the default values when the data or properties are not available.

Cons

- The proxy is like a mediator between the original object and a user, making the process more complex and adding to the performance overhead. Hence, when the performance is of the utmost importance, you must avoid using the proxy.
- No operation proxy forwarding is the method in which all operations and functionality of the target object are forwarded through a proxy, which can be achieved through the empty proxy handler function.

JavaScript Proxy Handlers List

Below is the list of proxy handlers in JavaScript:

Methods	Description
revocable()	Creates a new Proxy object similar to the Proxy() constructor.
apply()	Trap for a function call that changes its default behavior.
get()	Changes the behavior for accessing a property value.
set()	Adds or updates a property value of the object and returns a boolean value.
has()	Hides the property you want.
defineProperty()	Defines a new property and modifies the existing ones directly on an object.

deleteProperty()	Deletes the new property.
construct()	Trap for constructing the new object.
getOwnPropertyDescriptor()	Trap for the Object.getOwnPropertyDescriptor() method.
getPrototypeOf()	Trap for the internal methods.
setPrototypeOf()	Customizes the Object.setPrototypeOf() method and returns a Boolean value.
isExtensible()	Trap for the Object.isExtensible() method and returns a boolean value.
preventExtensions()	Trap for the Object.preventExtensions() method. It prevents the extension of an object and returns a boolean value.
ownKeys()	Trap for the Reflect.ownKeys() method and returns an enumerable object.

JavaScript Classes

JavaScript ES6 introduced classes that are blueprints to create objects. They are templates that encapsulate the data with code to work on the data. Classes in JS are the syntactic used to extend object properties and methods efficiently and elegantly. Earlier, inheritance and prototypes were used to achieve the same.

What are Classes in JS?

JavaScript classes are blueprints that are used to create objects. They encapsulate data and behavior by defining methods and properties, which allows object-oriented programming.

A class provides a cleaner syntax to write blueprints for object creation. It makes inheritance simpler and makes the code more organized. Classes and functions are similar, but we use the class keyword rather than function in classes. Also, classes are not hoisted.

The class name is user-defined, and we use the constructor method to initialize it.

JS Classes Syntax

```
class ClassName {
  constructor(param1) {
    this.proerty1 = param1;
  }
}
```

- We use the class keyword to create **JS classes.**
- ClassName refers to the name of the class, which is user-defined.
- The constructor() method is used to initialize the object properties and is executed when we create a new object of the class.
- We use the this keyword in Javascript to refer to the current object in a constructor. It ends the confusion between class attributes and parameters passed to the constructor with the same name.

Constructor Method

The constructor method is a special method to initialize objects created from a class. When a new instance of a class is created, the constructor method is called automatically. Based on the parameters passed, it initially assigns values to objects so that objects are initialized properly after creation.

Syntax

```
class ClassName {
  constructor() { ... }
  method() { ... }
}
```

Defining JavaScript Classes

The syntax for defining classes in JS contains the class keyword. Similar to functions using function declaration and function expression, classes use class declaration or class expression.

Syntax

```
// class declaration
class ClassName {
   // Class body
}
//Class expression
const ClassName = class {
   // class body
}
```

ClassName refers to the name of the class. A JS class is a function but can't be used as a regular function.

Example

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

  makeSound() {
    console.log(`${this.name} makes a sound`);
  }
}

const dog = new Animal('Dog');
dog.makeSound();
```

Output

Dog makes a sound

Using a Javascript Class

After declaring or defining a JS class, it's time to use it. Classes are blueprints to create objects, so we can't use them directly. Hence, we create objects to use classes. An object is a standalone entity that users create.

Example

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

```
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}

const person1 = new Person('Alice', 25);
person1.greet();
```

Output

Hello, my name is Alice and I am 25 years old.

Extending a Class in JavaScript

One of the best and most beneficial features of classes is that they can be extended to make a new object blueprint based on the parent class. This helps us avoid writing the code repeatedly for objects that are the same but require a few additional features. Through inheritance, we can define a new class that acquires the functionality of the parent class and extends that class while developing its own features. We can extend a class using the extend keyword.

Example

```
class Vehicle {
  drive() {
    console.log('Driving...');
  }
}

class Car extends Vehicle {
```

```
  honk() {
    console.log('Honking...');
  }
}

const car = new Car();
car.drive();
car.honk();
```

Output

Driving...

Honking...

JavaScript constructor() Method

As we use the function as a blueprint for an object, we can initialize its properties inside a function body. Similarly, we use the constructor() method with the class to initialize the properties in classes. When we create an instance of a class, it automatically invokes the constructor() method of the class.

If we define a class in JS without any constructor, the class will automatically create an empty constructor upon initialization. The constructor() method can be empty or take parameters.

Syntax

```
constructor(){
    ...
}
```

- We use the constructor keyword to create the constructor method in JS classes.
- () contains parameters to be assigned to the class attributes.

- {} contains the method definition.

Example

```
class Book {
  constructor(title, author) {
    this.title = title;
    this.author = author;
  }

  getDetails() {
    console.log(`${this.title} by ${this.author}`);
  }
}

const book = new Book('1984', 'George Orwell');
book.getDetails();
```

Output

1984 by George Orwell

Creating JavaScript Objects

We create an object of a class in JavaScript by using the new operator followed by the name of the class and a pair of parentheses.

Example: (Creating class objects without arguments)

```
class User {
  constructor() {
    this.name = 'Default User';
  }
}
```

```
const user = new User();
console.log(user.name);
```

Output:

Default User

Example: (Creating class objects with arguments)

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }
}

const product = new Product('Laptop', 1200);
console.log(product.name, product.price);
```

Output:

Laptop 1200

JavaScript Class Methods

You can define methods inside a class and access them using the class instance.

Syntax

Here is the syntax to define methods inside the class.

```
class car {
   methodName(params) {
                    // Method body
          }
}
obj.methodName();
```

methodName is the name of the method. We don't need to use any keyword before the method name to define a class method. We can invoke the class method by using the instance of the class. The obj in the syntax shows the class instance, and we can also pass parameters to the method.

There are three categories of class methods in JavaScript:

1. Instance methods
2. Static methods
3. Getter and Setter methods

Example

```
class Car {
  startEngine() {
    console.log('Engine started');
  }
}

const myCar = new Car();
myCar.startEngine();
```

Output:

Engine started

JavaScript Class Hoisting

The class declaration in JavaScript is not hoisted at the top of the code. It means that the classes in JS can't be accessed before declaring them in the program. You must define the class before using it. Hoisting refers to the process where the interpreter appears to move the declaration of classes to the top of their scope before executing the code.

Example

```
try {
  const obj = new ExampleClass();
} catch (e) {
  console.log(e.message);
}


class ExampleClass {
  constructor() {
    console.log('Class initialized');
  }
}
```

Output:

Error: Cannot access 'ExampleClass' before initialization

Strict Mode with Classes

The strict mode in JavaScript was introduced in ES5. By default, the class code is always in the strict mode. We use it to mention that the code should be executed in strict mode. It helps avoid unusual errors.

The strict mode introduced a few changes in JavaScript, which we have discussed below:

- It adds syntax restrictions, prohibiting some declarations that could have been made in the code if the struct mode was not present.

- Classes always follow the use-strict mode, so the code inside classes is always in the strict mode.
- It optimizes the code through rigorous checking.

Example

```
class StrictExample {
  constructor() {
    'use strict';
    let x = 3.14;
    console.log(x);
  }
}

const obj = new StrictExample();
```

Output

3.14

JavaScript Inheritance

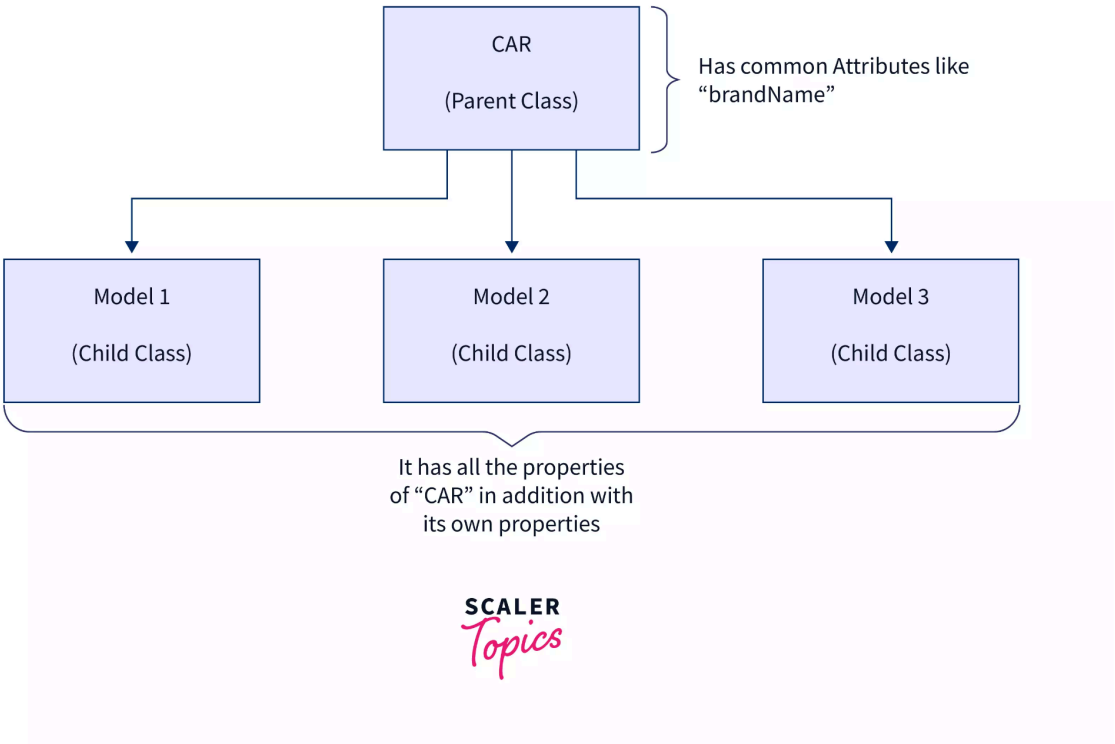
In programming, you may need to add properties and methods of one class to another. For example, you create a class for fruits that includes the same methods and properties for each fruit. So, if you create a separate class for a specific fruit, say Mango, and need to add the same properties and methods to it, you can do so using inheritance.

In general, inheritance means acquiring something. A child can inherit a habit from their parents, or a new species can inherit the property of an older species during evolution. **Inheritance in JavaScript** is a fundamental concept of object-oriented programming. It allows the child class to inherit the methods, properties, and functionalities of the parent class. JavaScript inheritance includes two aspects:

Child Class- The class that acquires the functionality of another class.

Parent Class- The class whose functionality is being acquired or inherited.

Here is an image demonstrating the concept of **JS inheritance**:



What is Inheritance in JavaScript?

JavaScript inheritance is the method that allows objects to inherit the properties and methods from other objects. It means passing down the characteristics from the parent objects to the child object.

Inheritance enables code reusability and structuring of relationships between objects. It creates a hierarchy where a child object can access the functionality and features of the parent object. Using the concept of inheritance, you can create new classes based on existing ones and the child class can reuse the properties and methods of the parent class.

How to Achieve Inheritance in JavaScript

There are three ways to achieve inheritance in JavaScript:

1. Prototypal Inheritance
2. Classical Inheritance
3. Functional Inheritance

1. Prototypal Inheritance

When an object inherits the prototype from other objects through prototypes, it is known as **Prototype inheritance in JavaScript.** Every object has a prototype and objects inherit methods and properties from that prototype.

Example

```
function Parent() {
  this.name = 'Parent';
}

Parent.prototype.greet = function () {
  console.log('Hello from Parent');
};

function Child() {
  Parent.call(this); // Call Parent constructor
  this.name = 'Child';
}

Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;

const child = new Child();
child.greet();
console.log(child.name);
```

Output

Hello from Parent

Child

2. Classical Inheritance

With ECMAScript6 (ES6), JavaScript introduced the class keyword for classical inheritance. It uses the class-based approach. You can achieve classical inheritance in the following ways:

Using the extend keyword

JavaScript ES6 introduced the extend keyword, which is used to perform the inheritance between two classes.

Points to Remember

- We use the extend keyword in class declarations or class expressions.
- Using the prototype-based approach, you can achieve inheritance.
- It creates an IS-A relationship.
- With the extend keyword, you can inherit the behavior and properties of the built-in objects and custom classes.

Example

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

  speak() {
    console.log(`${this.name} makes a sound`);
  }
}

class Dog extends Animal {
```

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

const dog = new Dog('Rex');
dog.speak();
```

Output

Rex barks

Using the super keyword

The super super keyword is used to call all the methods and properties of the parent class of an object. As you make objects from the child class, you must pass the parameters to the parent class using the super() keyword, which will invoke the properties of the parent class. The super keyword used within a child class represents its parent class.

Syntax

```
super(param);
```

We place it inside the constructor of the child class, and the param indicates the parameter for the parent class.

Example

```
class Parent {
  constructor(name) {
    this.name = name;
  }
```

```
}

class Child extends Parent {
  constructor(name, age) {
    super(name);
    this.age = age;
  }

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

const child = new Child('Sultan', 10);
child.details();
```

Output

Name: Sultan, Age: 10

Inheritance in static members

The inheriting static members belong to their own class and not to their instances, as inheritance is applied to the static members of the class. The parent class' static methods and attributes are inherited during JavaScript inheritance.

Example

```
class Parent {
  static sayHello() {
    console.log('Hello from Parent');
  }
```

```
}

class Child extends Parent {}

Child.sayHello();
```

Output

Hello from Parent

3. Functional Inheritance

Objects can inherit methods and properties from other objects using function constructors. Through functions, they create objects and build relationships. The following are the ways to achieve functional inheritance in JavaScript:

Example

```
function Parent(name) {
  this.name = name;
  this.sayHello = function () {
    console.log(`Hello from ${this.name}`);
  };
}

function Child(name, age) {
  Parent.call(this, name); // Call Parent function
  this.age = age;
}

const child = new Child('Bob', 12);
```

```
child.sayHello();

console.log(`Age: ${child.age}`);
```

Output

Hello from Bob

Age: 12

Constructor Overriding

When we perform inheritance without declaring the constructor of the child class, JS creates an empty constructor by default:

```
constructor(...args) {
   super(...args);
}
```

However, we can override the constructor while extending a class. This is known as constructor overriding. Use the super keyword to pass the attributes of the parent class as it invokes the parent constructor.

Example

```
class Parent {
  constructor(name) {
    this.name = name;
  }
}

class Child extends Parent {
  constructor(name, age) {
    super(name); // Call Parent constructor
```

```
    this.age = age;
  }
}

const child = new Child('Tom', 20);
console.log(`Name: ${child.name}, Age: ${child.age}`);
```

Output

Name: Tom, Age: 20

Inheritance from built-in data types

In JavaScrip, we can extend the functionality of the non-primitive built-in data types, such as array, error, map, and set, to create custom classes.

Example

```
class CustomArray extends Array {
  sum() {
    return this.reduce((acc, val) => acc + val, 0);
  }
}

const numbers = new CustomArray(1, 2, 3, 4, 5);
console.log(numbers.sum());
```

Output

15

Uses of Inheritance in JavaScript

Below, we have shared some uses of JavaScript inheritance:

- Inheritance enables us to organize data in a hierarchy.
- It allows you to add your properties in the cold class, so you can override a few methods of the parent class while inheriting useful functionalities and defining required features.
- It makes the code debugging-friendly.
- Class definitions become less complex as they allow you to inherit properties from another class.
- Inheritance promotes code reusability because the child class can inherit all functionalities of the parent class.
- When a functionality is developed, you can inherit the properties instead of reinventing the wheel. This makes the code cleaner and easier to maintain.

Advantages of JavaScript Inheritance

Here are the top benefits of JS inheritance:

Consistency

Inheritance ensures that the derived classes use the same base class and have a standard set of methods and properties, which promotes consistency within software.

Code Reusability

A child class can inherit the functionality of the parent class, enhancing the code reusability.

Scalability

It also improves scalability because you can add new classes to the existing hierarchy without much effort. The new child class can acquire the properties and methods of the base class and integrate new functionalities according to the requirement.

Reduced Code Duplication

As derived classes use the properties and methods defined in the base class, code duplication is reduced significantly.

Code Maintenance

You can maintain the code easily by dividing it into sub-classes.

Faster Development

As you use the existing code and don't need to spend time writing and testing it repeatedly, it improves the development speed. You can focus on other aspects of the derived class to streamline the process.

Easier Debugging

Developers can use the well-structured class hierarchy to find and resolve issues within a program as they know the location of the common functionality. Inheritance also makes debugging and troubleshooting less complex.

Functionality Extension

You can easily add new methods and properties to extend the functionality of the parent class in a child class.

Resource Optimization

When you reuse the code, you can optimize resource utilization, including memory usage and CPU time, which makes the software more efficient.

JavaScript Exception Handling

Errors are common in programming. They can occur due to a simple mistake, incorrect input, etc. An **exception in JavaScript** indicates an abnormal condition that requires special attention and operable techniques. It is an anomalous code that can disrupt code execution, break the normal flow of the code, and result in a poor user experience. Hence, exceptions need immediate and special programming constructs.

Effective exception handling is a process of finding and handling errors that occur during the execution of the program. Exception handling is crucial to build reliable and user-friendly applications in JavaScript.

Let's learn more about errors and **exception handling in JS,** different errors, types of exception handling statements, and more.

What is an Error?

An error is an inaccurate event that occurs during program execution and disrupts it from running smoothly. There are many reasons that can cause errors, such as incorrect user input, slow network connection, etc.

What is Exception Handling in JavaScript?

Exception means unexpected or abnormal activity that can occur during the execution of a program, and handling refers to dealing with such unusual situations.

Therefore, **JavaScript exception handling** is a powerful method for handling abnormal activities or errors in the code to avoid any sudden crashes during program execution. It helps maintain the normal flow of the code. Also, it enables us to track errors so that exceptions can explain the reason for errors and when and where they occurred in the code.

The common reasons for errors in JavaScript can include:

- Slow network connection.
- Wrong user input.
- Dividing a number by 0 can lead to infinity and show an exception during execution.
- The required file doesn't exist in the system.

Examples of Exception Handling in JavaScript

Example- 1: Example of an Error in JavaScript

```
let result = 10 / 0;

console.log(result);
```

Output

Infinity

Example -2: Example of Exception Handling in JavaScript

```
try {
  let value = undefinedVariable;
} catch (error) {
  console.log("Exception caught:", error.message);
}
```

Output

Exception caught: undefinedVariable is not defined

Developers must create an efficient and detailed plan to handle such exceptions to ensure the success of the project. Whenever the code raises an exception during execution, the JavaScript programmer assesses the code to detect the error. The developer will use exception handling statements to resolve the issue based on the error.

Before we dive into different exception-handling statements, let's understand the different types of errors that can occur while executing a JS code.

Types of Errors in JavaScript

In JavaScript, you can encounter three types of errors during the execution of the code:

1. Syntax Error
2. Runtime Error
3. Logical Error

Syntax Error

A syntax error in JS is when a user makes a mistake in the predefined syntax. It can include any error in the sequence of characters that must be written in a specific way. If the interpreter can't interpret the syntax, it raises an error during code compilation. Therefore, it is also known as the compile-time error. The syntax error can stop the program until it is corrected and fixed. Common examples of the syntax errors are:

- Incorrect formatting or indentation.
- Spelling errors.
- Avoiding essential characters, such as not using a semicolon at the end of the statement.

Example

console.log("Hello World";

Output

SyntaxError: missing) after argument list

Runtime Error

A runtime error occurs during the program execution. It is also known as the exception and is detected when you run a program. These errors can crash a program. Hence, we need exception handling to maintain the normal flow of code. Some examples of runtime errors include:

- Dividing a number by 0.

- Integer overflow.
- Segmentation faults.

Example

```
try {
  let num = 10 / 0;
  console.log(num);
} catch (error) {
  console.log("Runtime Error:", error.message);
}
```

Output

Infinity

Logical error

A logical error occurs in the case of a logical mistake in the program that may not generate the desired output and terminate the program abruptly. It is the most challenging error to trace because it occurs on the logical aspect of the code. Hence, you must run manual testing to find these errors. It is like a bug that can cause a program to operate incorrectly. These errors don't throw an error message or exception.

Example

```
let a = 10;
let b = 20;
let average = a + b / 2;
console.log("Average:", average);
```

Output

Average: 20

JavaScript Error Class and Error Object

When an error occurs in JavaScript during the program execution, the code stops abnormally, and JavaScript throws an error object class containing information about the error. We can use error() as a generic constructor for all types of errors, but there are different objects for different errors. Users can also create a user-defined error object using the throw statement. We can use this object as a base for the user-defined exceptions. An error object has two properties:

name- It refers to an object property that stores or returns the name of the error.

message- It describes the error message in a string form.

As the error gives the basic information about the condition, it is a generic constructor. However, various built-in error constructors provide the required details about the issue, making them crucial in exception handling.

Example

```
try {
  throw new Error("Custom error message");
} catch (error) {
  console.log("Error Name:", error.name);
  console.log("Error Message:", error.message);
}
```

Output

Error Name: Error

Error Message: Custom error message

Built-in Error Types in JavaScript

The following are the built-in standard error types that are helpful for handling exceptions in JavaScript:

Error Type	Description
Error	A generic constructor for the error. You can extend the Error object to create custom errors.
EvalError	Occurs in the eval() function when there is an issue with the argument. The eval() function takes a string with the code and evaluates it. Although the EvalError is not used within the current ECMAScript, the object exists for backward compatibility with older versions of the specifications.
SyntaxError	Occurs when there is an error in the syntax, or it breaks the rule. For example, invalid JSON or missing parentheses.
URIError	Occurs when you pass invalid parameters to the decodeURI or encodeURI functions.
InternalError	Occurs when the JavaScript engine gets overwhelmed by various recursions, too many switches are used, or the stack grows beyond the limit.
AggregateError	Aggregates multiple error objects into one, enabling you to handle multiple error objects.
ReferenceError	Occurs when a programmer tries to access variables that are not defined in the current scope.
RangeError	Occurs when the numeric value is out of valid range.
TypeError	Occurs when the variable type is invalid, or you try to operate the wrong data type.

Exception Handling Statements in JavaScript

An exception is any unexpected or abnormal condition that occurs during the runtime. JavaScript uses different statements or a combination of statements to handle these exceptions.

We can handle exceptions using the try-catch-finally statements and the throw operator. We will read about different types of exception handling in JavaScript, but first, let's have a quick look at these basic concepts:

try- The try block contains the main code, which can encounter an exception during program execution. In case of an error, the try block will stop the execution immediately and pass the control to the catch block.

catch- The catch block contains the code, which will run when the error in the corresponding try block is detected. When the code within the try block is executed successfully, the program will ignore the catch block.

finally- The code within the finally block is executed, no matter if there is an error or not.

throw- We use the throw keyword to throw custom or user-defined errors.

Now, let's learn about exception handling statements in JavaScript that you can use in programming.

1. throw Statement

The throw statement is used to raise custom or user-defined errors. It means you can create an error object with the name and message. You can use anything as an error object, including primitive, such as a string or number. Once the exception is identified, you can take the appropriate action.

Syntax

```
throw <error object>
```

Example

```
try {
  let age = 15;
  if (age < 18) {
    throw new Error("You must be 18 or older to proceed.");
  }
```

```
} catch (error) {
  console.log("Error caught:", error.message);
}
```

Output

Error caught: You must be 18 or older to proceed.

2. Try…catch Statements

The try block contains the code that is executed only when the try block throws an exception. If there is an error or exception, the code execution within the try block stops immediately, and the control is passed to the catch block. If the code within the try block is executed successfully, the catch block is ignored without leading to any problems.

Syntax

```
try {
  // code
} catch (err) {
  // Code only executes if try throws an exception
}
```

Example

```
try {
  let result = unknownVariable; // ReferenceError
} catch (error) {
```

```
  console.log("Caught an error:", error.message);
}
```

Output

Caught an error: unknownVariable is not defined

3. try…catch…finally Statements

It is the latest version that has been added to the exception-handling capabilities of JavaScript. The try-catch block is executed based on various conditions. The try block is executed if it doesn't throw any error. The catch block is executed if there is an exception in the try block. However, the finally block is executed whether an error occurs or not. It doesn't care about the try-and-catch blocks.

Syntax

```
try {
   // try to execute the code inside of it
} catch (err) {
   // take care of errors (execution depends on the try block)
} finally {
   // Always executed whether exception occurs or not
}
```

Make sure that the try block is followed by a catch block or a finally block. If the try block has an exception, it is placed in place of err, and the catch block is executed. The finally block is executed unconditionally after try and catch.

Example

```
try {
  let num = 10 / 0;
  console.log(num);
} catch (error) {
```

```
  console.log("Caught an error:", error.message);
} finally {
  console.log("Finally block executed.");
}
```

Output

Infinity

Finally block executed.

JavaScript Custom Errors

The Error() constructor is used to throw an error with a custom message:

```
const customError = new Error(message);
customError.name = "CustomError";
```

Example

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
  }
}

try {
  throw new CustomError("This is a custom error!");
} catch (error) {
  console.log(`${error.name}: ${error.message}`);
```

}

Output

CustomError: This is a custom error!

Using Error Objects in JavaScript

An error object in JS provides the name and message properties. We can use these properties based on the type of error to receive a sophisticated or custom message. The name property represents the idea about the class of the error, while the message property shows the detailed message in a string form. To throw custom exceptions, we can use these properties to find our exception or system-generated exception.

Property	**Description**
name	Used to set or get an error name.
message	Used to set or get an error message

JavaScript try...catch...finally Statement

JavaScript has two types of errors:

Syntax Errors- They occur in the syntax.

Runtime Errors- They occur during the execution of the program.

Errors that occur during the runtime are known as exceptions. JavaScript uses try, catch, and finally for exception handling. The throw statement throws an error, indicating an exceptional condition. Using the try block, we can wrap code that may throw an error, and the catch block is used to handle the error, preventing the program from crashing and enabling efficient error management.

Using the following five statements, we can handle all JavaScript errors:

- **The try…catch…finally statements handle errors without stopping JavaScript.**
- The try statement is used to test a code block to check for errors.
- The catch statement lets you define a code block to handle any error present.

- The throw statement lets you define your own custom errors.
- The finally statement lets you define a code block and run it regardless of the result of the try-catch block.

Let's explore **try, catch, finally in JS** in detail through examples.

JavaScript try...catch Statement

The try…catch statement in JavaScript handles the runtime errors. In the try block, you write the code with the possibility of errors and the one that needs possible error testing. The JavaScript try() statement contains the regular JavaScript code, and it is followed by one catch block or one finally block.

The catch statement in JavaScript handles errors by executing the set of statements written within the block. It allows you to display an error found in the try block. The catch block can have a user-defined exception handler or built-in handler. If the try block has an error, the JavaScript engine executes the code of the catch statement to handle errors. If there are no errors, the catch block is skipped, and the next line of code is executed.

Syntax

```
try {
    // body of try
}
catch(error) {
    // body of catch
}
```

The try block contains the main code, and if it contains any error during execution, it goes to the catch block, which handles errors according to the catch statements. In case of no error, the catch block is skipped, and the code within the try block is executed.

Example

```
try {
let result = 10 / 0;
console.log(result);
 } catch (error) {
```

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

Output

Infinity

JavaScript try...catch...finally Statement

We can also use the try…catch…finally statement to handle exceptions. JavaScript always executes the finally block code, whether the error occurs in the try statement or the code is executed successfully.

The final statement lets you execute the code after completing the try-and-catch statement execution.

Syntax

```
try {
    // try_statements
}
catch(error) {
    // catch_statements
}
finally() {
    // codes that gets executed anyway
}
```

Example

```
try {
  let result = 10 / "a";
  console.log(result);
} catch (error) {
  console.log("Caught an error: ", error.message);
} finally {
  console.log("This will always run, regardless of errors.");
```

```
}
```

Output:

NaN

This will always run,""

JavaScript throw Statement

When there is an error in the code, JavaScript stops execution and throws an error by default, generating an error message. You can also create a custom error and throw an error manually. These custom exceptions can be a JavaScript string, number, object, or boolean. You can use **try catch in JavaScript** to easily control the flow of the program and execute custom error messages.

Syntax

```
throw <error>;
```

Here, <error> can be an error object, number, string, primitive value, or a specific type of an error object. It is better to throw an Error object instead of throwing the primitive value.

Example

```
try {
  let age = -1;
  if (age < 0) {
    throw new Error("Age cannot be negative.");
  }
} catch (error) {
  console.log("Custom Error: ", error.message);
}
```

Output:

Custom Error: Age cannot be negative.

JavaScript try...catch in setTimeout

If an error occurs in the timed code, the try…catch statement will not catch the exception. If we use try…catch with asynchronous JavaScript code, it will not catch the error that the try block throws because JS executes asynchronous code when the main thread code is executed.

Example - 1

```
try {
  setTimeout(() => {
    throw new Error("Error in setTimeout!");
  }, 1000);
} catch (error) {
  console.log("This will not catch the error.");
}

console.log("Code continues...");
```

Output:

Code continues...

(Uncaught Error: Error in setTimeout!)

Example - 2 (To catch errors in `setTimeout`, you need to wrap the code within `setTimeout` in its own try...catch)

```
setTimeout(() => {
  try {
    throw new Error("Error in setTimeout!");
  } catch (error) {
    console.log("Caught the error inside setTimeout:", error.message);
  }
```

}, 1000);

Output:

Caught the error inside setTimeout: Error in setTimeout!

JavaScript Modules

When you work on a big program, it involves multiple lines of code. Organizing these lines and making the code readable can be tricky. Therefore, developers use modules to place codes in separate files based on their functionality. It makes the code easier to maintain and more organized.

Modules in JavaScript are files containing code to perform particular tasks. It can also contain functions, variables, and classes. Let's explore **modules in JS**, importing and exporting modules, and more.

What is a Module in JavaScript?

As your program grows, the number of code lines also grows. Creating a real-time application can involve millions of lines of code. In such cases, you need a single file that can contain these multiple lines of code without hassle and manage them properly. That is where modules come into the picture.

JavaScript modules are files that enable you to divide large code into multiple files to make it easier to maintain. Hence, you don't need to add the code for an entire application in a single file. You can break down the code and create separate files, keeping the code with the same functionality in one file.

A module can be imported from the external files using the import statement. You can import and export features by using type="module" in the <script> tag.

Syntax

Here is the syntax to export and import the **JS modules**:

```
export function func_name(parameters) {

// function code
```

```
}

import {func_name} from filename
```

JS Modules

Importing and Exporting Modules in JavaScript

We can export modules in two ways:

Named Export

You can export multiple values, objects, functions, or variables from a single module using their names.

Example

```
// Default export (module.js)

export default function sayHello() {

  return 'Hello from Default Export!';

}

// Default import (app.js)

import sayHello from './module.js';


console.log(sayHello());
```

Output

Hello from Default Export!

Default Export

In default export, you can export one value from a module. You can import it without using curly brackets. By using the default keyword, you can define the default export. Now, whenever you will import an object not defined in the module, it will return the object exported by default. It will help you avoid the import error.

Syntax

Here is the syntax to use for the default export:

```
export default function func_name(params) {
// function body
}
import random_name from filename;
```

Here, we use the default keyword after the export keyword to export the function by default.

Example

```
// Named export ( module.js )
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
// Renaming imports ( app.js )
import { add as sum, subtract as difference } from './module.js';

console.log('Sum:', sum(5, 3));
console.log('Difference:', difference(5, 3));
```

Output

Sum: 8

Difference: 2

Rename Import and Export

If the code file and module file contain objects of the same name and the developer wants to import the module into the code file, it can raise an error. Hence, the program takes the value from the main file rather than the imported file. You can resolve this conflict by changing the name of the module while exporting or importing it.

Syntax to Rename Imports

The following is the syntax that will allow users to rename the imports.

```
export {
    func1,
    func2
};
// import in the code file
import { func1 as N1, func1 as N2 } from './module.js';
```

Here, we have changed the function name while importing it.

Syntax to Rename Exports

The following is the syntax to rename the exports:

```
export {
    function1 as Name1,
```

```
    function2 as Name2
};
// import in the code file
import { N1, N2 } from './module.js';
```

Here, we have changed the name of function1 to Name1 and the name of function2 to Name2 while exporting the module. The as keyword allows us to change the function name.

Example

```
// ( module.js )
function multiply(a, b) {
  return a * b;
}

function divide(a, b) {
  return a / b;
}

export { multiply as mul, divide as div };

// Importing renamed exports ( app.js )
import { mul, div } from './module.js';

console.log('Multiplication:', mul(5, 3));
console.log('Division:', div(6, 2));
```

Output

Multiplication: 15

Division: 3

Benefits of Using Modules in JavaScript

Here are the reasons to use JavaScript modules:

- It makes the code reusable as you need to define a module once and can reuse it multiple times based on your requirements.
- As different codes with different functionalities are placed in separate files, the code is easier to maintain.

JavaScript JSON

JSON stands for JavaScript Object Notation. It is a text-based format used to organize, store, and transfer data. **JSON in JavaScript** is similar to XML in the way that it structures information. However, JSON is more lightweight and easier to read and write. It is commonly used by web applications to exchange data.

What is JSON in JavaScript?

JavaScript Object Notation (JSON) is a lightweight data interchange format used for storing and transporting data. For humans, **JavaScript JSON** is an easy to read and write text-based format and for machines, it easy to generate and parse. It is built on two structures:

- A collection of key/value pairs separated by a comma. In other languages, it is represented as a record, dictionary, object, keyed list, struct, hash table, or associative array.
- An ordered list of values, which is recognized as an array, list, sequence, or vector in many other languages.

The **JS JSON format** is a readable data collection that can be accessed logically. Its syntax resembles JS object literal syntax.

Key Characteristics of JavaScript JSON

- It is easy to read and write for humans.
- It is used to generate and store data.
- Although it is derived from JavaScript and its syntax resembles object literal syntax in JS, it can be accessed and used with other programming languages.
- It is used to transport data from server to client, server to server, and client to server.
- It is a common format used for interchanging data on the web.

- It can also be used to verify data.

What Makes JSON Helpful?

What if you need to transfer the data and use it somewhere else? You can do so by sending the entire memory of your computer along with the location address. However, it can be a risky method, so not recommended at all.

JSON proves to be a helpful way as it serializes the data and converts it into a format that is easy to read and understand for humans. JSON makes it possible to communicate and transfer data conveniently.

JSON Syntax

Use the following syntax while working with a .json file:

```
{
        "First_Name" :  "value";
         "Last_Name": "value ";
          }
```

When you work with JSON object in .html or .js file, the following syntax will be used:

```
var varName ={
            "First_Name" :  "value";
             "Last_Name":  "value ";
               }
```

JSON Syntax Rules

Here are a few rules that the syntax of JSON in JavaScript follows:

- Data in JSON is separated by commas.

- Data is stored in name/value pairs. The different types of values are:

Boolean- True or False

Object- Associative array of key/value pairs.

Array- Associative array of values.

String- Text-based characters commonly forming a word.

Number- Integer

- Square brackets contain arrays.
- Curly braces contain objects.

Example

```
{
  "First_Name": "John",
  "Last_Name": "Doe"
}
```

JSON Data

Similar to object properties in JavaScript, the JSON data is a collection of key/value pairs. The key and the value are placed in double quotes and separated by a colon.

```
// JSON data
"name": "Ram"
```

JSON Objects

A JSON object refers to a collection of key/value pairs, where keys are strings and values can be numbers, arrays, strings, null, true, or false. JSON objects are written within curly braces {} and contain multiple key/value pairs.

Example

```
{
  "name": "Ram",
  "age": 19
}
```

```
// JSON object
{ "name": "Ram", "age": 19 }
```

JSON Arrays

A JSON array in JavaScript is an ordered collection of values, which can be numbers, arrays, strings, objects, null, true, or false. It is written inside square brackets [].

Example

```
[
  "apple",
  "banana",
  "cherry"
]
```

Accessing JSON Data

To access JSON data, you can use the dot notation.

Syntax

```
variableName.key
```

Example

```
const person = { "name": "Ram", "age": 19 };

console.log(person.name);
```

Output:

Ram

You can also access JSON data using square brackets [].

Example

```
const person = { "name": "Ram", "age": 19 };

console.log(person["age"]);
```

Output:

19

Converting JSON to JavaScript Object

You can convert **JavaScript JSON data** into a JavaScript object using the built-in JSON.parse() method. It parses the JSON string and constructs a value or an object described by the string.

Example

```
const jsonString = '{"name": "Ram", "age": 19}';

const jsonObject = JSON.parse(jsonString);

console.log(jsonObject);
```

Output:

{ name: 'Ram', age: 19 }

Converting JavaScript Object to JSON

To convert a JavaScript object into JSON format, use the built-in JavaScript JSON.stringify() method.

Example

```
const person = { "name": "Ram", "age": 19 };

const jsonString = JSON.stringify(person);

console.log(jsonString);  // Output: '{"name":"Ram","age":19}'
```

Output

'{"name":"Ram","age":19}'

JavaScript JSON vs Objects

Although JSON and JavaScript objects have similar syntax, they are a bit different from each other.

JSON	**JavaScript Object**
We place the key in key/value pairs within double quotes.	We can place the key in key/value pairs without double quotes.
Can be used by other programming languages.	Can be used only in JavaScript.
Can’t contain functions.	Can contain functions.

JavaScript Promise

To many beginners, the Promise in JavaScript may look complex and tricky. However, as you dive deeper and gain a better understanding of JS, you get a clearer idea. When you work with time-consuming code, the **JS promise** makes various tasks easier, such as fetching data from a website.

This chapter covers everything about a promise in JavaScript, including the **JS promise example**, creating a promise, its methods, and more.

What is a Promise in JavaScript?

A **promise in JS** is an efficient way to handle asynchronous operations and helps us determine whether these operations were completed successfully. It is like a container, ensuring to produce a single value in the future. Similar to a promise where one person gives an assurance that a specific thing will happen, a JavaScript promise is also a way of saying that you may not have a value right now, but you are assured to have it in the future.

A promise enables you to continue working with the code while you wait for something else to finish, such as loading data from the server. Once the required data is ready, the promise is fulfilled and delivered to you.

JavaScript promise can have one of the following states:

Pending- It is the initial state, which indicates that the promise is neither rejected nor fulfilled.

Fulfilled- Once the operation is completed successfully, the promise is also fulfilled.

Rejected- If there is an error, the promise fails and is rejected.

For example, suppose you request data from the server through a promise. Initially, it will enter the pending state and then move to the fulfilled state once data arrives.

If there is an error, it will enter the rejected state.

Hence, we can conclude that a promise refers to the completion of an asynchronous operation with results. The promise can be fulfilled successfully or can fail, but it is eventually completed. We use then() with promise, which is executed once the promise is completed.

Need for JavaScript Promise

When we use events or callback functions to handle data, they offer restricted scope because events are unable to handle and operate asynchronous operations. Therefore, we need a **JavaScript promise** that provides a better and more straightforward approach for handling asynchronous operations.

How Does a Promise Work in JavaScript?

Now that you know why we need promises, it's time to understand **how promises work in JavaScript.** As we mentioned, a promise has three main stages:

- The pending stage is the first phase, where the promise waits for something to finish. For example, waiting to load data.
- The second stage is the fulfilled stage, where the promise has been completed. For example, data is made available.
- The rejected stage shows that the promise has failed due to a problem, such as server issues.

As you create a promise, you also write the code to tell if it is fulfilled or rejected. If it is fulfilled, it means the promise is successful; else, it is failed.

JavaScript Promise Syntax

```
let promise = new Promise(function(resolve, reject){
    //do something
});
```

Parameters

- We use the promise constructor that takes only one argument, a callback function.
- The callback function accepts two arguments: resolve and reject.
- Operations are performed within the callback function. The call is resolved when everything happens successfully.
- If the operation fails, the call is rejected.

Example

```
myPromise
  .then(function(result) {
    console.log(result);
  })
  .catch(function(error) {
```

```
  console.log(error);
});
```

Creating a Promise in JS

You can create a promise by using the Promise() constructor with a function as an argument. The function takes two arguments: resolve(), which is called when the promise returns successfully, and reject(), which is called when there is an error.

Example

```
let myPromise = new Promise(function(resolve, reject) {
  let success = true;
  if (success) {
    resolve("Data loaded successfully");
  } else {
    reject("There is an error in data loading");
  }
});
```

Using a Promise in JavaScript

After creating a promise, you can use it to accomplish something when the promise is fulfilled or rejected. You can use the then-and-catch methods for the purpose.

```
myPromise.then(function(message) {
// runs if the promise is fulfilled
  console.log(message);
```

```
}).catch(function(error) {
// runs if the promise is rejected
  console.log(error);
});
```

If the promise is fulfilled, the then method is called, which takes the function as an argument and runs when the promise is met. It will show “data loaded successfully” in the console.

If the promise is rejected, the catch method is called, which takes a function as an argument and runs in case of an error. It will show an error message “There is an error in data loading.”

Example

```
myPromise.then(function(message) {
  console.log(message);
}).catch(function(error) {
  // runs if the promise is rejected
  console.log(error);
});
```

Output:

Data loaded successfully

There is an error in data loading

Chaining Promises

In JavaScript programming, you may need to perform one task after the other. For example, loading data, processing it, and displaying it. Promises allow you to achieve this through the chaining then methods. JavaScript promise enables you to handle multiple asynchronous tasks through promise chaining. You can use then(), catch(), and finally() methods to perform an operation after a promise is resolved.

Example

```
myPromise
```

```
 .then(function(message) {
   console.log(message);
   return "Processing data";
 })
 .then(function(processMessage) {
   console.log(processMessage);
 })
 .catch(function(error) {
   console.log(error);
 });
```

Output:

Data loaded successfully

Processing data

Promise then() Method

The promise then() method is invoked when the promise is resolved or rejected. It is a carrier that takes data and executes it successfully.

Syntax

```
.then(function(result){
        //handle success
}, function(error){
        //handle error
})
```

This method takes two parameters. The first function is executed when the promise is resolved, while the second function is executed if the promise is failed or an error occurs. The second one is optional.

Example

```
myPromise.then(function(result) {
```

```
  console.log(result);
}, function(error) {
  console.log(error);
});
```

Output:

Data loaded successfully

There is an error in data loading

Promise catch() Method

The promise catch() method is invoked when the promise is rejected, or an error occurs during execution. It can be used as an Error Handler when there is a possibility of an error.

Syntax

```
.catch(function(error){
      //handle error
   })
```

It takes only one parameter, a function that handles errors or rejections. The .catch() method is a shorthand for .then.

Example

```
myPromise.catch(function(error) {
  console.log(error);
});
```

Output:

There is an error in data loading

JavaScript finally() Method

You can use the finally() method with the JS promise. It is executed when the promise is rejected or resolved.

Example

```
myPromise
  .then(function(message) {
    console.log(message);
  })
  .catch(function(error) {
    console.log(error);
  })
  .finally(function() {
    console.log("Operation completed");
  });
```

Output:

Data loaded successfully

There is an error in data loading

Operation completed

JavaScript Promise Versus Callback

Promises and callback functions are similar in that we can use them both to handle asynchronous tasks. However, they share a few differences, which we have discussed below:

JavaScript Promise

- Its syntax is easy to read and user-friendly.
- It makes error handling easier to manage.

Example

```
let promise = new Promise(function(resolve, reject) {
```

```
  setTimeout(() => resolve("Task Completed"), 2000);
});

promise.then(function(result) {
  console.log(result);
}).catch(function(error) {
  console.log(error);
});
```

Output:

Task Completed

JavaScript Callback

- Its syntax is comparatively difficult to understand.
- It makes error handling challenging.

Example

```
function fetchData(callback) {
  setTimeout(() => callback("Task Completed"), 2000);
}

fetchData(function(result) {
  console.log(result);
});
```

Output:

Task Completed

JavaScript Promise Methods

Promise methods in JavaScript are actions performed on objects. These are of two types:

Static Methods- They are called using the array class.

Instance Methods- They are called on an instance of a promise.

Static Methods	**Description**
all()	Handles all the asynchronous operations that take promises as input.
allSettled()	Returns a promise when all promises are settled, i.e., either fulfilled or rejected.
any()	Returns a single promise when any of the promises is fulfilled.
race()	Returns a single Promise when any of the promises is settled.
reject()	Returns a Promise object rejected with a given reason.
resolve()	Returns a Promise object resolved with a given value.
catch()	The function is called when the promise is rejected.

then()	The function is called when the promise is rejected or fulfilled.
finally()	Returns a promise when it is settled.

JavaScript Generators

Generators in JavaScript is a new way to work with iterators and functions. You can use generators to stop the execution of a function from anywhere within the function and to continue code execution from a halted place.

Here is a chapter exploring JS generators, the syntax for generator creation, and more.

What is a Generator Function in JavaScript?

A regular function only returns a single value or nothing. However, generators can return multiple values one after another if required. **JavaScript generators** work with iterables, so we can create data streams seamlessly.

The generator function uses the yield keyword to generate values. It pauses the execution and sends the value to the caller. While retaining the state, execution resumes after the yield and continues after the last yield.

Syntax

```
// An example of generator function

function* gen(){

    yield 1;

    yield 2;

    ...

    ...

}
```

Generator Object

The **JS generator function** returns a generator object, which is used by calling the next method of the generator object or by using it in a for…of loop. The generator object conforms to the iterable protocol and iterator protocol.

Example

```
function* fruitGenerator() {

  yield 'Apple';

  yield 'Banana';

  yield 'Cherry';

}



const fruits = fruitGenerator();




console.log(fruits.next());

console.log(fruits.next());




for (const fruit of fruits) {

  console.log(fruit);

}
```

Output

```
{ value: 'Apple', done: false }
```

```
{ value: 'Banana', done: false }

Cherry
```

Creating JavaScript Generators

You can create a generator by first defining a generator function using the function* symbol.

Syntax

```
// define a generator function

function* generator_function() {

   ... .. ...

}


// create a generator

const generator_obj = generator_function();
```

We use * to represent the generator function. We can create them by using either function* generatorFunc(){...} or function *generatorFunc(){...}.

Example 1-

```
function* numberGenerator() {

  yield 1;

  yield 2;

  yield 3;

}


const generator = numberGenerator();
```

```
console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
```

Output

```
{ value: 1, done: false }
{ value: 2, done: false }
{ value: 3, done: false }
{ value: undefined, done: true }
```

Example 2-

```
function* rangeGenerator(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

const generator = rangeGenerator(1, 5);

for (const value of generator) {
  console.log(value);
}
```

Output

```
1
2
```

3

4

5

Example 3-

```
function* infiniteCounter() {
  let count = 1;
  while (true) {
    yield count++;
  }
}

const counter = infiniteCounter();

console.log(counter.next().value);
console.log(counter.next().value);
console.log(counter.next().value);
console.log(counter.next().value);
```

Output

1

2

3

4

Using yield to Pause Execution

By using the yield keyword, JavaScript allows you to pause the generator function's execution without having to execute the entire function body.

Example

```
function* generatorFunc() {

  console.log('Before yield');

  yield 'First Yield';

  console.log('After first yield');

  yield 'Second Yield';

}



const generator = generatorFunc();

console.log(generator.next());

console.log(generator.next());

console.log(generator.next());
```

Output

```
Before yield

{ value: 'First Yield', done: false }

After first yield

{ value: 'Second Yield', done: false }

{ value: undefined, done: true }
```

yield- It will pause the execution of the generator function and return the expression value written after the yield keyword.

yield*- It iterates over the operand, returning each value until it is true.

Example

```
// Generator function demonstrating yield

function* numberGenerator() {

  console.log('Start');

  yield 1;

  console.log('Paused after 1');

  yield 2;

  console.log('Paused after 2');

  yield 3;

  console.log('End');

}


// Create generator object

const generator = numberGenerator();


console.log(generator.next());

console.log(generator.next());

console.log(generator.next());

console.log(generator.next());
```

Output

```
Start

{ value: 1, done: false }

Paused after 1

{ value: 2, done: false }

Paused after 2

{ value: 3, done: false }

End

{ value: undefined, done: true }
```

Example- How to create iterable

```
// Generator using yield* to delegate to another generator
function* innerGenerator() {
  yield 'Inner 1';
  yield 'Inner 2';
}

function* outerGenerator() {
  yield 'Outer 1';
  yield* innerGenerator();
  yield 'Outer 2';
}

// Create generator object
const generatorObj = outerGenerator();

for (const value of generatorObj) {
```

```
  console.log(value);
}
```

Output

Outer 1

Inner 1

Inner 2

Outer 2

Working of Multiple yield Statements

Unlink the return statement, the yield expression doesn't terminate the program but returns the value. Hence, we can continue the code execution from the last yielded position.

Example

```
// Generator with multiple yields and state persistence
function* counter() {
  let count = 0;
  while (count < 3) {
    yield count++;
  }
  return 'Done';
}


// Create generator object
```

```
const countGen = counter();


console.log(countGen.next());
console.log(countGen.next());
console.log(countGen.next());
console.log(countGen.next());
```

Output

```
{ value: 0, done: false }
{ value: 1, done: false }
{ value: 2, done: false }
{ value: 'Done', done: true }
```

Passing Arguments to Generator Functions

JavaScript allows you to pass arguments to a generator function.

Example

```
function* generatorFunc() {
  const value = yield 'Enter a value:';
  console.log('You entered:', value);
}


const generator = generatorFunc();
console.log(generator.next());
```

```
console.log(generator.next(42));
```

Output

```
{ value: 'Enter a value:', done: false }

You entered: 42

{ value: undefined, done: true }
```

Using Generators for Implementing Iterables

The JS generator function offers an easier way to implement iterators. To create an iterator manually, create an iterator using the next() method and save the state.

Example

```
function* iterableGenerator() {
  yield 'A';
  yield 'B';
  yield 'C';
}

for (const value of iterableGenerator()) {
  console.log(value);
}
```

Output

```
A
```

B

C

Using the JavaScript Generator Function With return

We can also use the return statement with the generator function. It returns a value while terminating the function.

Example- Return from a generator function

```
function* generatorFunc() {
  yield 1;
  return 2;
  yield 3;
}

const generator = generatorFunc();
console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
```

Output

```
{ value: 1, done: false }
{ value: 2, done: true }
{ value: undefined, done: true }
```

JavaScript Generator Throw Method

Use the throw() method to explicitly throw an error on the generator function. It will terminate the function.

Example- Throw an error from the generator

```
function* generatorFunc() {
  try {
    yield 1;
    yield 2;
  } catch (e) {
    console.log('Caught error:', e.message);
  }
}


const generator = generatorFunc();
console.log(generator.next());
console.log(generator.throw(new Error('Something went wrong')));
console.log(generator.next());
```

Output

```
{ value: 1, done: false }
Caught error: Something went wrong
{ value: undefined, done: true }
```

Generator Methods

Here is a table explaining three generator methods and their descriptions.

Method	Description
next()	Resumes the execution of the generator function and returns a value of yield.
throw()	Throws an error and terminates the generator.
return()	Returns a value while terminating the generator.

JS return Vs yield Keyword

Let’s see the difference between the return and yield keywords used in JavaScript generators.

return Keyword	yield Keyword
Available in normal functions and generator functions.	Available only in generator functions.
Returns a value while terminating the function.	Returns a value and pauses the function but doesn’t terminate the function.

Uses of JavaScript Generators

- They provide an easier method to implement iterators.
- They are memory efficient.
- They enable us to write cleaner code.
- They execute code only when required.

JavaScript RegEx

Regular **Ex**pressions (RegEx) in JavaScript are powerful programming tools that describe a sequence of characters that define a search pattern. They enable efficient text processing. Developers must understand how to solve problems using **JS Regular Expressions** to improve productivity.

This chapter of our JavaScript tutorial will help you learn the fundamentals of regular expressions, how to create RgEx, regular expression modifiers, metacharacters, and more.

What is RegEx in JavaScript?

Regular Expressions in JavaScript are a sequence of characters used to define a search pattern. They allow us to describe, match, or parse text. We can use them to find and replace text, verify that the input data adheres to the required format, etc.

JS RegEx helps us perform versatile and precise operations, including matching, searching, and replacing text according to the defined patterns, be it complex rules or simple characters.

JavaScript RegEx Syntax

```
/pattern/modifiers;
```

JS RegEx Example

```
const text = 'Order ID: 12345';

const result = text.replace(/\d/g, '#');


console.log(result);
```

Output

Order ID: #####

Creating a RegEx in JavaScript

We can create a JS regular expression in two ways:

Using Regular Expression Literal

It consists of a pattern enclosed between slashes /. We can write it with or without a flag. The syntax is:

Syntax

```
const regExpLiteral = /abc/;          // Without flags


const regExpLiteralWithFlags = /abc/; // With flags
```

Here, abc is a regular expression, and forward slashes /…/ show that we are creating a regular expression.

Example

const regExpLiteral = /hello/;

const regExpLiteralWithFlags = /hello/i;

console.log(regExpLiteral.test('hello world'));

console.log(regExpLiteralWithFlags.test('Hello world'));

Output

true

true

Using the RegExp() Constructor Function

Another method to create a regular expression in JS is by calling the RegExp() constructor function. Here is the syntax.

Syntax

```
const regularExp = new RegExp('abc');
```

Example

const regularExp = new RegExp('abc');

console.log(regularExp.test('123abc456'));

console.log(regularExp.test('123def456'));

Output

true

false

Which Method to Choose?

When you know the regular expression pattern while writing the code, use the RegEx literal. However, if you want to create a regular expression dynamically, use the RgExp() constructor function. It will allow you to write the pattern using the template literal, which is not possible with the RegEx literal syntax.

Using String Methods

We commonly use JavaScript RegEx with two string methods- search() and replace(). Let's understand each of them:

- The search() method uses an expression to search for a match and return its position.

Example

const str = "Hello, welcome to JavaScript regex!";

const index = str.search(/JavaScript/);

console.log(index);

Output:

18

- The replace() method replaces the modified string where the pattern is replaced.

Example

```
const str = "I love JavaScript!";
const newStr = str.replace(/JavaScript/, "RegEx");

console.log(newStr);
```

Output:

"I love RegEx!"

Regular Expression Modifiers

They are used for multiline searches that can be set to case-insensitive matching.

Expressions	**Descriptions**
g	Enables the global search, so it finds all the matches and doesn’t return after the first match.

i	Used for case-insensitive matching, so there is no difference between A and a.
d	Used for a case-sensitive search and finds the beginning and end match.
m	FInds multiline matching and affects the behavior of start ^ and end $.

Regular Expression Brackets

These enable group searches, so you can find characters within a specified range.

Expressions	**Description**
[abc]	Finds matches for any of the characters inside the brackets
[^abc]	Specifies matches for any character not inside the brackets
[0-9]	Finds any of the digits between the brackets. The digitals within brackets can be a number or a span of numbers from 0 to 9.

[^0-9]	Finds any non-digit character.
(x | y)	Finds any of the alternatives specified between x or y separated using |. The alternatives can be any character.

Regular Expression Metacharacters

They are the characters with a special meaning.

Metacharacter	**Description**
\.	Search single characters, except newline or other line terminators.
\w	Finds the word character that can be a character from a to z, A to Z, and 0 to 9.
\W	Matches non-word characters.
\d	Finds a digit from 0 to 9.

\D	Searches for non-digit characters, which means all the characters except digits.
\r	Searches for the carriage return characters.
\t	Finds the horizontal tab characters (tabulators).
\b	Searches for a match at the beginning or at the end of a word.
\B	Searches for a match that is not at the beginning or end of a word.
\s	Finds a whitespace character, which includes a new line character, space character, form feed character, tab character, and vertical tab character.
\S	Finds the non-whitespace characters.
\0	Finds the NULL character.
\n	Finds the newline character.

\f	Matches the form feed character.
\v	Finds the vertical tab characters.
\xxx	Matches the Latin character by an octal number (xxx).
\uxxxx	Matches the Unicode character specified by the hexadecimal number xxxxx.

Regular Expression Quantifiers

They are used to define quantities occurrence.

Quantifier	**Description**
n+	Matches a string that contains at least one n.
n*	Matches a string that contains zero or more occurrences of n.

n?	Matches a string that contains zero or one occurrence of n.
n{X}	Matches a string that contains a sequence of n, X times. Here, X is a number.
n{X, Y}	Matches a string that contains a sequence of n, X to Y times. Here, X and Y are numbers.
n{X,}	Matches a string that contains a sequence of n, at least X times.
n$	Matches a string with n at the end of it.
^n	Matches a string that contains m at the beginning of it.
?=n	Matches a string that is followed by a specific string n.
?!n	Matches a string that is not followed by a specific string n.

Regular Expression Object Properties

The constructor property returns the constructor function of a regular expression in JavaScript.

Property	Description
constructor	Returns the function that created the RegExp prototype.
global	Finds whether the **g** modifier is set or not. If the g modifier is set, it returns true; else, it returns false.
ignoreCase	Finds whether the i modifier is set or not. If it is set, it returns true; else, it returns false.
lastindex	Specifies the index at which the next match will start.
multiline	Specifies whether the **m** modifier is set or not. If the m modifier is set, it returns true; else, it returns false.
source	Returns the text of the RegExp pattern.

Regular Expression Object Methods

Method	Description
compile()	Compiles the regular expression during the script execution.

exec()	Tests for the match in a string.
test()	Tests for a match in a string.
toString()	Returns the string value of the RegEx.

JavaScript this

The **this keyword in JavaScript** is the object that is currently executing a method. Its value is based on how a function is called and it represents the context of the function, which allows access to the methods and properties of the containing object.

What is this in JavaScript?

JavaScript this keyword contains the reference to the object and represents the context of the current code or function. In methods, this refers to the object, whereas in the global scope, it refers to the global object. We use this to access the methods and properties of the current object.

When used in the function, **this in JavaScript** refers to the object with which the function is called. As functions are also objects in JavaScript, we can use this with functions. As this is a keyword and not a variable, we can’t change its value.

Which object does the 'this' refer to?

This **this keyword in JS** refers to different objects based on how it is used:

- When used in an object method, this refers to an object.
- In the function (default), this refers to the global object.
- In the function (strict mode), this is undefined.
- When used alone, this refers to the global object.
- When used in an event, this refers to the element that received the event.
- In methods like apply(), call(), and bind(), this can refer to any object.

JavaScript this in the Global Scope

When used in the global scope, this refers to the global object. Use this in the global scope to access global variables.

Example

```
console.log(this);

// In browsers, it refers to the `window` object.
```

JavaScript this in a Method

When we use this in an object method, it refers to the object itself, which allows access to its methods and properties within the scope of the method. It allows interaction with the data and behavior of the object so we can access and manipulate its state.

Example

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

obj.greet();
```

Output:

Alice

JavaScript this in a Child Function of an Object Method

When we define a function within the object method and use this in the function, the this keyword in JS refers to the global object and not just an object.

Example

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

obj.greet();
```

JavaScript this in a Function (Default)

When used in the function, the this keyword refers to the global scope or global (window) object. In a function, the global object is the default binding for this.

Syntax

```
function exampleFunction() {
  console.log(this); // Refers to the current execution context
}
```

Example

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

showThis();
```

JavaScript this in a Function (Strict Mode)

When the this keyword in JavaScript is used in a function in the strict mode, it becomes undefined. The strict mode doesn't allow default binding.

Example

```
'use strict';

function showThis() {

  console.log(this);

}


showThis();
```

Output:

undefined

Using JavaScript this Alone

When used alone in JavaScript outside of any specific context, this refers to the global scope because it is running in the global scope. The behavior of this can vary based on whether the code is running in strict mode or not.

Example

```
console.log(this);
```

JavaScript this in the Constructor Function

We use constructor functions in JavaScript to create objects. When a function is used as a constructor function, this indicates the object within which it is used.

Example

```
function Person(name) {

  this.name = name;
```

```
}

const person = new Person('Alice');
console.log(person.name);
```

Output:

Alice

JavaScript this in an Arrow Function

Arrow functions don't have their own this, so when we use the this keyword in an arrow function, it refers to the parent object. If we define the arrow function within another function, this refers to the global object.

Example

```
const obj = {
  name: 'Alice',
  greet: () => {
    console.log(this);
  },
};

obj.greet();
```

JavaScript this in the Inner Function

When we use this in an inner function, it refers to the global object.

Example

```
function outer() {
  function inner() {
    console.log(this);
  }
```

```
  inner();
}

outer();
```

JavaScript this in Event Handlers

In event handlers, this refers to the HTML element that received the event.

Example

```
const button = document.createElement('button');
button.innerText = 'Click me';
button.onclick = function() {
  console.log(this);
};
document.body.appendChild(button);
```

Implicit Binding in JavaScript

When a function is called as an object method, the this keyword refers to the calling object.

Example

```
const obj = {
  name: 'Sultan',
  greet() {
    console.log(this.name);
  },
};

obj.greet();
```

Output:

Sultan

Explicit Binding in JavaScript

In JavaScript, we use methods like apply(), call(), and bind() for explicit binding. It allows us to borrow the method for a specific object. By using these binding methods, we can explicitly define the context of the this keyword. The reference of this changes based on the object called using the given methods.

The call() and apply() methods are predefined and can be used to call an object method with another object as an argument.

Example

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

const obj = { name: 'Harun' };
greet.apply(obj);
```

Output:

Harun

Example-Using the bind() method

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

const obj = { name: 'Sultan' };
```

```
const boundGreet = greet.bind(obj);
boundGreet();
```

Sultan

Example-Using the call() method

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

const obj = { name: 'Sultan' };
greet.call(obj);
```

Output:

Sultan

Function Borrowing

If we use the bind() keyword, an object can borrow a method from another object.

Example

```
const obj1 = {
  name: 'Harun',
  greet() {
    console.log(this.name);
  },
};
```

const obj2 = { name: 'Bob' };

obj1.greet.call(obj2);

Output:

Bob

Precedence Order of the this Keyword

Here is the precedence order we use to determine the context of the this keyword in JavaScript:

1. bind() method

2. call and apply() methods

3. Object method

4. Global scope

JavaScript Events

Any interaction of JavaScript with HTML is handled through events. They can occur when a user or a browser manipulates a page. An **event in JavaScript** is an occurrence or action happening in the browser. Events can be triggered by different user interactions or the browser.

Common **JS events** include page loads, keyboard presses, form submissions, mouse clicks, window resizing, and window closing. Event handlers refer to JavaScript functions that respond to these events so developers can create interactive web applications.

In this chapter, we will help you understand JavaScript events in detail, including event handlers, event attributes, and more.

What are Events in JavaScript?

Events in JS are actions that happen when a user interacts through the browser with the help of an input field or other interactive elements available in the browser. Events are a part of the Document Object Model (DOM) Level 3 and each HTML element includes a set of events that can trigger the code in JS.

JavaScript developers use these events to execute coded responses, causing messages to be displayed to users, buttons to close windows, data to be validated, and more.

A **JavaScript event** helps us create interactive and dynamic web applications and web pages. We can use it through the **JavaScript eventListeners**.

Syntax

```
<HTML-element Event-Type = "Action to be performed">
```

Example

<button onclick="alert('Button Clicked!')">Click Me</button>

Common JavaScript Events

Event Attribute	**Description**
onclick	Triggers when a user clicks an HTML element.
onmouseover	Occurs when a user moves the mouse pointer over an element.
onmouseout	Occurs when the mouse pointer moves away from an element.

onkeydown	Fires when a key is pressed down.
onkeyup	Occurs when a key is released.
onchange	Occurs when the value of an HTML element changes.
onload	Occurs when the browser finishes loading the page.
onsubmit	Occurs when a form is submitted.
onfocus	Triggers when an element gets focus.
onblur	Triggers when an element loses focus.

JavaScript Event Handlers

Event handlers in JavaScript are functions executed in response to a specific event that occurrs in the browser. To attach events to HTML elements, we use event attributes like onmouseover, onclick, etc. We can also add events dynamically using the addEventListener() method.

They are used to handle and verify user actions, inputs, and browser actions.

- Actions taken when a user clicks a button.
- Actions taken every time a page loads.
- Content to be verified when a user inputs data.

- Actions to be performed when a page is closed.

There are various methods used to allow JavaScript to work with events, including:

- HTML vent attributes are used to call JavaScript functions.
- Stop events from being sent or being handled.
- HTML event attributes are used to execute JS code directly.
- Assign your event handler functions to HTML elements.

Syntax

Here is the syntax for using event handlers with HTML elements.

```
<div eventHandler = "JavaScript_code"> </div>
```

Here, replace the ‘eventHandler’ with the actual event handler, like onemouseover and onclick. The Javascript_code is used to execute the function or run JS inline.

Example

```
<button id="btn">Click Me</button>

<script>

  document.getElementById('btn').onclick = function() {

    alert('Button Clicked via Event Handler!');

  };

</script>
```

JavaScript Custom Events

As we have learned, events in JS can be used in almost every web application. For example, using the onclick event, code can be executed when a user clicks on something. Javascript has various built-in events that we can use. However, what if we want to customize an event? Hence, we create custom events.

Creating a Custom Event in JavaScript

You can create a custom event by using the Event constructor or CustomEvent interface. The former creates an Event, while the latter creates an event with added functionality.

Using the Event Constructor

Here are the steps to follow to create an event using a new Event:

- Create an event with the Event constructor.
- Listen to the event using the addEventListner() method.
- Dispatch or trigger the event using element.dispatchEvent(eventName) method.
- You now have a newly created custom event.

Syntax

```
// To assign event
const startEvent = new Event("start");
// To trigger the event Listener
document.addEventListener("start", () => {
    console.log("The start event was triggered")
});
// To trigger the Event
document.dispatchEvent(startEvent);
```

Example

```
const startEvent = new Event("start");

document.addEventListener("start", () => {
    console.log("The start event was triggered");
});

document.dispatchEvent(startEvent);
```

Using the CustomEvent Interface

When we create a custom event using the CustomEvent interface, it can send custom data, providing an edge over the previous method. Here are the steps to create a new CustomEvent.

- Create an event using the CustomEvent constructor.
- The method takes two arguments- the event name and an object constraining the data.
- The property name within the name of the object should be detailed else it will not work.
- Create a listener for the event.
- Trigger or dispatch the event.
- Now you have a custom event containing data.

Syntax

```
// To assign event
const event = new CustomEvent("start", {
    detail: {
      platform : "GeeksforGeeks"
    }
});
// To trigger the event Listener
document.addEventListener('start', (e)=>{
    console.log(
      `start event triggered on platform :
      ${e.detail.platform}`
      );
});
```

```
// To trigger the Event
document.dispatchEvent(event);
```

Example

```
const customEvent = new CustomEvent("start", {
   detail: { platform: "WsCubetech" }
});

document.addEventListener('start', (e) => {
   console.log(`Start event triggered on platform: ${e.detail.platform}`);
});

document.dispatchEvent(customEvent);
```

JavaScript Event Attributes

Windows Events

These events occur globally, so they are associated with web pages and windows.

Event	Description
onload	Triggers when the browser finishes loading other resources on the web page. It is mostly used to load the script file on the web page.
onunload	Occurs when the visitor leaves the current webpage, and the browser unloads it.
onresize	Occurs when a user resizes the browser window.

onstorage	Occurs when a web storage area is updated.
onerror	Occurs when an error occurs.

Example

```
<body onload="alert('Page Loaded Successfully!')">
  <h1>Welcome to the Page</h1>
</body>
```

Mouse Events

These are the events triggered when there is any interaction due to the mouse.

Attribute	**Description**
onclick	Occur when a mouse clicks on the element.
ondblclick	Triggers when a mouse double-clicks on the element.
onmousedown	Occurs when a mouse pointer is pressed down on an element.
onmousemove	Triggers when the pointer of a mouse moves while it is over an element.

onmouseout	Fires when a user moves the mouse pointer out of an element.
onmouseover	Occurs when the mouse pointer hovers over a specific element that is under observation.
onmouseup	Triggers when a mouse button is released over an element.
onmousewheel	Uses the onwheel attribute.
onwheel	Fires when the mouse pointer rolls up or down over an element.

Example

```
<button onclick="console.log('Mouse Clicked!')">Click Me</button>
```

Keyboard Events

When the keys present in the keyboard move, we call them keyboard events, such as keypress, keydown and keyup. These different events may look the same, but their behavior is distinct from each other.

Attribute	**Description**
onkeydown	Occurs when a user presses a key on the keyboard. It doesn’t need any character production.

onkeypress	Fires when a user presses a key on the keyboard that generates a character value. As the user presses the key, it detects the keypress event.
onkeyup	Triggers after the keydown when a user releases the pressed key.

Example

```
<input type="text" onkeydown="console.log('Key Pressed!')" />
```

Form Events

As the name suggests, these events can be used in forms.

Attribute	**Description**
onblur	Occurs when the element loses focus.
onchange	Occurs when the value of an element is changed. It tracks the changes a user makes on the HTML page. When the event is used directly in JavaScript, we must use change in addEventListener, but if used directly into the HTML page, we must use onchange as the attribute in the element.
oncontextmenu	Occurs when a context menu is triggered.
onfocus	Triggers when a user focuses on an element. Once the element gets the focus, we can make any changes.

oninput	Occurs when an element gets user input.
oninvalid	Triggers when an element is invalid.
onreset	Occurs when a user clicks the Reset button in a form.
onsearch	Occurs when a user writes something in a search field.
onselect	Triggers after a user selects some text in an element.
onsubmit	Fires when a user submits a form.

Example

```
<form onsubmit="alert('Form Submitted!'); return false;">
  <input type="text" required />
  <button type="submit">Submit</button>
</form>
```

JavaScript Getter and Setter

JavaScript has two types of object properties- **Data Properties** and **Accessor Properties.** We have used data properties in previous chapters. Accessor properties in JavaScript are methods that get or set the value of an object using two keywords:

get- Used to define a getter method to get the value of the property.

set- Used to define a setter method to set the value of the property.

The getter is used to read the values of properties, whereas the setter is used to write values to properties. When we define a getter or setter, we define a method associated with a specific property of an object. **Getter and setter in JS** provide a way to encapsulate the

implementation details of an object property and allow external code to access and alter its value.

This chapter will explore how to define and use getters and setters in JavaScript.

JavaScript Getter

Getters are functions in JavaScript used to access the properties of an object. Although they are similar to methods, they are not called on the instance of an object but are called on the object itself. We use getters to access values from an object that are not directly accessible.

We define getters using the get keyword followed by a function, which takes no arguments and returns a value. The returned value can be any valid JS value, such as arrays and objects. The value of the object property is not calculated until it is accessed.

Syntax

```
{get prop() { … } }
{get [expression]() { … } }
```

prop- Name of the property used to bind to the given function.

expression- An ES6 addition used for a computed property name to bind to the given function.

Example

```
const person = {
  firstName: "Anil",
  lastName: "Kumar",
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

console.log(person.fullName);
```

Output:

Anil Kumar

JavaScript Setter

The setter method in JavaScript is used to set the value of a property of an object. This special method is used to change the value of an object. Its syntax binds an object property to a function that is called when a user tries to set the property. We define a setter using the set keyword, and the method takes one argument, i.e., the value to which we want to set the property.

The set is executed when a specific property is changed. Setters are used in correspondence with getters to create a type of pseudo-property.

Syntax

```
{set prop(val) { . . . }}

{set [expression](val) { . . . }}
```

prop- Name of the property to bind to the given function.

val- Used as an alias for the variable holding the value trying to be assigned to prop.

expression- Used for a computed property name to bind to the given function.

Example

```
const person = {
  firstName: "Ved",
  lastName: "Prakash",
  set fullName(name) {
    const parts = name.split(" ");
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
};

person.fullName = "Vedangi Mishra";
```

console.log(person.firstName);

console.log(person.lastName);

Output:

Vedangi

Mishra

Data Quality While Using JavaScript Getters and Setters

We use getters and setters to secure better-quality data. These methods provide central control over how a specific field is initialized and provided to the user, making it easier to verify and debug.

Object.defineProperty() in JavaScript

In JavaScript, the Object.defineProperty() method is used to add getters and setters and define a new property directly on an object. This method returns an object and is also used to change the flags of an attribute.

Syntax

```
Object.defineProperty(obj, prop, descriptor);
```

obj- Object name is the first argument, it refers to the object on which a property is defined.

prop- It is the second argument that indicates the name of the property to be defined or modified.

descriptor- It is the third argument that indicates the descriptor for the property to be defined or modified.

Example

const person = {};

Object.defineProperty(person, "fullName", {

get() {

```
    return this.firstName + " " + this.lastName;
  },
  set(name) {
    const parts = name.split(" ");
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
});

person.fullName = "Sultan Khan";
console.log(person.fullName);
```

Output:

Sultan Khan

Benefits of Getters and Setters in JavaScript

JS getter and setter offers several benefits to programmers, which we have discussed below:

- It has a simple syntax.
- It allows you to encapsulate the implementation details of the object property, so you can change the internal implementation of a property without changing the external code that uses the property.
- It improves data quality.
- It allows you to add validation and error-checking to the properties of an object, which mitigates the chances of bugs and enhances code robustness.
- It has equal syntax for methods and properties.
- It is useful for behind-the-scenes tasks.

JavaScript setTimeout()

The **setTimeout() in JavaScript** calls a function after a number of milliseconds. It is executed once, and to repeat execution, use setInterval() instead. To clear a timeout, you can use the id returned from setTimeout().

```
myTimeout = setTimeout(function, milliseconds);
```

And then stop the execution by calling clearTimeout()

```
clearTimeout(myTimeout);
```

What is setTimeout in JavaScript?

The **setTimeout() in JS** is the method used to execute the code block after a specific time. It delays or schedules the execution of a specific function after a certain time. It executes the code only once. JavaScript setTimeout() is a key feature of Node.JS and browser environments and enables asynchronous behavior in code execution. It returns an intervalID, which is a positive integer.

Syntax

```
setTimeout(function, delay, arg1, arg2, ...);
```

Parameters

function- The function containing the code block to be executed after a specific period.

milliseconds- The delay in time after which the function is executed.

arg1, arg2- Optional parameters.

Example

```
setTimeout(() => {
   console.log('Hello after 2 seconds!');
}, 2000); // Executes after 2000ms (2 seconds)
```

Cancellation of the setTimeout() Function

In JavaScript, you can prevent the setTimeout() method from executing the function using the corresponding function provided by JavaScript, known as clearTimeout(). It cancels the scheduled timeout before execution. The clearTimeout() method uses the returned ID by setTimeout() to know the setTimeout() method to cancel.

```
clearTimeout(id);
```

Example

```
const timeoutId = setTimeout(() => {
   console.log('This will not be logged!');
}, 3000); // Scheduled after 3 seconds


clearTimeout(timeoutId); // Cancels the above timeout
```

Use Cases of setTimeout() in JavaScript

Asynchronous Operations- When setTimeout() is combined with callbacks, it facilitates asynchronous behavior, which enables non-blocking code execution.

Delayed Execution- The setTimeout() function can delay code execution, which is useful for timed events, animations, and deferred operations.

Web Development- When used in web development, we can handle timeouts, such as refreshing content and showing a notification after a specific time.

JavaScript clearTimeout()

The setTimeout() function executes the code after a specified period. To stop the function call, use the clearTimeout() method.

Syntax

```
clearTimeout(intervalID);
```

intervalID represents the return value of setTimeout().

Example

```
let timeoutID;

// Start a timeout when the button is clicked
function startTimeout() {
    timeoutID = setTimeout(() => {
        console.log('This message appears after 5 seconds!');
    }, 5000); // 5 seconds delay

    console.log('Timeout started! Message will appear in 5 seconds.');
}

// Clear the timeout when the cancel button is clicked
function cancelTimeout() {
    clearTimeout(timeoutID);
    console.log('Timeout canceled!');
}

// Simulating button clicks
startTimeout();

// Simulate canceling the timeout after 2 seconds
setTimeout(() => {
    cancelTimeout();
}, 2000); // Cancels timeout after 2 seconds
```

Output

Timeout started! Message will appear in 5 seconds.

Timeout canceled!

JavaScript Date and Time

In JavaScript, the date object enables you to work with date and time. Using the new Date(), you can create a date. It handles **JS date and time** in precise milliseconds and can represent dates up to 100 million days starting from January 1, 1970, to the year 275755. This date and time is the same as the UNIX epoch, which is the base value for computer-recorded data and time values.

The data object in JavaScript provides the date and time information while providing several methods. Let's understand **JavaScript dates**, ways to create date objects, and more.

What is the JavaScript Date Object?

The JavaScript date object is used to get and set different parts of a date, such as a year, month, day, hour, minute, second, and millisecond. It tracks time as the number of milliseconds since the Unix Epoch. Using the new Date() constructor, you can create a date object and specify the current date and time or a specific date.

The date object allows you to work with date and time by offering methods for different tasks, such as date formatting, manipulation, and calculations.

JavaScript Date Object Syntax

```
new Date()

new Date(milliseconds)

new Date(dataString)

new Date(year, month, date, hour, minute, second, millisecond)
```

Creating Date Objects in JavaScript

JavaScript allows you to initialize the data object in different ways using the new Date() constructor. It supports several formats for defining JavaScript dates:

- new Date();

- new Date(value);
- new Date(dateString);
- new Date(year, month, day, hours, minutes, seconds, milliseconds);

new Date()

The new Date() constructor allows you to create a date object. It sets the current date and time depending on the time zone of the browser. This method doesn't accept any value.

Syntax

```
const currentDate = new Date();
```

Example

const currentDate = new Date();

console.log(currentDate);

Output

`2024-07-04T10:30:45.123Z` ***//(Current date and time in ISO format)***

new Date(milliseconds)

The data object consists of a number representing milliseconds since January 1, 1970, UTC. The new Date(milliseconds) method creates a new date object by adding the milliseconds to the zero time. It takes a single parameter (milliseconds) that indicates the numeric value, representing the internal numeric of the date in milliseconds.

```
const dateFromMilliseconds = new Date(1609459200000);  // Represents January 1, 2021
```

Example

const dateFromMilliseconds = new Date(1609459200000); // Jan 1, 2021

console.log(dateFromMilliseconds);

Output:

`2021-01-01T00:00:00.000Z`

new Date(date string)

The new Date(data string) method is used to create a new date object from a date string. It takes a single parameter (datestring) that represents a string value, returning the datestring with the day.

JavaScript has three data input formats, which include:

ISO Date Format

You can pass ISO date format to create a date object.

Example

```
const dateFromString = new Date('2024-07-04');

console.log(dateFromString);
```

Output

2024-07-04T00:00:00.000Z

Short and Long Date Format

These two formats include short date format and long date format.

Example

```
// Short Date Format

const shortDate = new Date('2024-07-04'); // ISO format (YYYY-MM-DD)

console.log(shortDate);

// Long Date Format

const longDate = new Date('July 4, 2024'); // Long date format

console.log(longDate);
```

Output

2024-07-04T00:00:00.000Z

2024-07-04T00:00:00.000Z

new Date(year, month, date, hour, minute, second, millisecond)

Using a new Date(year, month, date,....), you can create a date object by passing a specific date and time. In this method, you can pass the following seven parameters.

year- It is an integer value representing the year. It must always specify the complete year, so you need to write 2024 instead of 24.

month- This integer value represents the month and starts from 0 indicating January to 11, which is used for December.

date- It is also an integer value which represents the date.

hour- This integer value represents the hour in the 24-hour format.

minute- The integer value representing the minute.

second- This integer value represents the second.

millisecond- The integer value representing the millisecond.

Example

```
const specificDate = new Date(2024, 6, 4, 10, 30, 15, 500);

console.log(specificDate);
```

Output:

```
2024-07-04T10:30:15.500Z
```

Properties of Date Object

The date object in JavaScript has two properties, as explained below:

Prototype- It allows you to add properties and methods to an object.

Date constructor- It defines the function for creating the prototype of the Date object.

Example

```
// Adding a custom method to the Date prototype

Date.prototype.getFormattedDate = function () {

   return `${this.getDate()}-${this.getMonth() + 1}-${this.getFullYear()}`;

};
```

```
// Creating a date object
const today = new Date();
console.log(today.getFormattedDate());
```

Output

4-7-2024

Parameters for the Date Constructor

Using the data constructor, different parameter types define a specific date and time. Here is a list of possible parameters.

Field	Description
value	Represents the number of milliseconds since January 1, 1970, 00:00:00 UTC.
dateString	Represents a date format.
year	An integer value that represents the year, ranging from 1900 to 1999.

month	Also an integer representing the month, from 0 for January to 11 for December.
day	Optional integer that represents the day of the month.
hours	Also an optional integer that represents the hour of the day.
minutes	It is an optional integer that represents the minute of the time.
seconds	Optional integer representing the second of the time.
milliseconds	Another optional integer representing the millisecond of the time.

Retrieving Date Components

Using the following methods, you can retrieve different parts of a date:

getFullYear()- Returns the full year, for example, 2022.

getMonth(): Returns the month, from 0 for January to 11 for December.

getDate(): Returns the day of the month, from 1-31.

getHours(): Returns the hour, from 0-23.

getMinutes(): Returns the minutes, from 0-59.

getSeconds(): Returns the seconds, from 0-59.

getMilliseconds(): Returns the milliseconds, from 0-999.

Example

```
const now = new Date();
```

```
console.log(now.getFullYear()); // Year: e.g., 2024
console.log(now.getMonth());    // Month (0-11): e.g., 6
console.log(now.getDate());     // Day of the month: e.g., 4
```

Output

2024

6

4

Formatting Dates

In JavaScript, you can format dates either manually or use built-in options like Intl.DateTimeFormat to convert it into a readable format.

Example

```
const now = new Date();
const formattedDate = now.toLocaleDateString('en-US', {
   year: 'numeric',
   month: 'long',
   day: 'numeric'
});
console.log(formattedDate);
```

Output:

```
July 4, 2024
```

Manipulating Dates

Use the following methods of the Date object to modify dates:

setDate(): Used for adjusting the day of the month.

setFullYear(): Used to set the year.

setMonth(): Used to set the month.

setHours(): Used to set the hour.

Example 1: Adding 7 Days

```
const now = new Date();

now.setDate(now.getDate() + 7);

console.log(now);
```

Output

2024-07-11T10:30:45.123Z

Example 2: Creating a Specific Date

```
// Creating a specific date: July 4, 2024, at 10:30:15.500 AM

const specificDate = new Date(2024, 6, 4, 10, 30, 15, 500); // Month is zero-based (6 = July)

console.log(specificDate);
```

Output:

2024-07-04T10:30:15.500Z

AutoCorrection in Date Object

If you assign an out-of-range value in the date object, it auto-corrects itself.

Example

```
const autoCorrectedDate = new Date(2024, 1, 32); // Feb 32 auto-corrects to Mar 3

console.log(autoCorrectedDate);
```

Output:

2024-03-03T00:00:00.000Z

JavaScript Date Methods

Here are the non-static methods that define the date object usage and return the values based on the local time.

Method	**Description**
Date()	Returns the present day's date and time.
getDate()	Returns the day for a specific date (1-31) according to the local time.
getDay()	Returns the day of the week (0-6) for the specific date.
getFullYear()	Returns the year for the specific date.
getYear()	Returns the year for the specific date.
getHours()	Returns the hour (0-23) according to the local time.
getMilliseconds()	Returns the milliseconds based on the local time.

getMinutes()	Returns the minutes (0-59) for the specific date.
getMonth()	Returns the month (0-11) based on the local time.
getSeconds()	Returns the seconds.
getTime()	Returns the date in a numeric value as milliseconds.
setDate()	Sets the day of the month for a specific date.
setMonth()	Sets the month according to the local time.
setFullYear()	Sets the full year.

Methods to return values according to universal time

Methods	**Description**
getUTCDate()	Returns the day of a month (1-31).
getUTCDay()	Returns the day of the week.

getUTCFullYear()	Returns the year.
getUTCHours()	Returns the hours.
getUTCMilliseconds()	Returns the milliseconds.
getUTCMinutes()	Returns the minutes.
getUTCMonth()	Returns the month.