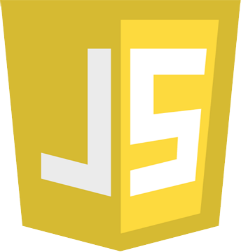
**JAVASCRIPT**



JavaScript is the **Programming Language** for the Web.

JavaScript can update and change both **HTML** and **CSS.**

JavaScript can **calculate**, **manipulate** and **validate** data.

|  |  |
| --- | --- |
| **Born** | 4 July 1961 (age 63)  [Pittsburgh](https://en.wikipedia.org/wiki/Pittsburgh), [Pennsylvania](https://en.wikipedia.org/wiki/Pennsylvania), US |
| **Alma mater** | [University of Illinois Urbana-Champaign](https://en.wikipedia.org/wiki/University_of_Illinois_Urbana-Champaign) [Santa Clara University](https://en.wikipedia.org/wiki/Santa_Clara_University) |
| **Known for** | Creation of [JavaScript](https://en.wikipedia.org/wiki/JavaScript) and co-founder of [Mozilla](https://en.wikipedia.org/wiki/Mozilla) project, [Mozilla Foundation](https://en.wikipedia.org/wiki/Mozilla_Foundation), and [Mozilla Corporation](https://en.wikipedia.org/wiki/Mozilla_Corporation) |
| **Website** | [brendaneich.com](https://brendaneich.com/) |



**Brendan Eich**

**Brendan Eich** ([/ˈaɪk/](https://en.wikipedia.org/wiki/Help:IPA/English); born July 4, 1961)[[1]](https://en.wikipedia.org/wiki/Brendan_Eich#cite_note-:0-1) is an American computer programmer and technology executive. He created the [JavaScript](https://en.wikipedia.org/wiki/JavaScript) [programming language](https://en.wikipedia.org/wiki/Programming_language) and co-founded the [Mozilla](https://en.wikipedia.org/wiki/Mozilla) project, the [Mozilla Foundation](https://en.wikipedia.org/wiki/Mozilla_Foundation), and the [Mozilla Corporation](https://en.wikipedia.org/wiki/Mozilla_Corporation). He served as the Mozilla Corporation's [chief technical officer](https://en.wikipedia.org/wiki/Chief_technical_officer) before he was appointed [chief executive officer](https://en.wikipedia.org/wiki/Chief_executive_officer), but resigned shortly after his appointment due to pressure over his firm opposition to same-sex marriage. He subsequently became the CEO of [Brave Software](https://en.wikipedia.org/wiki/Brave_(web_browser)).

**JAVASCRIPT ECMASCRIPT HISTORY**

**-JavaScript** was invented by **Brendan Eich** in **1995**.

-It was developed for **Netscape 2**, and became the **ECMA-262** standard in 1997.

-After Netscape handed JavaScript over to ECMA, the Mozilla foundation continued to develop JavaScript for the Firefox browser. Mozilla's latest version was 1.8.5. (Identical to ES5).

**-Internet Explorer** (IE4) was the first browser to support ECMA-262 Edition 1 (ES1).

|  |  |  |
| --- | --- | --- |
| Year | ECMA | Browser |
| 1995 |  | JavaScript was invented by Brendan Eich |
| 1996 |  | Netscape 2 was released with JavaScript 1.0 |
| 1997 |  | JavaScript became an ECMA standard (ECMA-262) |
| 1997 | ES1 | ECMAScript 1 was released |
| 1997 | ES1 | IE 4 was the first browser to support ES1 |
| 1998 | ES2 | ECMAScript 2 was released |
| 1998 |  | Netscape 42 was released with JavaScript 1.3 |
| 1999 | ES2 | IE 5 was the first browser to support ES2 |
| 1999 | ES3 | ECMAScript 3 was released |
| 2000 | ES3 | IE 5.5 was the first browser to support ES3 |
| 2000 |  | Netscape 62 was released with JavaScript 1.5 |
| 2000 |  | Firefox 1 was released with JavaScript 1.5 |
| 2008 | ES4 | ECMAScript 4 was abandoned |
| 2009 | ES5 | ECMAScript 5 was released |
| 2011 | ES5 | IE 9 was the first browser to support ES5 \* |
| 2011 | ES5 | Firefox 4 was released with JavaScript 1.8.5 |
| 2012 | ES5 | Full support for ES5 in Safari 6 |
| 2012 | ES5 | Full support for ES5 in IE 10 |
| 2012 | ES5 | Full support for ES5 in Chrome 23 |
| 2013 | ES5 | Full support for ES5 in Firefox 21 |
| 2013 | ES5 | Full support for ES5 in Opera 15 |
| 2014 | ES5 | Full support for ES5 in all browsers |
| 2015 | ES6 | ECMAScript 6 was released |
| 2016 | ES6 | Full support for ES6 in Chrome 51 |
| 2016 | ES6 | Full support for ES6 in Opera 38 |
| 2016 | ES6 | Full support for ES6 in Safari 10 |
| 2017 | ES6 | Full support for ES6 in Firefox 54 |
| 2017 | ES6 | Full support for ES6 in Edge 15 |
| 2018 | ES6 | Full support for ES6 in all browsers \*\* |

## The ECMA Technical Committee 39

In 1996, Netscape and Brendan Eich took JavaScript to the ECMA international standards organization, and a technical committee (TC39) was created to develop the language.

ECMA-262 Edition 1 was released in June 1997.

## From ES4 to ES6

When the TC39 committee got together in Oslo in 2008, to agree on ECMAScript 4, they were divided into 2 very different camps:

**The ECMAScript 3.1 Camp**:  
Microsoft and Yahoo who wanted an incremental upgrade from ES3.

**The ECMAScript 4 Camp**:  
Adobe, Mozilla, Opera, and Google who wanted a massive ES4 upgrade.

August 13 2008, Brendan Eich wrote an [email](https://mail.mozilla.org/pipermail/es-discuss/2008-August/006837.html):

*-It's no secret that the JavaScript standards body, Ecma's Technical Committee 39, has been split for over a year, with some members favoring ES4, a major fourth edition to ECMA-262, and others advocating ES3.1 based on the existing ECMA-262 Edition 3 (ES3) specification. Now, I'm happy to report, the split is over.*

The solution was to work together:

* ECMAScript 4 was renamed to ES5
* ES5 should be an incremental upgrade of ECMAScript 3.
* Features of ECMAScript 4 should be picked up in later versions.
* TC39 should develop a new major release, bigger in scope than ES5.

The planned new release (ES6) was codenamed "Harmony" (Because of the split it created?).

ES5 was a huge success. It was released in 2009, and all major browsers (including Internet Explorer) were fully compliant by July 2013:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Chrome 23 | IE10 / Edge | Firefox 21 | Safari 6 | Opera 15 |
| Nov 2012 | Sep 2012 | May 2013 | Jul 2012 | Jul 2013 |

ES6 was also a huge success. It was released in 2015, and all major browsers were fully compliant by March 2017:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Chrome 51 | Edge 14 | Firefox 52 | Safari 10 | Opera 38 |
| May 2016 | Aug 2016 | Mar 2017 | Sep 2016 | Jun 2016 |

**What is Website?**   
Website is the collection of web pages, different multimedia content such as text, images, and videos which can be accessed by the URL which you can see in the address bar of the browser. For example: [https://www.geeksforgeeks.org](https://www.geeksforgeeks.org/) .

**How to access Websites?**   
When we type a certain URL in a browser search bar, the browser requests the page from the Web server and the Web server returns the required web page and its content to the browser.

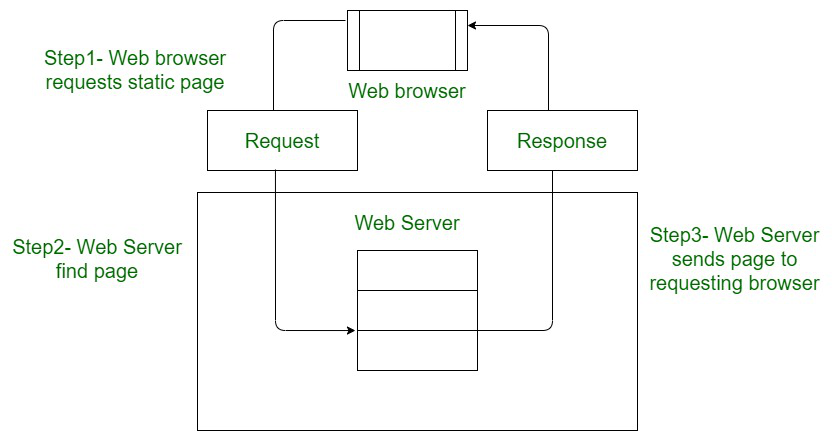
**Types of Website:**

* Static Website
* Dynamic Website

**Static Website:**

-These websites are made up of individual HTML files and display fixed content that doesn't change. They don't interact with users or change in real time. Static websites are built using HTML, CSS, and JavaScript.

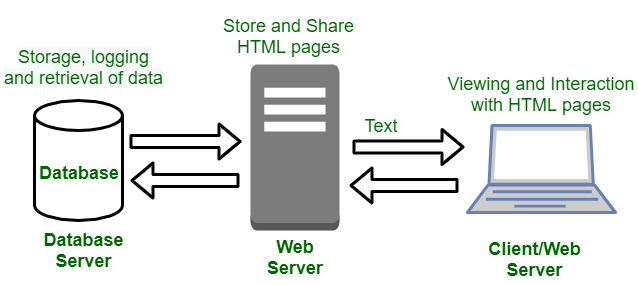
-In Static Websites, Web pages are returned by the server which are prebuilt source code files built using simple languages such as HTML, CSS, or JavaScript. There is no processing of content on the server (according to the user) in Static Websites. Web pages are returned by the server with no change therefore, static Websites are fast. There is no interaction with databases. Also, they are less costly as the host does not need to support server-side processing with different languages.



**Dynamic Website:**

-These websites use back-end code to build each web page on the fly, allowing them to change based on the user, time, or other factors. They can adjust to the viewer's screen size, change language automatically, and provide custom recommendations. Dynamic websites use server-side languages like PHP, Ruby, Python, and server-side JavaScript.

-In Dynamic Websites, Web pages are returned by the server which are processed during runtime means they are not prebuilt web pages but they are built during runtime according to the user’s demand with the help of server-side scripting languages such as PHP, Node.js, ASP.NET and many more supported by the server. So, they are slower than static websites but updates and interaction with databases are possible.



**Difference Between Static Website and Dynamic Website**

-The main difference between static and dynamic websites is that static websites display the same content to all users, while dynamic websites can change based on the user or other factors:

|  |  |
| --- | --- |
| Static Website | Dynamic Website |
| Content of Web pages can not be change at runtime. | Content of Web pages can be changed. |
| No interaction with database possible. | Interaction with database is possible |
| It is faster to load as compared to dynamic website. | It is slower than static website. |
| Cheaper Development costs. | More Development costs. |
| No feature of Content Management. | Feature of Content Management System. |
| HTML, CSS, Javascript is used for developing the website. | Server side languages such as PHP, Node.js are used. |
| Same content is delivered everytime the page is loaded. | Content may change everytime the page is loaded. |

**Differences Between Web Application And Website:**

|  |  |
| --- | --- |
| Web Application | Website |
| Web application is designed for interaction with end users. | Website basically contains static content. |
| The user of web application can read the content of web application and also manipulate the data. | The user of website only can read the content of website but not manipulate . |
| The web application site should be precompiled before deployment. | The website does not need to be precompiled . |
| The function of a web application is quite complex. | The function of website is simple. |
| Web application is interactive for users. | Web site is not interactive for users. |
| The browser capabilities involved with a web application is high. | The browser capabilities involved with web site is high. |
| Integration is complex for web application because of its complex functionality. | Integration is simpler for web site. |
| Web application mostly requires authentication | In web site authentication is not necessary. |
| EXAMPLE :- Amazon, Facebook, etc. | EXAMPLE :- Breaking News, Aktu website, etc. |



**Difference Between Web App And Android App**

-The main difference between a web app and an app is that web apps are accessible through a browser and don't need to be installed on a device, while apps are built for a specific platform and need to be installed on a device

Other differences include:

* Platform

Web apps work on any platform, device, and version of the system, while apps are built for a specific platform or device type.

* Development

Web apps are built using technologies like JavaScript, CSS, HTML, and Python, while apps are built using platform-specific technologies.

* Features

Apps may have more complex features and functionalities than web apps, and may have access to system resources like GPS and the camera.

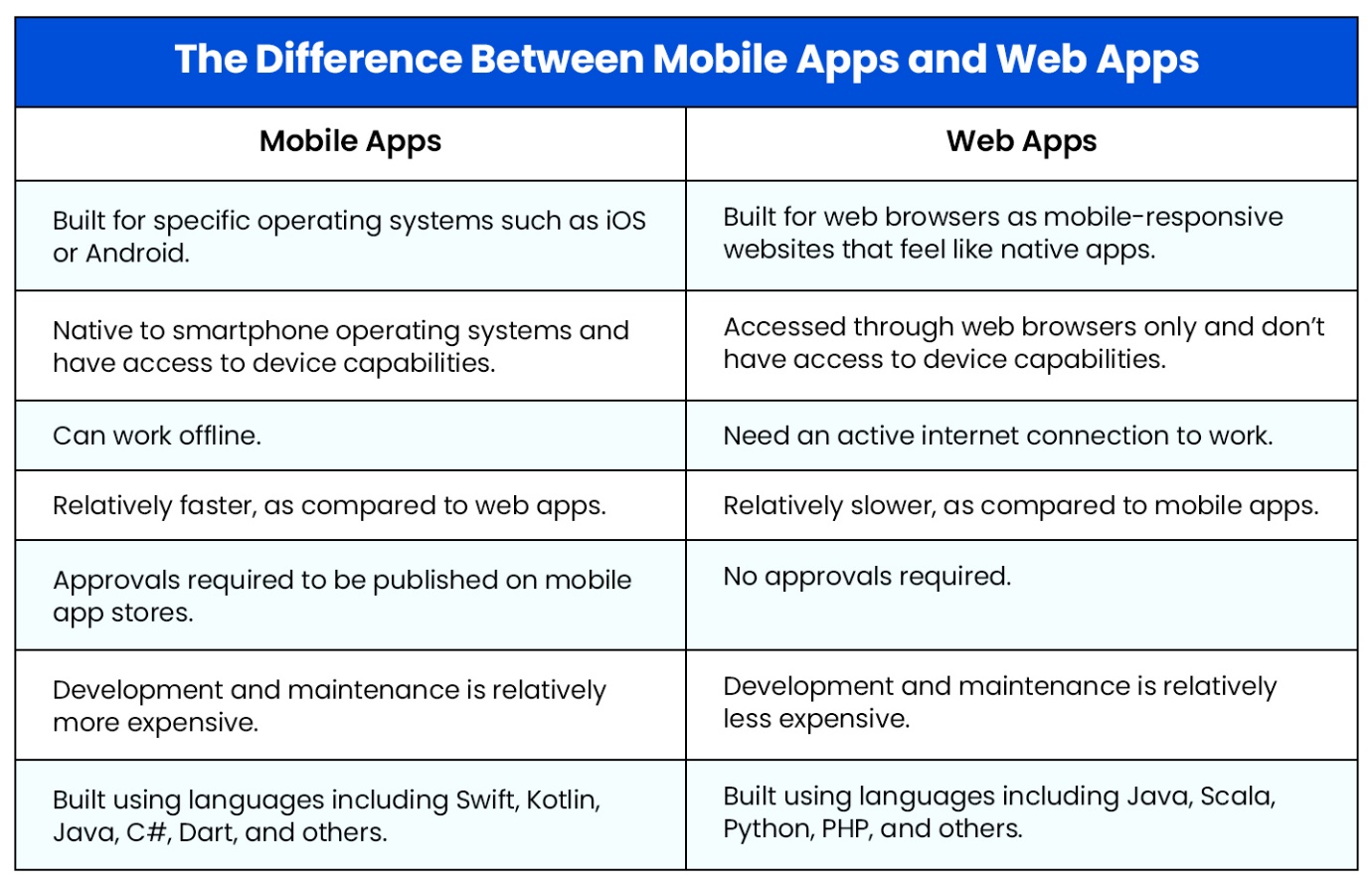
* Offline use

Apps may work offline, while web apps do not.

* Testing

Web app testing ensures quality, functionality, and usability, while app testing refers to testing application software for controlling and handling mobile devices.

| **Mobile Apps[Native] Apps** | **Web Apps** |
| --- | --- |
| Mobile apps are developed for a specific platform, such as iOS for the Apple iPhone or Android | On the other hand, Web Apps are accessed via the internet browser and will function according to the device you’re viewing them on |
| They are downloaded and installed via an app store such as Google Play Store and Apple Store and have access to system resources, such as GPS and the camera of the device. | Web apps are not native to a particular system and there is no need to be downloaded or installed. |
| Mobile apps may work offline. | In order to run web apps need an active internet connection. |
| Native Apps are comparatively faster. | Web Apps are comparatively slower. |
| It is difficult to have a native mobile app approved by the App Store. | In Web Apps App store approval is not required, so web apps can be launched easily. |
| Native apps have more safety and security. | Web apps have comparatively low security. |
| Maintaining and consistently update of native apps cause more cost. | These apps can be set to update themselves or automatically. |



**Script**

-In programming, a script is a set of instructions that a computer or programming language can execute to perform a task.

-a script is a program or sequence of instructions that is interpreted or carried out by another program rather than by the computer processor (as a compiled program is).

-According to your instruction the components or objects perform the tasks.

**Advantages of scripts**

* Allows users to view and edit the **script** if needed.
* Does not require the file to be [compiled](https://www.computerhope.com/jargon/c/compile.htm), but it may be when necessary.
* Easy to learn and write.
* Easy to [port](https://www.computerhope.com/jargon/p/port.htm) between different [operating systems](https://www.computerhope.com/jargon/o/os.htm).
* Much faster to develop than an actual program - some individuals and companies write scripts as a prototype for actual programs.

**What Is JavaScript?**

-JavaScript is a scripting language for creating dynamic web page content. It creates elements for improving site visitors’ interaction with web pages, such as dropdown menus, animated graphics, and dynamic background colors.

-JavaScript (often shortened to JS) is a lightweight, interpreted, or just-in-time compiled programming language commonly used for adding interactivity and dynamic features to websites.

ere are a few key points about JavaScript:

* **Purpose**:

-JavaScript is primarily used for front-end web development, meaning it runs directly in the user's web browser. It allows developers to create dynamic content, handle user interactions, update web pages without reloading, and build interactive web applications.

* **Relationship with HTML and CSS:**

-JavaScript works alongside HTML (structure) and CSS (styling) to create the complete user interface of a website. HTML provides the basic structure, CSS styles the elements, and JavaScript adds behavior and interactivity.

* **Syntax**:

-JavaScript's syntax is similar to other C-style languages like Java and C++. It's known for being relatively beginner-friendly, although it has its complexities as you dive deeper.

In simple words, JavaScript is the language that makes websites interactive and dynamic.

1.Scripting Language

2.Interpreted Language

3.Ligth-Weight language

4.WeaklyTyped Language

5.Object Based Language

7. Client-Side Validation

8.Versatility

9. DOM Manipulation

10.Event Handling

**FEATURES OF JAVASCRIPT**

## 1. Light-Weight Scripting Language

* JavaScript is a lightweight scripting language because it is made for data handling in the browser or the client side.
* Because JavaScript is meant for client-side execution for web applications, hence the lightweight nature of JavaScript is a great feature.

## 2. Dynamic Typing

* JavaScript supports dynamic typing which means types of the variable are defined based on the stored value.
* For example, if you declare a variable x then you can store either a string or a [Number type value in JavaScript](https://www.studytonight.com/javascript/javascript-number-object) or an array or an object. This is known as dynamic typing.
* JavaScript allows you to declare variables without specifying their type. The type is determined at runtime, providing flexibility in coding.

**<!--! ======= WEAKLY TYPED LANGUAGE =========-->**

*<! --\* 1.We can directly declare the variable without variable type.  -->*

    <script>

        a=5;

    </script>

*<! --\* 2. In below example Semicolon Is Not Mandatory -->*

*<! --\* For better Code Indentation or to main company standard We can write semicolon -->*

*<! --\* So, JS cannot give errors -->*

    <script>

        a=10

    </script>

*<! --\* 3. In JavaScript, a variable can be declared and assigned a value of any type, including objects, arrays, and functions.  -->*

    <script>

        var a=5;

        console.log(typeof(a));

        a="Amol";

        console.log(typeof(a));

        a=["Amol",24,2000];

        console.log(typeof(a));

        a=true;

        console.log(typeof(a));

        a=null;

        console.log(typeof(a));

    </script>

## 3. Object-Oriented Programming support

* Starting from ES6, the concept of class and OOP is better defined.
* Also, in JavaScript, two important principles with OOP in JavaScript are:
  + Object Creation Patterns (Encapsulation)
  + Code Reuse patterns (Inheritance)
* Although JavaScript developers rarely use these features, it's there for everyone to explore.

## 4. Functional Style

* This implies that JavaScript uses a functional approach, even objects can be created using constructor functions and each constructor function represents a unique object type.
* Also, functions in JavaScript can be used as objects and can be passed to other functions too.
* Many important JavaScript concepts and features like callbacks, closures, etc. are implementations of functions only.

## 5. Platform Independent

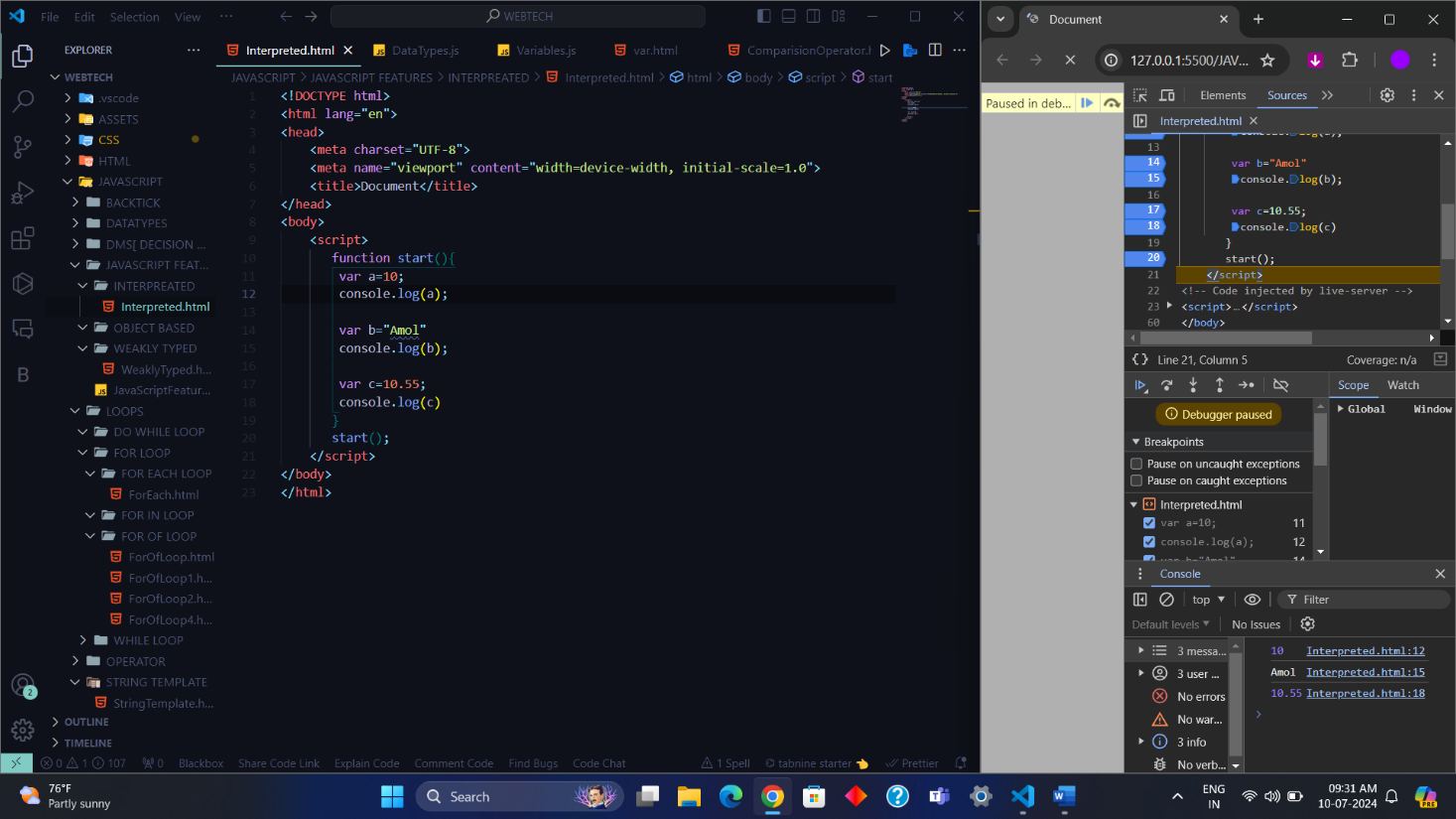
* This implies that JavaScript is platform-independent or we can say it is portable.
* This simply means that you can simply write the script once and run it anywhere and anytime.
* In general, you can write your JavaScript applications and run them on any platform or any browser without affecting the output of the Script.

## 6. Prototype-based Language

* JavaScript is a prototype-based scripting Language.
* This means JavaScript uses prototypes instead of classes or inheritance.
* In languages like Java, we create a class and then we create objects for those classes.
* But in JavaScript, we define an object prototype, and then more objects can be created using this object prototype.

## 7. Interpreted Language

* JavaScript is an interpreted language which means the script written inside JavaScript is processed line by line.
* The JS code is interpreted by JavaScript interpreter which is a built-in component of the Web browser.
* But these days many JavaScript engines in browsers like the V8 engine in Chrome use just-in-time compilation for JavaScript code.



## 8. Single threaded

* JavaScript doesn't support multi-threading, by default it is single-threaded, which means it can execute only a single task at a time.
* But JavaScript provides some features using which you can implement parallel execution. They are:
  + Async processing
  + Web workers

## 9. Async Processing

* JavaScript supports Promise which enables asynchronous requests wherein a request is initiated and JavaScript doesn't have to wait for the response if a request takes time and may block the request processing.
* Also starting from ES8, Async functions are also supported in JavaScript, these functions don't execute one by one, rather they are processed parallelly which has a positive effect on the processing time, reducing it to a great extent.

## 10. Web Workers

* Using Web workers you can run processes in background threads so that you can have parallel execution.
* If you have any task that performs some heavy-duty work, then you can use a Web worker to run it in the background.

## 11. Client-side Validations

* This is a feature that has been available in JavaScript since the beginning and is still widely used because every website has a form in which users enter values, and to make sure that users enter the correct value, we must put proper validations in place, both on the client side and on the server-side.
* JavaScript is used for implementing client-side validations.

## 12. More control in Browser

JavaScript being a client-side language provides many features that help developers to divide processing between browser and server hence reducing the load on servers by having basic processing, validations, temporary data saving using cookies, etc. on the browser itself.

Along with all these features, JavaScript provides the following useful features too:

* Detecting browser type, name OS version, etc information of the client for analysis.
* Extensive in-built library with many useful functions for validation, data type conversion, string operations, etc.
* It is an object-centered language with Window Object being the most important object in JavaScript and it also supports Polymorphism.
* Support for commonly used complex data types like arrays, Maps, lists, etc. with support of in-built functions to operate on them.

## 13. Backend Development

With NodeJS, backend development can also be done using JavaScript. MERN stack is one of the most popular stacks based on JavaScript for [Fullstack Development](https://www.studytonight.com/courses/fullstack-development/) using JavaScript. ExpressJS can be used for API or REST service development. NodeJS brings in npm which is a package manager for JavaScript modules, hence you can use 3rd party packages too, making the backend development even more fun and easy in JavaScript.

**Compiler**

A language processor that converts a program written in high-level language into machine language, entire program at once, is called a **compiler**. Thus, the input of a compiler is a high-level language code (called source code), while its output is a machine language code (called object code).

A compiler scans whole program and then check it for syntactic and semantic error, once the code is checked for errors, it is converted into an object code. Then, it can be processed by the machine to perform the corresponding task. The common programming languages that use compiler are C, C++, C#, etc.

**Role of a Compiler**

## For Converting the code written in a high-level language into machine-level language so that computers can easily understand, we use a compiler. Converts basically convert high-level language

## Advantages of Compiler

There are various advantages of the compiler which are as follows −

* A compiler translates a program in a single run.
* It consumes less time.
* CPU utilization is more.
* Both syntactic and semantic errors can be checked concurrently.
* It is easily supported by many high-level languages like C, C++, JAVA, etc.

**Disadvantages of Compiler**

* The compiler can catch only [syntax errors and some semantic errors](https://www.geeksforgeeks.org/difference-between-syntax-and-semantics/).
* Compilation can take more time in the case of bulky code.

**Interpreter**

-A language translator that converts a high-level language program into a machine language program, one line at a time, is referred to as an **interpreter**. Interpreters converts the codes slower than compiler. This is because the interpreter can scan and translate only one statement of the program at a time. Therefore, interpreters convert the source code into machine code during the execution of the program.

-Interpreters do not generate an object code corresponding to the source code. However, these are relatively easy to use and execute the code. The programming languages that use interpreters are Perl, Ruby, Python, METLAB, etc.

## Advantages of Interpreter

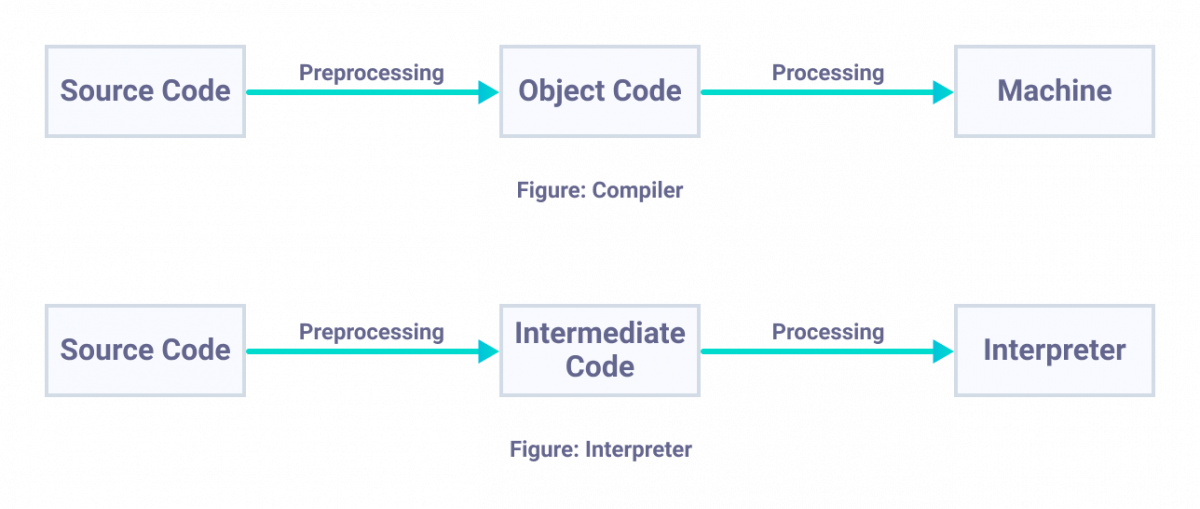
There are various advantages of the interpreter which are as follows −

* An interpreter translates the program line by line.
* The interpreter is smaller in size.
* It is flexible.
* Error localization is easier.
* The interpreter facilitates the implementation of computer programming language constructs.

**Disadvantages of Interpreter**

* The interpreter can run only the corresponding Interpreted program.
* Interpreted code runs slower in comparison to Compiled code.

## Working of Compiler and Interpreter



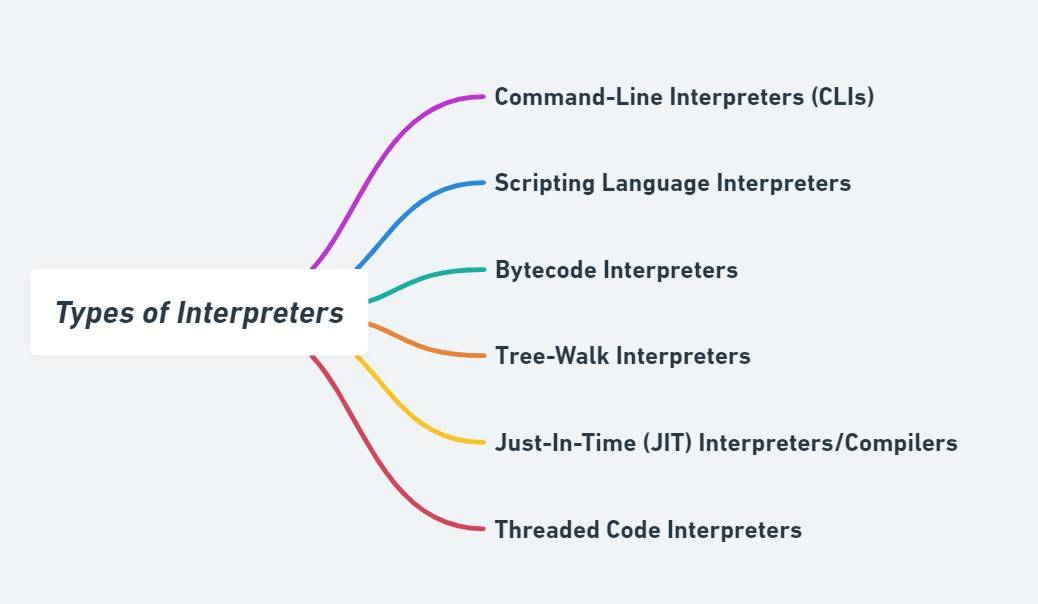
-Both **compilers** and **interpreters** are the language processors used to convert software codes written in high-level language into machine language codes.

**Difference between Compiler and Interpreter**

The following table highlights all the significant differences between a Compiler and an Interpreter −

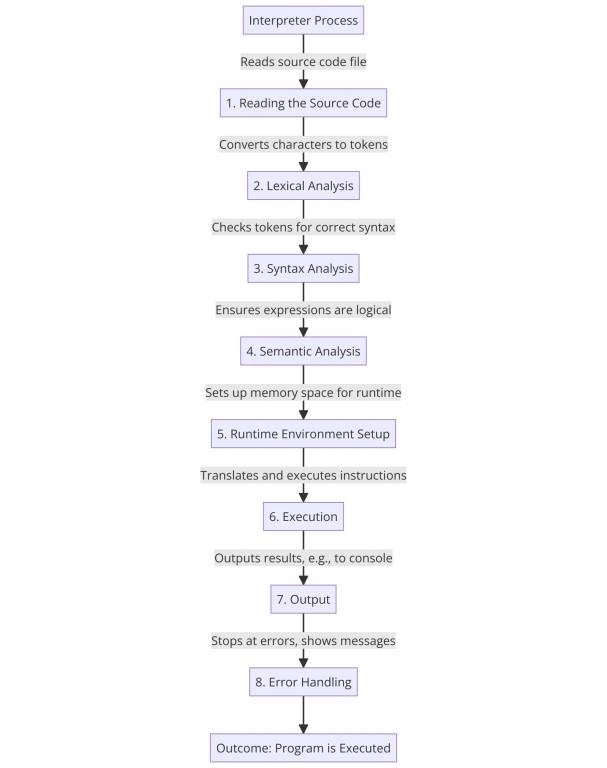
|  |  |  |
| --- | --- | --- |
| Parameter | Compiler | Interpreter |
| Program scanning | Compilers scan the entire program in one go. | The program is interpreted/translated one line at a time. |
| Error detection | As and when scanning is performed, all the errors are shown in the end together, not line by line. | One line of code is scanned, and errors encountered are shown. |
| Object code | Compilers convert the source code to object code. | Interpreters do not convert the source code into object code. |
| Execution time | The execution time of compiler is less, hence it is preferred. | It is not preferred due to its slow speed. Usually, interpreter is slow, and hence takes more time to execute the object code. |
| Need of source code | Compiler doesn’t require the source code for execution later. | It requires the source code for execution later. |
| Programming languages | Programming languages that use compilers include C, C++, C#, etc.. | Programming languages that uses interpreter include Python, Ruby, Perl, MATLAB, etc. |
| Types of errors detected | Compiler can check syntactic and semantic errors in the program simultaneously. | Interpreter checks the syntactic errors only. |
| Size | Compiler are larger in size. | Interpreters are smaller in size. |
| Flexibility | Compilers are not flexible. | Interpreters are relatively flexible. |
| Efficiency | Compilers are more efficient. | Interpreters are less efficient. |

### **Types of Interpreters**

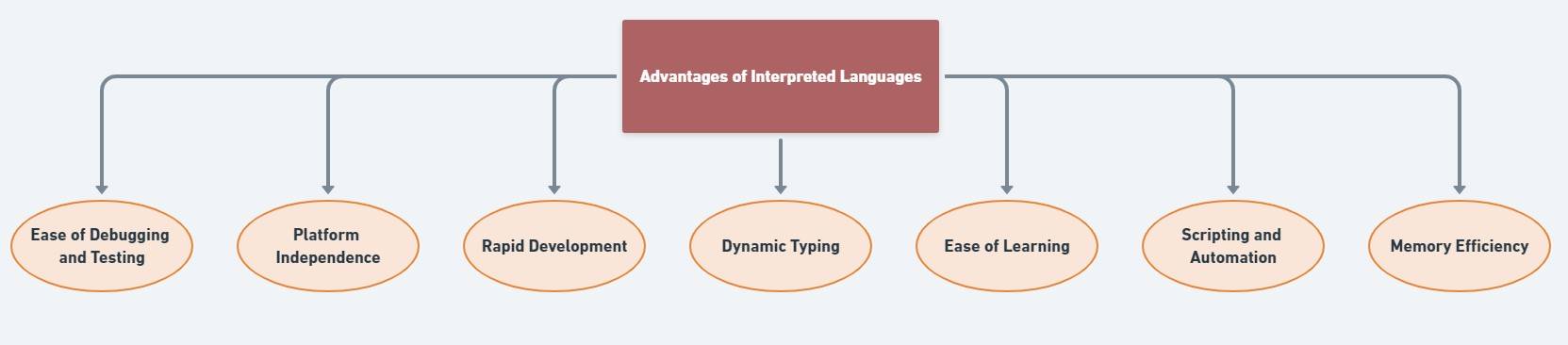


**-Let's read about each one in detail below!**

|  |  |  |
| --- | --- | --- |
| Type | Description | Examples |
| *Command-Line Interpreters (CLIs)* | Execute user commands one at a time. | Unix shells like Bash. |
| *Scripting Language Interpreters* | Used for scripting languages to automate tasks. | Python, Ruby, Perl. |
| *Bytecode Interpreters* | Interpret an intermediate bytecode, not the high-level source code directly. | Java Virtual Machine (JVM). |
| *Tree-Walk Interpreters* | Parse code into an abstract syntax tree and traverse it to interpret the program. | Simple language interpreters in educational tools. |
| *Just-In-Time (JIT) Interpreters/Compilers* | Compile source code or bytecode to machine code just in time for execution. | Modern JavaScript engines like V8 (in Chrome) or SpiderMonkey (in Firefox). |
| *Threaded Code Interpreters* | Execute sequences of simpler instructions for better performance. | Forth, some implementations of Python. |



### **Advantages of Using an Interpreter**



*Let's read about each one in detail below!*

|  |  |
| --- | --- |
| Advantages | Explanation |
| *Ease of Debugging and Testing* | Errors can be detected and fixed as the code is executed line by line. |
| *Platform Independence* | Code can run on multiple platforms without modification as long as the interpreter is available. |
| *Rapid Development* | Immediate execution allows for quick testing and iteration without the compile-link-execute cycle. |
| *Dynamic Typing* | Supports on-the-fly typing, which can lead to faster development times and less code verbosity. |
| *Ease of Learning* | Simplifies the learning process for beginners by allowing immediate execution of code snippets. |
| *Scripting and Automation* | Ideal for writing scripts that automate tasks due to their ease of use and execution. |
| *Memory Efficiency* | No need for executable binaries, which conserves memory space on the system. |

**Note :-** We cannot perform the Function Overloading in JavaScript because JavaScript has Default Datatype called as "Undefined"

**Note** :- In JavaScript Entire Memory Management is doing the **JsEngine** we do not worry about the Memory Management

-A JavaScript engine is a computer program that executes JavaScript code and converts it into computer understandable language.

List of JavaScript Engines:

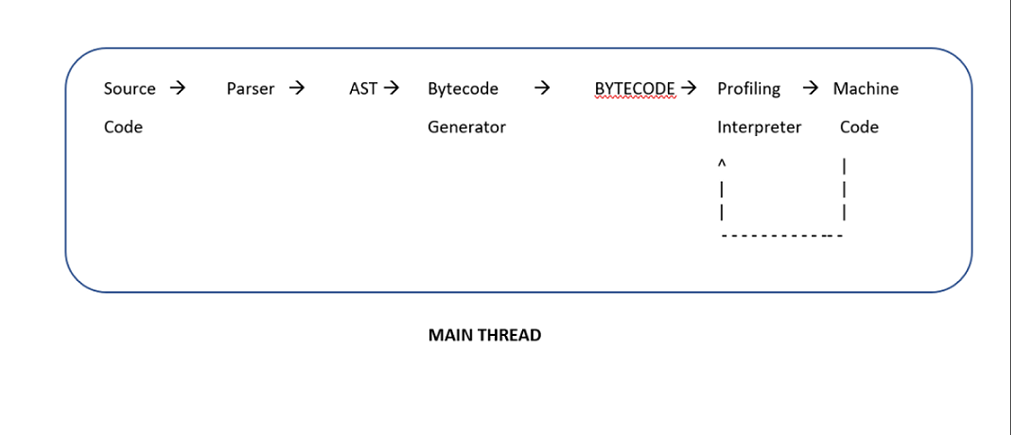
|  |  |
| --- | --- |
| Browser | Name of Javascript Engine |
| Google Chrome | V8 |
| Edge (Internet Explorer) | Chakra |
| Mozilla Firefox | Spider Monkey |
| Safari | Javascript Core Webkit |

Let’s understand each of them.

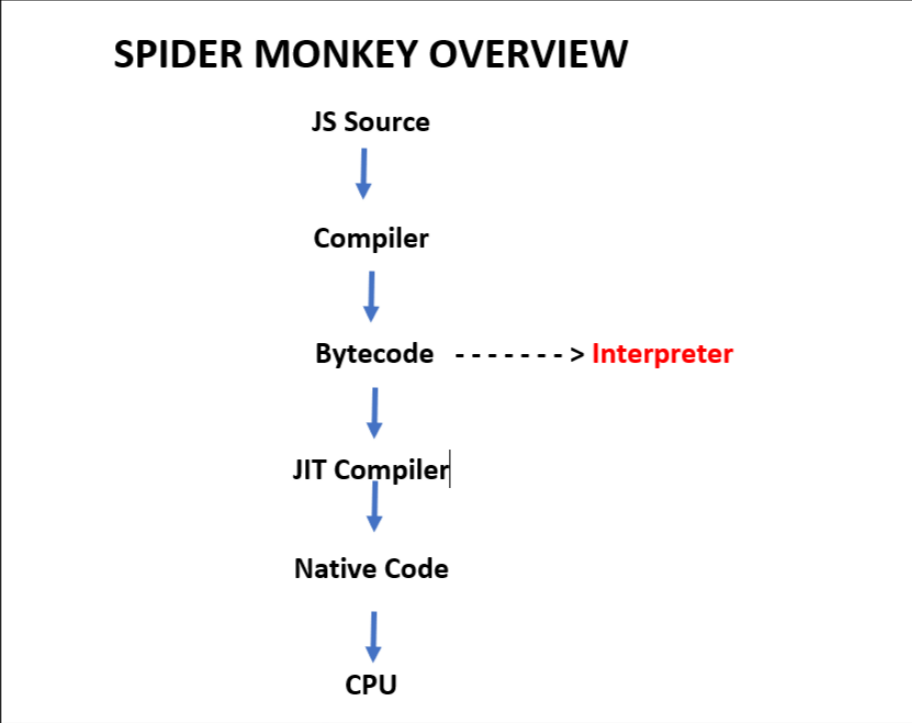
**1.** **V8**: V8 is a JavaScript engine developed by the Chromium Project for Google Chrome and Chromium web browsers. It is a JavaScript engine that can run standalone, or be embedded into any C++ application. Using its own parser, it generates an abstract syntax tree. Then, Ignition generates bytecode from this syntax tree using the internal V8 bytecode format. Bytecode is compiled into machine code by TurboFan. It also handles memory allocation for objects, and garbage collects objects it no longer needs. Optimization techniques such as elision of expensive runtime properties, and inline caching. The garbage collector is a generational incremental collector.

-V8 provides an edge as it allows JavaScript to run much faster, which improves users’ experience of the web, paves the way for the development of web applications, and spurs rapid growth of server-side JavaScript through projects like Node.js.

**2.** **Chakra**: Chakra is a JScript engine developed by Microsoft. It is proprietary software. It is used in the Internet Explorer web browser. A distinctive feature of the engine is that it JIT compiles scripts on a separate CPU core, parallel to the web browser.



**3.** **Spider Monkey**: SpiderMonkey is the first JavaScript engine, written by Brendan Eich at Netscape Communications, later released as open-source and currently maintained by the Mozilla Foundation. It is still used in the Firefox web browser.



**4. Webkit:** WebKit is developed by Apple and used in its Safari web browser, as well as all iOS web browsers. It is used by the BlackBerry Browser, PlayStation consoles beginning from the PS3, the Tizen mobile operating systems, and a browser included with the Amazon Kindle e-book reader. WebKit’s C++ Application Programming Interface (API) provides a set of classes to display Web content in windows and implements browser features such as following links when clicked by the user, managing a back-forward list, and managing a history of pages recently visited.

JavaScript is one of the 3 languages all web developers must learn:

   1. [HTML](https://www.w3schools.com/html/default.asp) to define the content of web pages

   2. [CSS](https://www.w3schools.com/css/default.asp) to specify the layout of web pages

**HOW TO ADD JAVASCRIPT TO HTML**

-We can write or add a script in two ways

**1]Internal**

-You can add JavaScript code in an HTML document by employing the dedicated HTML tag <script> that wraps around JavaScript code.

-The <script> tag can be placed in the <head> section of your HTML or in the <body> section, depending on when you want the JavaScript to load.

**A]Inside Head Section**

Generally, JavaScript code can go inside the document <head> section in order to keep it contained and out of the main content of your HTML document.

-We can write multiple scripts mean there can be multiple <script> tags.

-But the order of the execution is line-by-line in Top-To-Bottom order.

<!DOCTYPE html>

<html lang="en-US">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>In Body JavaScript</title>

*<!--! 1.First Script -->*

    <script>

        function myFunction() {

          document.getElementById("demo").innerHTML = "Paragraph changed.";

        }

    </script>

*<!--! 2.Second Script -->*

    <script>

        let d = new Date();

        document.write("Today's date is " + d);

    </script>

</head>

<body>

    <h2>JavaScript in Head</h2>

    <p id="demo"></p>

    <button type="button" onclick="myFunction()">Try it</button><br>

</body>

</html>

**B]Inside Body Section**

-We can write a script inside the body section also for that we have add **<script>** <**/script>** tag In body section as show below.

-We can write multiple scripts mean there can be multiple <script> tags.

-But the order of the execution is line-by-line in Top-To-Bottom order.

<!DOCTYPE html>

<html lang="en-US">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>In Body JavaScript</title>

</head>

<body>

    <h2>JavaScript in Head</h2>

    <p id="demo"></p>

    <button type="button" onclick="myFunction()">Try it</button><br>

*<!--! 1.First Script -->*

    <script>

        function myFunction() {

          document.getElementById("demo").innerHTML = "Paragraph changed.";

        }

    </script>

*<!--! 2.Second Script -->*

    <script>

        let d = new Date();

        document.write("Today's date is " + d);

    </script>

</body>

</html>

**2]External**

-For That you have create a new file with **.js** extension then inside that JavaScript file you have directly write the JavaScript code. There is no <script></script> tag is required.

-After Writing Script then you have link the script to **HTML** document.

-For that you have to use **<script src=”./”>** tag. Where **src** means Source or Location of that file.

**NOTE : - You can link JavaScript file in Head Section also.**

**External.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

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

</head>

<body>

    <h2>JavaScript in Head</h2>

    <p id="demo"></p>

    <button type="button" onclick="myFunction()">Try it</button><br>

</body>

</html>

**External.js**

*//! 1.First Script  -->*

    function myFunction() {

      document.getElementById("demo").innerHTML = "Paragraph changed.";

    }

*//! 2.Second Script-->*

    let d = new Date();

    document.write("Today's date is " + d);

**NOTE : - You can link Javascript File in Body Section also.**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

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

</body>

</html>

**External JavaScript Advantages**

Using external JavaScript files offers several advantages:

**1. Improved Code Organization and Readability:**

* Separates HTML structure from JavaScript behaviour, making both files cleaner and easier to understand and maintain.
* Improves code reusability as the same JavaScript file can be linked to multiple HTML pages.

**2. Faster Page Loading:**

* Browsers can cache external JavaScript files. This means that once a user visits a page, the browser can store the file locally, so it doesn't need to be downloaded again on subsequent visits, resulting in faster loading times.

**3. Easier Maintenance and Update:**

* Updating JavaScript code becomes simpler when it's in a separate file. You can make changes to the script without touching the HTML structure or other parts of the website, and these changes will be reflected across all pages that use the script. reducing the risk of introducing errors.

**4. Separation of Concerns (SoC):**

* Promotes a better development workflow by allowing designers and developers to focus on their respective areas without interfering with each other's work.
* Improves overall code quality and maintainability.

**5.Code Reusability**:

External JavaScript files can be reused across multiple web pages within the same site or even across different sites. This promotes consistency and reduces duplication of code.

6.**Accessibility**: Separating JavaScript into an external file makes it easier for developers to collaborate. Multiple developers can work on different parts of the codebase simultaneously without conflicts.

7.**Reduced HTML Size**: Placing JavaScript code externally reduces the size of the HTML file itself. This is especially beneficial for larger scripts, helping to keep your HTML code clean and more readable.

8.**Compatibility**: External JavaScript files are compatible with various development tools, version control systems, and content management systems (CMS). This flexibility makes them suitable for a wide range of web development projects.

**JAVASCRIPT CODING CONVENTIONS**

Coding conventions **secure quality**:

* Improve code readability
* Make code maintenance easier

**Naming Conventions**

-Always use the same naming convention for all your code. For example:

* Variable and function names written as camelCase
* Global variables written in UPPERCASE (We don't, but it's quite common)
* Constants (like PI) written in UPPERCASE

**var name = 'Robin Wieruch';**

**var Name = 'Dennis Wieruch';**

**var NAME = 'Thomas Wieruch';**

**console.log(name);**

***// "Robin Wieruch"***

**console.log(Name);**

***// "Dennis Wieruch"***

**console.log(NAME);**

***// "Thomas Wieruch"***

-A JavaScript variable should be self-descriptive. It shouldn't be necessary to add a comment for additional documentation to the variable:

***// bad***

**var value = 'Robin';**

***// bad***

**var val = 'Robin';**

***// good***

**var firstName = 'Robin';**

**// bad**

**var firstname = 'Robin';**

**// bad**

**var first\_name = 'Robin';**

**// bad**

**var FIRSTNAME = 'Robin';**

**// bad**

**var FIRST\_NAME = 'Robin';**

**// good**

**var firstName = 'Robin';**

**JAVASCRIPT NAMING CONVENTIONS: BOOLEAN**

-A prefix like *is*, *are*, or *has* helps every JavaScript developer to distinguish a boolean from another variable by just looking at it:

***/ bad***

**var visible = true;**

***// good***

**var isVisible = true;**

***// bad***

**var equal = false;**

***// good***

**var areEqual = false;**

***// bad***

**var encryption = true;**

***// good***

**var hasEncryption = true;**

**JAVASCRIPT NAMING CONVENTIONS: FUNCTION**

-JavaScript functions are written in camel case too. In addition, it's a best practice to actually tell *what the function is doing* by giving the function name a verb as prefix.

***// bad***

**function name(firstName, lastName) {**

**return `${firstName} ${lastName}`;**

**}**

***// good***

**function getName(firstName, lastName) {**

**return `${firstName} ${lastName}`;**

**}**

**JAVASCRIPT NAMING CONVENTIONS: CLASS**

-A JavaScript class is declared with a PascalCase in contrast to other JavaScript data structures:

**class SoftwareDeveloper {**

**constructor(firstName, lastName) {**

**this.firstName = firstName;**

**this.lastName = lastName;**

**}**

**}**

**var me = new SoftwareDeveloper('Robin', 'Wieruch');**

**JAVASCRIPT NAMING CONVENTIONS: COMPONENT**

Components are not everywhere in JavaScript, but commonly found in frontend frameworks like [React](https://www.robinwieruch.de/javascript-fundamentals-react-requirements/). Since a component is kinda instantiated -- but appended to the DOM instead -- like a JavaScript class, they are widely declared with Pascal Case too.

***/ bad***

**function userProfile(user) {**

**return (**

**<div>**

**<span>First Name: {user.firstName}</span>**

**<span>Last Name: {user.lastName}</span>**

**</div>**

**);**

**}**

***// good***

**function UserProfile(user) {**

**return (**

**<div>**

**<span>First Name: {user.firstName}</span>**

**<span>Last Name: {user.lastName}</span>**

**</div>**

**);**

**}**

-When a component gets used, it distinguishes itself from native HTML and [web components](https://www.robinwieruch.de/web-components-tutorial/), because its first letter is always written in uppercase

**<div>**

**<UserProfile**

***user*={{ firstName: 'Robin', lastName: 'Wieruch' }}**

**/>**

**</div>**

**JAVASCRIPT NAMING CONVENTIONS: METHODS**

-Identical to JavaScript functions, a method on a JavaScript class is declared with camelCase:

**class SoftwareDeveloper {**

**constructor(firstName, lastName) {**

**this.firstName = firstName;**

**this.lastName = lastName;**

**}**

**getName() {**

**return `${this.firstName} ${this.lastName}`;**

**}**

**}**

**var me = new SoftwareDeveloper('Robin', 'Wieruch');**

**console.log(me.getName());**

***// "Robin Wieruch"***

**JAVASCRIPT NAMING CONVENTIONS: CONSTANT**

Last but not least, there are constants -- intended to be non-changing variables -- in JavaScript which are written in capital letters (UPPERCASE):

**var SECONDS = 60;**

**var MINUTES = 60;**

**var HOURS = 24;**

**var DAY = SECONDS \* MINUTES \* HOURS;**

If a variable has more than one word in its variable declaration name, it makes use of an underscore (\_):

**var DAYS\_UNTIL\_TOMORROW = 1;**

**JAVASCRIPT NAMING CONVENTIONS: PRIVATE**

Rarely you will find an underscore (\_) in front of a variable/function/method in JavaScript. If you see one, it is *intended* to be *private*.

**class SoftwareDeveloper {**

**constructor(firstName, lastName) {**

**this.firstName = firstName;**

**this.lastName = lastName;**

**this.name = \_getName(firstName, lastName);**

**}**

**\_getName(firstName, lastName) {**

**return `${firstName} ${lastName}`;**

**}**

**}**

**var me = new SoftwareDeveloper('Robin', 'Wieruch');**

***// good***

**var name = me.name;**

**console.log(name);**

***// "Robin Wieruch"***

***// bad***

**name = me.\_getName(me.firstName, me.lastName);**

**console.log(name);**

***// "Robin Wieruch"***

-A private variable/function can occur in a JavaScript file as well. -This could mean that the variable/function shouldn't be used outside of this file

**JAVASCRIPT NAMING CONVENTIONS: GLOBAL VARIABLE**

-A JavaScript variable is globally defined, if all its context has access to it. Often the context is defined by the JavaScript file where the variable is declared/defined in, but in smaller JavaScript projects it may be the entire project. There are no special naming conventions for global JavaScript variables.

* A global JavaScript variable is declared at the top of a project/file.
* A global JavaScript variable is written in camelCase if it is mutable.
* A global JavaScript variable is written in UPPERCASE if it is immutable.

**JAVASCRIPT NAMING CONVENTIONS: FILES**

-HTML files should have a **.html** extension (.htm is allowed).

-CSS files should have a **.css** extension.

-JavaScript files should have a **.js** extension.

-There are two strategies of naming files in JavaScript: PascalCase and kebab-case. In JavaScript frontend applications, you will often see PascalCase for naming components (e.g. React components).

**- components/**

**--- user/**

**----- UserProfile.js**

**----- UserList.js**

**----- UserItem.js**

**--- ui/**

**----- Dialog.js**

**----- Dropdown.js**

**----- Table.js**

-In contrast, in JavaScript backend application, kebab-case is the common sense:

**- routing/**

**--- user-route.js**

**--- messages-route.js**

**Spaces Around Operators**

-Always put spaces around operators ( = + - \* / ), and after commas:

**Examples:**

let x = y + z;  
const myArray = ["Volvo", "Saab", "Fiat"];

**Statement Rules**

-General rules for simple statements:

* Always end a simple statement with a semicolon.

**Examples:**

const cars = ["Volvo", "Saab", "Fiat"];  
  
const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};

**Object Rules**

General rules for object definitions:

* Place the opening bracket on the same line as the object name.
* Use colon plus one space between each property and its value.
* Use quotes around string values, not around numeric values.
* Do not add a comma after the last property-value pair.
* Place the closing bracket on a new line, without leading spaces.
* Always end an object definition with a semicolon.

**Examples:**

const person = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};

# **JavaScript Best Practices**

-Avoid global variables, avoid new, avoid ==, avoid eval()

## Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead, and learn how to use [closures](https://www.w3schools.com/js/js_function_closures.asp).

## Always Declare Local Variables

All variables used in a function should be declared as local variables.

Local variables must be declared with the var, the let, or the const keyword, otherwise they will become global variables.

**Declarations on Top**

-It is a good coding practice to put all declarations at the top of each script or function.

This will:

* Give cleaner code
* Provide a single place to look for local variables
* Make it easier to avoid unwanted (implied) global variables
* Reduce the possibility of unwanted re-declarations

// Declare at the beginning  
let firstName, lastName, price, discount, fullPrice;  
  
// Use later  
firstName = "John";  
lastName = "Doe";  
  
price = 19.90;  
discount = 0.10;  
  
fullPrice = price - discount;

**Initialize Variables**

It is a good coding practice to initialize variables when you declare them.

This will:

* Give cleaner code
* Provide a single place to initialize variables
* Avoid undefined values

// Declare and initiate at the beginning  
let firstName = "";  
let lastName = "";  
let price = 0;  
let discount = 0;  
let fullPrice = 0,  
const myArray = [];  
const myObject = {};

**Declare Objects with const**

-Declaring objects with const will prevent any accidental change of type:

### **Example**

let car = {type:"Fiat", model:"500", color:"white"};  
car = "Fiat";      // Changes object to string

### **Example**

const car = {type:"Fiat", model:"500", color:"white"};  
car = "Fiat";      // Not possible

## Declare Arrays with const

Declaring arrays with const will prevent any accidential change of type:

### **Example**

let cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Changes array to number

### **Example**

const cars = ["Saab", "Volvo", "BMW"];  
cars = 3;    // Not possible

## Don't Use new Object()

* Use "" instead of new String()
* Use 0 instead of new Number()
* Use false instead of new Boolean()
* Use {} instead of new Object()
* Use [] instead of new Array()
* Use /()/ instead of new RegExp()
* Use function (){} instead of new Function()

### **Example**

let x1 = "";             // new primitive string  
let x2 = 0;              // new primitive number  
let x3 = false;          // new primitive boolean  
const x4 = {};           // new object  
const x5 = [];           // new array object  
const x6 = /()/;         // new regexp object  
const x7 = function(){}; // new function object

## Beware of Automatic Type Conversions

JavaScript is loosely typed.

A variable can contain all data types.

A variable can change its data type:

### **Example**

let x = "Hello";     // typeof x is a string  
x = 5;               // changes typeof x to a number

-Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

When doing mathematical operations, JavaScript can convert numbers to strings:

### **Example**

let x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number  
let x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string  
let x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string  
let x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number  
let x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number  
let x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number  
let x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

### **Example**

"Hello" - "Dolly"    // returns NaN

## Use === Comparison

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type:

### **Example**

0 == "";        // true  
1 == "1";       // true  
1 == true;      // true  
  
0 === "";       // false  
1 === "1";      // false  
1 === true;     // false

## End Your Switches with Defaults

-Always end your switch statements with a default. Even if you think there is no need for it.

### **Example**

switch (new Date().getDay()) {  
  case 0:  
    day = "Sunday";  
    break;  
  case 1:  
    day = "Monday";  
    break;  
  case 2:  
    day = "Tuesday";  
    break;  
  case 3:  
    day = "Wednesday";  
    break;  
  case 4:  
    day = "Thursday";  
    break;  
  case 5:  
    day = "Friday";  
    break;  
  case 6:  
    day = "Saturday";  
    break;  
  default:  
    day = "Unknown";  
}

-This a common mistake to forget that switch statements use strict comparison:

This case switch will display an alert:

let x = 10;  
switch(x) {  
  case 10: alert("Hello");  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_3)

This case switch will not display an alert:

let x = 10;  
switch(x) {  
  case "10": alert("Hello");  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_4)

# **JavaScript Common Mistakes**

## Expecting Loose Comparison

In regular comparison, data type does not matter. This if statement returns true:

let x = 10;  
let y = "10";  
if (x == y)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_1)

In strict comparison, data type does matter. This if statement returns false:

let x = 10;  
let y = "10";  
if (x === y)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_loose_2)

## Confusing Addition & Concatenation

-Addition is about adding numbers.

-Concatenation is about adding strings.

-In JavaScript both operations use the same + operator.

let x = 10;  
x = 10 + 5;       // Now x is 15  
  
let y = 10;  
y += "5";        // Now y is "105"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_add_1)

When adding two variables, it can be difficult to anticipate the result:

let x = 10;  
let y = 5;  
let z = x + y;     // Now z is 15  
  
let x = 10;  
let y = "5";  
let z = x + y;     // Now z is "105"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_add_1)

## Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

### **Example 1**

let x =  
"Hello World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_1)

But, breaking a statement in the middle of a string will not work:

### **Example 2**

let x = "Hello  
World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_2)

You must use a "backslash" if you must break a statement in a string:

### **Example 3**

let x = "Hello \  
World!";

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_string_3)

## Breaking a Return Statement

-It is a default JavaScript behavior to close a statement automatically at the end of a line.

-Because of this, these two examples will return the same result:

### **Example 1**

function myFunction(a) {  
  let power = 10   
  return a \* power  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_1)

### **Example 2**

function myFunction(a) {  
  let power = 10;  
  return a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_2)

-JavaScript will also allow you to break a statement into two lines.

-Because of this, example 3 will also return the same result:

### **Example 3**

function myFunction(a) {  
  let  
  power = 10;   
  return a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_3)

-But, what will happen if you break the return statement in two lines like this:

### **Example 4**

function myFunction(a) {  
  let  
  power = 10;   
  return  
  a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_4)

-The function will return undefined!

-Why? Because JavaScript thought you meant:

### **Example 5**

function myFunction(a) {  
  let  
  power = 10;   
  return;  
  a \* power;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_mistakes_return_5)

## Explanation

-If a statement is incomplete like:

let

-JavaScript will try to complete the statement by reading the next line:

power = 10;

-But since this statement is complete:

return

-JavaScript will automatically close it like this:

return;

-This happens because closing (ending) statements with semicolon is optional in JavaScript.

-JavaScript will close the return statement at the end of the line, because it is a complete statement.

Never break a return statement.

## Undefined is Not Null

-JavaScript objects, variables, properties, and methods can be undefined.

In addition, empty JavaScript objects can have the value null.

This can make it a little bit difficult to test if an object is empty.

You can test if an object exists by testing if the type is undefined:

### **Example:**

if (typeof myObj === "undefined")

## The Lifetime of JavaScript Variables

-The lifetime of a JavaScript variable starts when it is declared.

-Function (local) variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser -window (or tab).

**Scope Best Practices**

-Scope best practices are essential guidelines for writing clean, efficient, and maintainable code in JavaScript.Below are some key scope best practices to consider:

**Use Local Scope Whenever Possible**

* Limit the use of global variables .
* Encapsulate variables within functions whenever feasible, as local variables are only accessible within the function where they are declared, reducing the risk of unintended side effects and conflicts.

**Avoid Global Variables**

* Minimize the use of global variables to prevent variable pollution and potential conflicts with other parts of the codebase.

**Use Block Scope with let and const**

* Embrace block scope introduced in ES6 using let and const keywords for variable declaration within block statements (e.g., if, for, while).
* Unlike var, let and const have block scope, meaning they are only accessible within the block where they are defined, reducing the risk of unintentional variable hoisting and improving code predictability.

**Avoid Variable Hoisting Pitfalls**

* Be aware of variable hoisting, a behavior in JavaScript where variable declarations are moved to the top of their containing scope during the compilation phase.
* To avoid confusion and potential bugs, declare variables at the beginning of their scope and assign values later in the code.

**Use Strict Mode**

-Enable strict mode (“use strict”) at the beginning of JavaScript files or functions to enforce stricter parsing and error handling. This will help catch common programming mistakes and promote cleaner code practices.

**When to Use Different Types of Scope**

-Understanding when to use different types of scope in JavaScript is crucial for writing efficient and maintainable code. JavaScript offers various types of scope, including global scope, function scope, and block scope, each serving distinct purposes. Here's a guide on when to use each type of scope:

**Global Scope**

* When to Use: Use global scope for variables and functions that need to be accessible from anywhere within the codebase.
* Example Use Cases:
  + Constants or configuration variables that are used throughout the application.
  + Utility functions or libraries that provide standard functionality across different modules or components.
* Considerations: Be cautious when using global scope to avoid polluting the global namespace and potential conflicts with other parts of the codebase.

**Function Scope**

* When to Use: Use function scope for variables and functions only needed within a specific function.
* Example Use Cases:
  + Temporary variables used for calculations or intermediate results within a function.
  + Helper functions or callbacks that are only relevant to a particular function's execution context.
* Considerations: Variables declared within a function are not accessible outside that function, promoting encapsulation and preventing unintended side effects.

**Block Scope (Introduced with let and const)**

* When to Use: Use block scope for variables that are limited in scope to a specific block of code, such as if statements, for loops, or while loops.
* Example Use Cases:
  + Loop counters or iterators that are only relevant within a loop.
  + Temporary variables are used for conditionally executing code blocks.
* Considerations: Block-scoped variables are only accessible within the block where they are declared, providing tighter control over variable visibility and lifetime.

**Lexical Scope**

* When to Use: Lexical scope is inherent in JavaScript and is automatically applied based on the physical placement of code within the source file.
* Example Use Cases:
  + Variables declared in outer scopes are accessible to nested functions, allowing for closures and maintaining access to variables even after the outer function has returned.
* Considerations: Lexical scope enables powerful programming patterns such as closures, which can encapsulate state and create modular and reusable code.

-In JavaScript, objects and functions are also variables.

**JAVASCRIPT VARIABLES**

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

**VARIABLES**

-Variables are containers for storing information.

-Variable is a Block of Container used to store the value.

-Variable is a Block of Memory used to store the value.

-Variable are **Case – Sensitive**.

**1.Variable Declaration**

-Creating a variable syntax: -

**VariableType VariableName = Value**

//var/let/const carName = “BMW “

**e.g.: -var carName=”BMW”**;

**1.Variable Declaration**

-Creating a variable in JavaScript is called "declaring" a variable:

**var carName**;

**2. Variable Initialization**

-After the declaration, the variable is empty (it has no value).

-To assign a value to the variable, use the equal sign:

**var carName**; //Declaration

**carName = "Volvo";**//Initailization

**3.Variable Declaration with Initialization.**

-You can also assign a value to the variable when you declare it:

**var carName = "Volvo";**//Declaration with Initialization

**4.Re-Initialization**

**var carName = "Volvo";**//Declaration with Initialization

**carName = “BMW”;**//Re-Initialization

**5.Re-Declaration**

**var carName = "Volvo";**//Declaration with Initialization

**var carName = “BMW”;**//Re-Declaration

**TYPES OF VARAIBLES**

-There are Three types of Variables in JavaScript.

**1]var 2] let 3] const**

**1]var**

-var has a **Global/Function** Scope Variable

-In Var **Declaration** of Variable is **Possible**.

-In Var **Declaration with Initialization** is also **Possible**.

-In Var **Re-Declaration** is also **Possible**.

-In Var **Re-Initialization** is also **Possible**.

-In Var **Hoisting** is also **Possible**.

**2]let**

-let has a **Block Scope** Variable.

-In let **Declaration** of Variable is **Possible**.

-In let **Declaration with Initialization** is also **Possible**.

-In let **Re-Declaration** of Variable is **Not Possible**.

-In let **Re-Initialization** of Variable is also **Possible**.

-In let **Hoisting** is **Not Possible**

**3]const**

-const has a **Block / Local** Variable.

-In const **Declaration** of variable is **Not Possible**.

-In const **Declaration** with Initialization is **Possible**.

-In const **Re-Declaration** of Variable is Not Possible.

-in const **Re-Initialization** of Variable is **Not** **Possible**.

-In Const **Hoisting** is **Not Possible**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Difference Between Var, Let, Const | | | | |
| Sr.No | **Parameters** | **var** | **let** | **const** |
| 1. | Scope | Global | Block / Local | Block |
| 2. | Declaration | Checkmark with solid fill | Checkmark with solid fill | Close with solid fill |
| 3. | Declaration-Initailization | Checkmark with solid fill | Checkmark with solid fill | Checkmark with solid fill |
| 4. | Re-Declaration | Checkmark with solid fill | Close with solid fill | Close with solid fill |
| 5. | Re-Initialization | Checkmark with solid fill | Checkmark with solid fill | Close with solid fill |
| 6. | Hoisting | Checkmark with solid fill | Close with solid fill | Close with solid fill |

**JavaScript Display Possibilities**

-JavaScript can "display" data in different ways:

* Writing into an HTML element, using innerHTML.
* Writing into the HTML output using document.write().
* Writing into the HTML output using document.writeln().
* Writing into an alert box, using window.alert().
* Writing into the browser console, using console.log().

**1]innerHTML**

-We can write Text From JavaScript also for that we have take help of geElementId() Method for to access HTML Element.

-Then after write **.innerHTML=”HELLO THIS INNERHTML”**

**document.getElementById().innerHTML=”HELLO THIS INNERHTML”**

<body>

    <h1 id="Demo"></h1>

    <script>

        document.getElementById("Demo").innerHTML="HELLO THIS IS WRITING BY USING INNER HTML"

    </script>

</body>

**2]innerText**

-Using innerText we can write or fetch the text inside in the HTML elements.

**document.getElementById().innerText=”HELLO THIS INNERHTML”**

<body>

    <h1 id="Head1"></h1>

    <script>

        document.getElementById("Head1").innerText = "This is innerText";

    </script>

</body>

**3]textContent**

**document.getElementById().textContent=”HELLO THIS INNERHTML”**

<body>

    <h1 id="Head1"></h1>

    <script>

        document.getElementById("Head1").textContent = "This is innerText";

    </script>

</body>

**4]Document.write(“”)**

-In JavaScript, the document object represents the entire HTML document loaded in the browser window. It serves as the root node of the DOM (Document Object Model) and provides access to all elements, attributes, and content within the page.

-For that we write **document** object and call **write(“ ”)** method.

**document.write("Hello This Text is Rendering By using Write Method")**

    <script>

        document.write("Hello This Text is Rendering By using Write Method")

    </script>

**5]Document.writeln(“”)**

-Writeln(“ “) method only increase the readability of the code.

-It adds the Space between word.

-For that we write **document** object and call **writeln(“ ”)** method.

    <script>

       document.writeln("Hello This Text is Rendering By using WriteIn Method")

    </script>

**DIFFERENCE BETWEEN WRITE() and WRITElN()**

**document.write():**

Writes a string of text to the document without adding any newline characters.

**document.writeln():**

-Writes a string of text to the document and appends a newline character (\n) after it.

-When writing to an HTML document, this nearly never makes a difference. Writeln only increases the readability of the code.

    <script>

    document.write("Hello");

    document.write("World!");

    document.writeln("This")

    document.write("<br>")

    document.writeln("Hello");

    document.writeln("World!");

    document.writeln("This")

    </script>

**NOTE : -**because of windows.document.write(), windows.document. writeln() methods are the problem...if we want to break the line we have to use **br, \n, or pre tag** or you can use the para tag in the script.

-We have no option with writeln() method to break the line.

**6] Using console.log()**

-For debugging purposes, you can call the console.log() method in the browser to display data.

-It displays only on Console Tab Of the Browser.

    <script>

        console.log("This Text is rendered By From Console Log ()")

    </script>

**Note:-There are also More Ways To Print.(alert(),confirm());**

**JAVASCRIPT SCOPE**

-Scope determines the accessibility (visibility) of variables.

JavaScript variables have 3 types of scope:

* Block scope
* Function scope
* Global scope
* Local Scope
* Lexical Scope

**1.Global Scope**

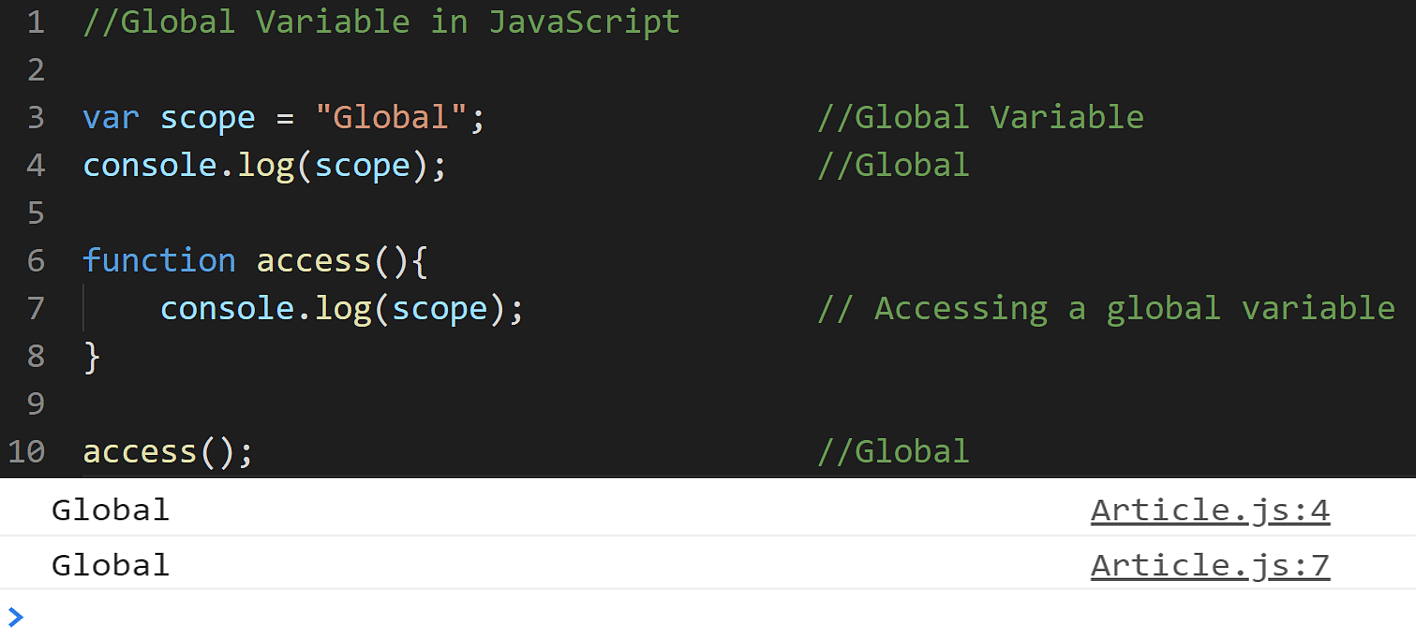
-Any variable declared outside of a function is said to have **Global Scope.**

-Global variables can be accessed from anywhere in a JavaScript program.

-Globally scoped variables can be defined using any of the three keywords: let, const, and var.

-A global variable has **Global Scope**:

-All scripts and functions on a web page can access it.



-Variables declared with var, let and const are quite similar when declared outside a block.

They all have Global Scope:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <script>

        var a=10;//Global Scope

        let b=20; //Global Scope

        const c=30; //Global Scope

        console.log(a);

        console.log(b);

        console.log(c);

    </script>

</body>

</html>

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! Global variables defined with the let,Const keyword do not belong to the window object: -->*

    <script>

        var car="Fortuner";

        let Mobile="Iphone15";

        const Bike="NS400";

          function Global(*params*) {

            console.log(window.car);

            console.log(window.Mobile);

            console.log(window.Bike);

        }

        Global();

    </script>

</body>

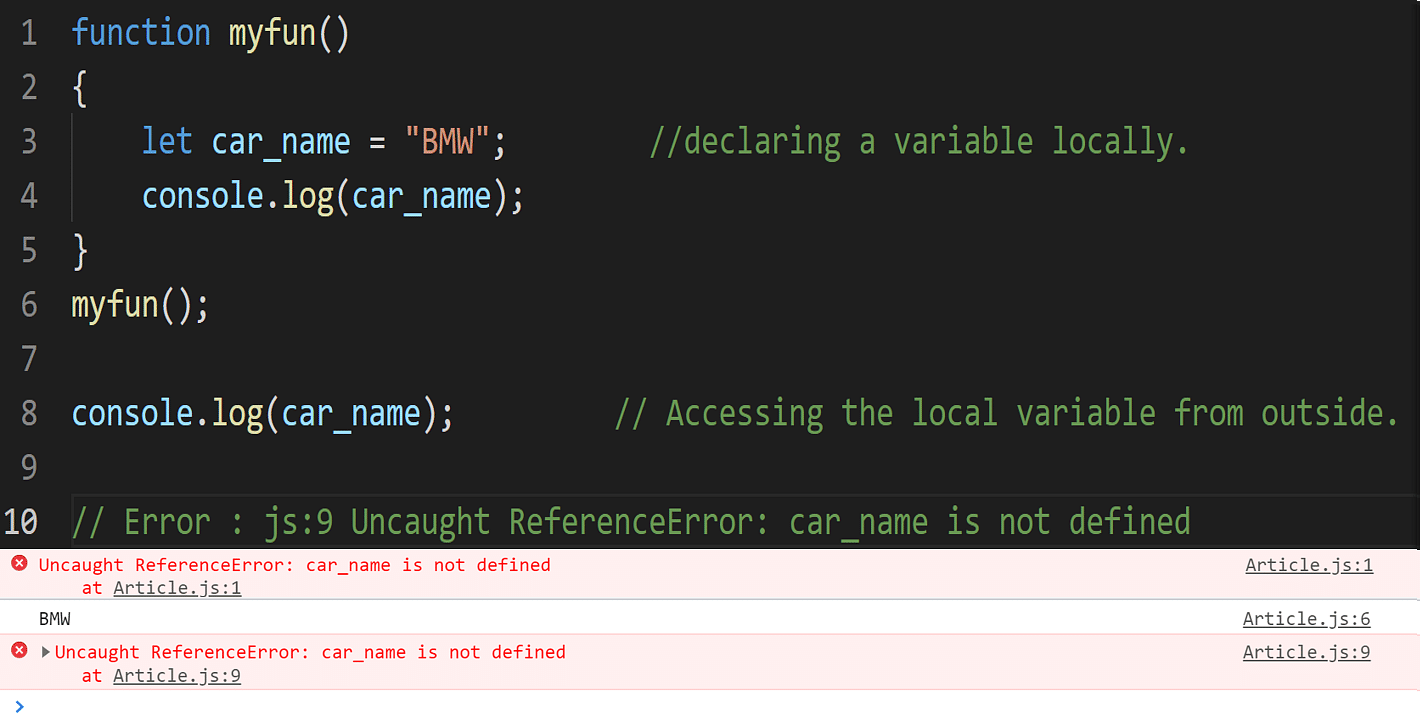
</html>

**2.Local Scope?**

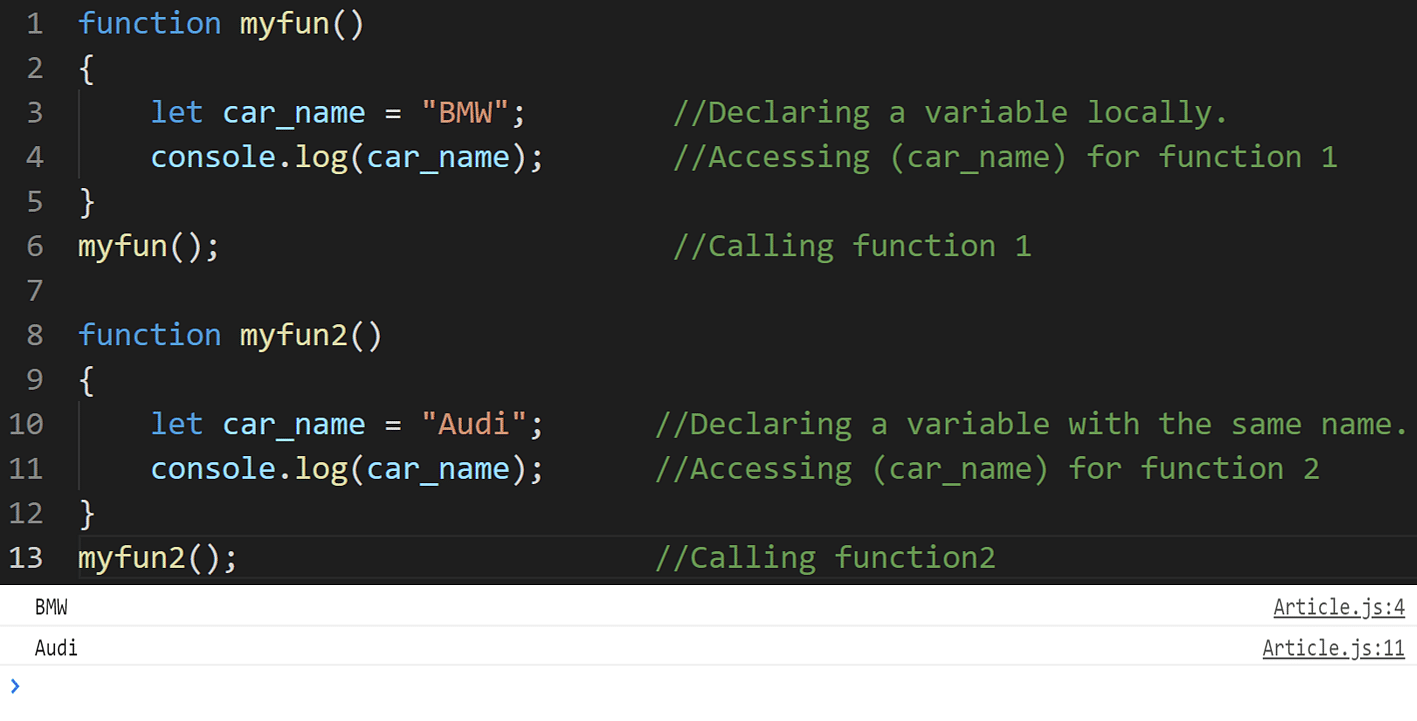
**-**Any variable that you declare inside a function is said to have **Local Scope**. You can access a local variable can within a function. If you try to access any variable defined inside a function from outside or another function, it throws an error.

-**Local** **variables** have **Function** Scope:

-They can only be accessed from **within** the **function**.



-Since you cannot access a local variable from outside the function, you can have a variable of the same name in another function as well.



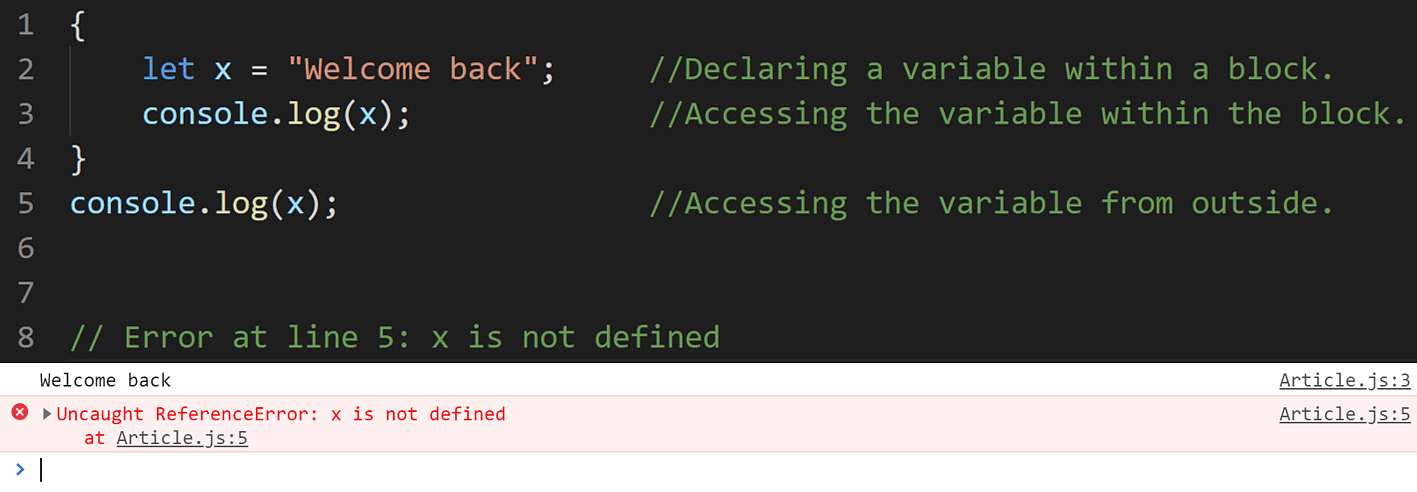
**3.Block Scope{}**

-Before ES6 (2015), JavaScript variables had only Global Scope and Function Scope.

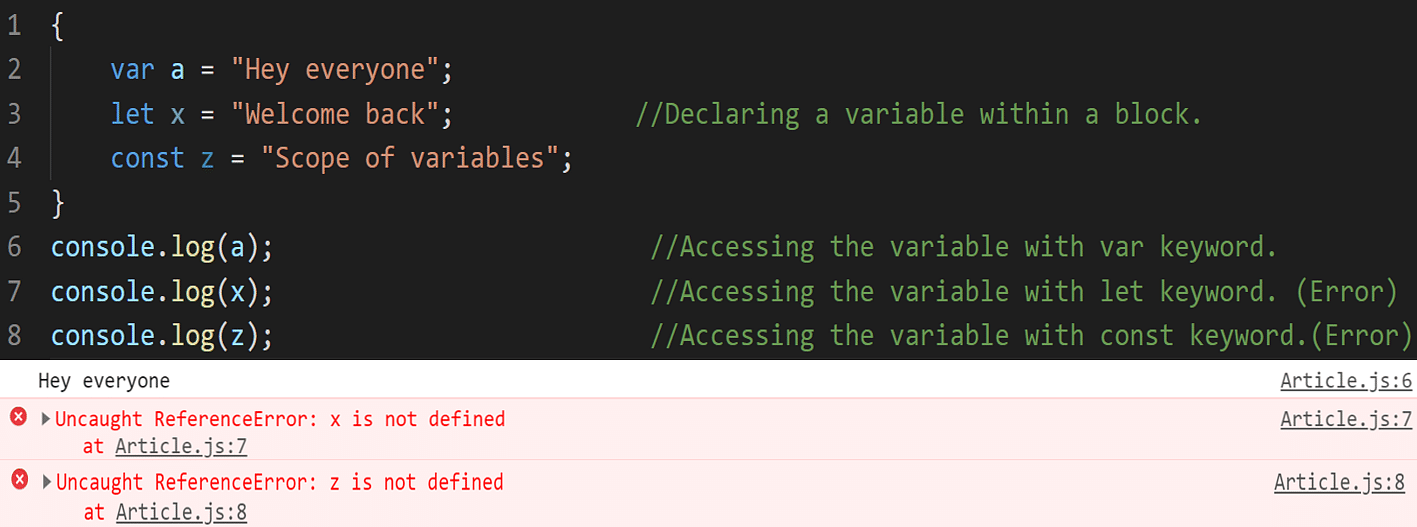
-ES6 introduced two important new JavaScript keywords: **let** and **const**.

-These two keywords provide Block Scope in JavaScript.

-Variables declared inside a **{ }** block cannot be accessed from outside the block:



-The block scope does not work with the var keyword. You can either use let or const keywords for that.

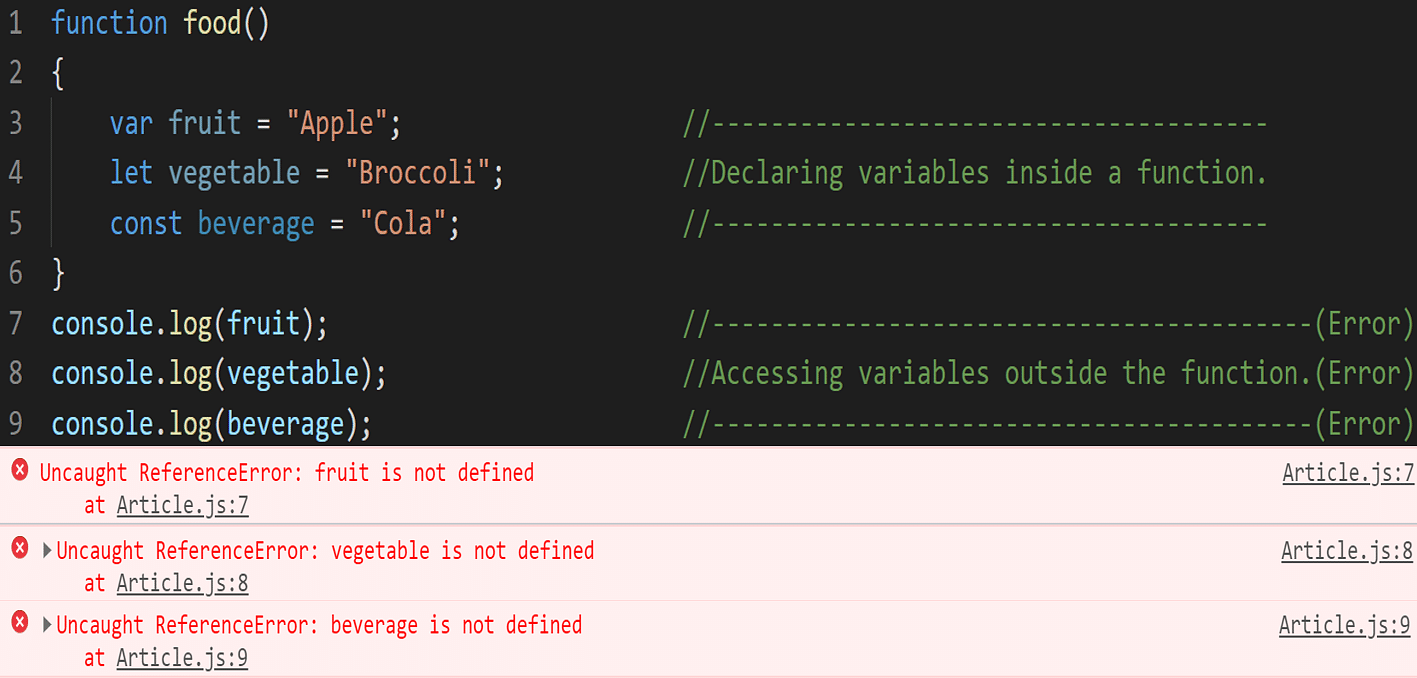


**4.Function Scope**

**-**With the creation of each new function, it creates a new scope in JavaScript. You cannot access variables defined inside a function from outside the function or from another function. Var, let, and const work similarly when used inside a function.

-JavaScript has function scope: Each function creates a new scope.

They all have Function Scope:



**5.Lexical Scope**

* Lexical: Refers to the structure of the code, specifically the nesting of functions and blocks.
* How it works: In JavaScript, the scope of a variable is determined by where it is declared within the nested structure of your code. A variable declared inside a function is only accessible within that function and any nested functions within it.

function outer() {  
 let x = 10;  
  
 function inner() {  
 console.log(x); *// Can access 'x' from the outer function*  
 }  
  
 inner();   
}  
  
outer();   
console.log(x); // Error: 'x' is not defined

Key Points:

* Inner functions can access variables from their outer functions.
* Variables declared outside any function are global.
* let and const introduce block scope, meaning variables declared with these keywords are only accessible within the block they are defined (e.g., inside an if statement or for loop).
* Closures: are a powerful feature of JavaScript that rely on lexical scoping. A closure is a function that retains access to its outer function's scope even after the outer function has finished executing.

**JavaScript Strict Mode for Defining Scope of a Variable**

-In JavaScript, if you forget to declare a variable with a keyword: var, let, and const, JavaScript automatically takes it as a global variable, and you can access it anywhere in the program.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! AUTOMATICALLY GLOBAL -->*

*<! --\* If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable: -->*

    <p id="demo"></p>

<script>

    myFunction ();

*// code here can use carName as a global variable*

    document. getElementById("demo").innerHTML = "I can display " + carName;

    function myFunction() {

    carName = "Volvo";

    }

    console.log(carName);

</script>

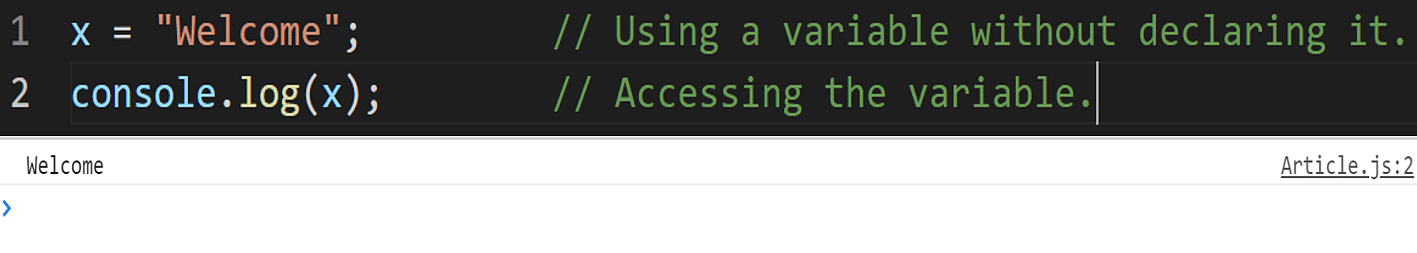
</body>

</html>

## Strict Mode

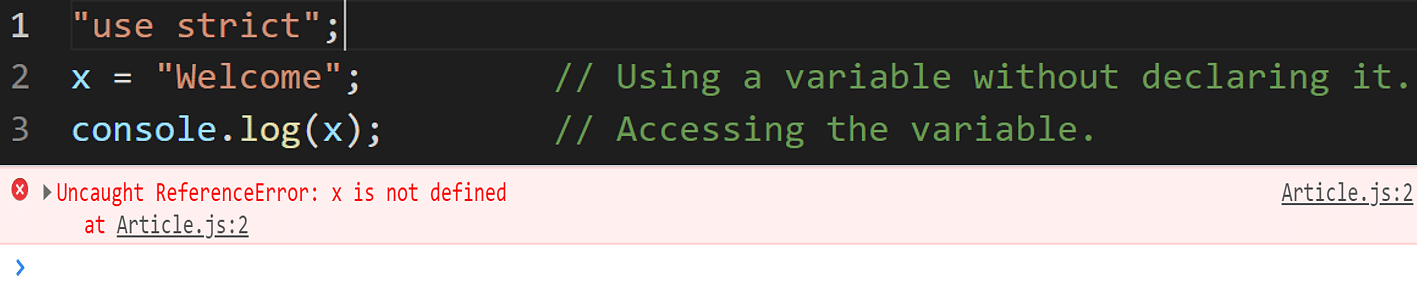
All modern browsers support running JavaScript in "Strict Mode".

In "Strict Mode", undeclared variables are not automatically global.

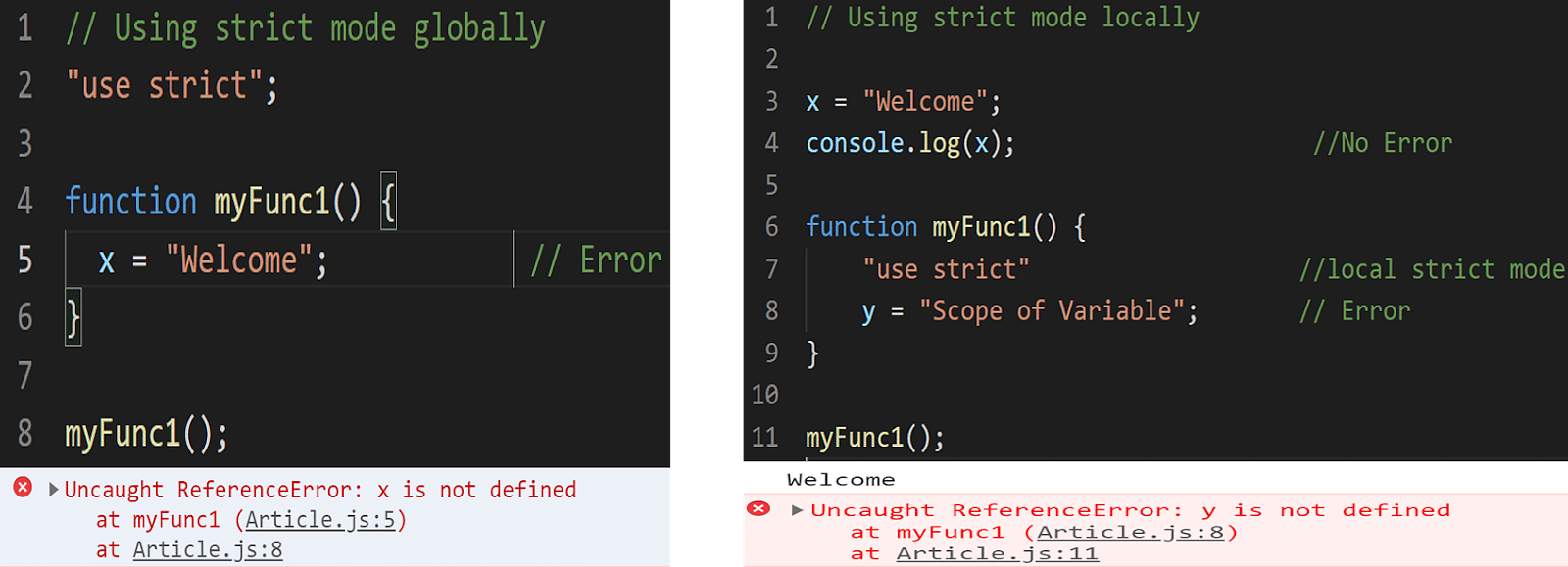


-To avoid such mistakes and confusion, the strict mode was introduced in JavaScript with ES5 (ECMAScript 5) in the year 2009.

* Strict mode will throw an error if you try to use the same syntax after including "strict mode" in your program. It will help you in writing cleaner and more secure code.



* Just like variables, strict mode too can be used either globally or locally. If you write "use strict" at the program's start, it will be used globally. Otherwise, you can also use strict mode locally inside a function.



**Warning**

-Do NOT create global variables unless you intend to.

-Your global variables (or functions) can overwrite window variables (or functions).  
-Any function, including the window object, can overwrite your global variables and functions.

**HOISTING**

- Hoisting is JavaScript's default behavior of moving declarations to the top.

-It is the process of before Declaration Variable we can Initialize it.

-In JavaScript, a variable can be declared after it has been used.

-In other words; a variable can be used before it has been declared

- Hoisting is works with **Variables**(var,let,const).

-When there no type of variables are not declared

-JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables, classes, or imports to the top of their scope, prior to execution of the code.

console.log(x); // undefined

var x = 5;

-This code behaves as if it were interpreted like this:

var x; // Declaration is hoisted

console.log(x); // undefined

x = 5; // Initialization remains here

**Function Hoisting:**

-Function declarations are also hoisted similarly, meaning the entire function declaration is moved to the top of the scope. For example:

foo(); // "Hello, world!"

function foo() {

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

}

-Here, foo() can be called before its declaration in the code because the function declaration itself is hoisted to the top of its scope.

**Hoisting Doesn't Move Code:**

-It's important to note that hoisting doesn't physically move any code in your script. It's a concept that helps explain the behavior of variable and function declarations during the compilation phase of JavaScript code execution. The actual code you write remains in the same order you wrote it; only the declarations are conceptually moved to the top.

**Function Declarations vs. Function Expressions:**

Function declarations are fully hoisted, meaning the entire function body is moved to the top of its containing scope. This allows functions to be called before they are declared in the code.

foo(); // "Hello, world!"

function foo() {

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

}

-However, function expressions (where a function is assigned to a variable) are not hoisted in the same way:

bar(); // TypeError: bar is not a function

var bar = function() {//Here bar is declared with var

console.log("This won't work!");

};

-In this case, bar is hoisted as a variable declaration (var bar;), but its assignment to a function expression remains where it is in the code, resulting in a TypeError when trying to call bar() before it's assigned.

**Hoisting Behaviour of let and const:**

**Temporal Dead Zone (TDZ)**:

-Variables declared with let and const are hoisted to the top of their block scope (just like var), but they are not initialized.

-Accessing these variables before their declaration (initialization) results in a ReferenceError. This period between the hoisting and the actual initialization is called the Temporal Dead Zone (TDZ).

**Example**:

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

let x = 10;

-In this example, x is hoisted to the top of its scope (global or function scope), but it's not initialized. Trying to access x before the initialization (let x = 5;) results in a ReferenceError due to the TDZ.

**Benefits**:

-Hoisting let and const declarations allows you to understand the scope where they are available, similar to var.

-The Temporal Dead Zone (TDZ) behavior ensures that variables are properly declared and initialized before being used, helping to catch potential issues early.

**Correct Usage**:

-To avoid TDZ errors, ensure that you only access let and const variables after they have been properly declared and initialized.

**Additional Example Showing TDZ in a Conditional Block:**

let y = 20;

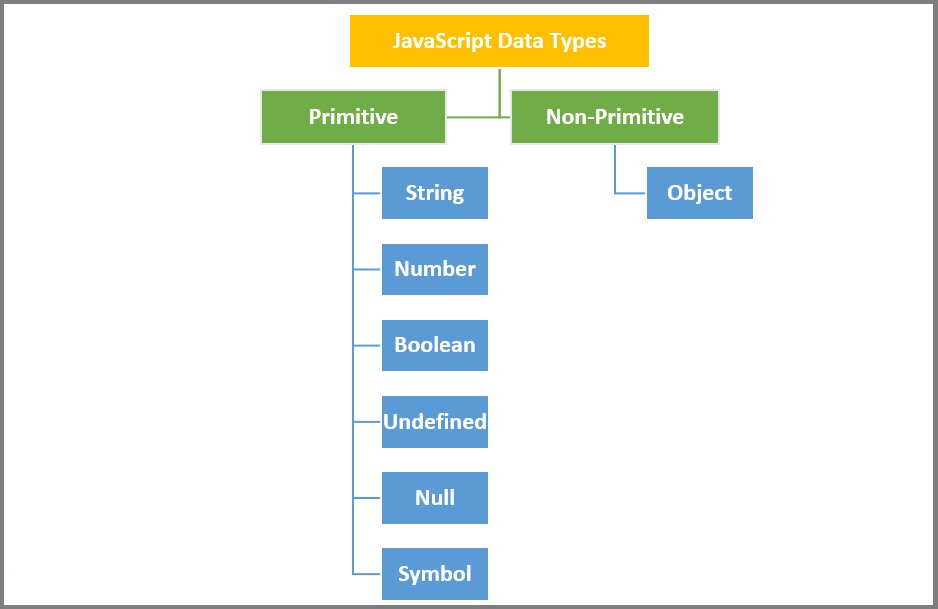
if (true) {

console.log(y); // Throws ReferenceError

let y = 30;

}

**DATATYPES**



- Data types in JavaScript refers to the types of the values that we are storing or working with.

-Datatype is the Type of the data.

-Datatype is the data storage format that contains specific type of data.

-Datatype contains different types of values we used in programming.

-Data Types or Types are attributes that are predefined or can be created by the user so that the program can easily detect the different types of information.

[**Dynamic and weak typing**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#dynamic_and_weak_typing)

-JavaScript is a [dynamic](https://en.wikipedia.org/wiki/Dynamic_programming_language) language with [dynamic types](https://en.wikipedia.org/wiki/Type_system#DYNAMIC). Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types:

let foo = 42; // foo is now a number

foo = "bar"; // foo is now a string

foo = true; // foo is now a boolean

foo= null; // foo is now a null

There are altogether **8** basic data types in JavaScript.

|  |  |  |
| --- | --- | --- |
| Data Type | Description | Example |
| String | Textual data. | 'hello', "hello world!", etc. |
| Number | An integer or a floating-point number. | 3, 3.234, 3e-2, etc. |
| BigInt | An integer with arbitrary precision. | 900719925124740999n, 1n, etc. |
| Boolean | Any of two values: true or false. | true and false |
| undefined | A data type whose variable is not initialized. | let a; |
| null | Denotes a null value. | let a = null; |
| Symbol | A data type whose instances are unique and immutable. | let value = Symbol('hello'); |
| Object | Key-value pairs of collection of data. | let student = {name: "John"}; |

-In JavaScript there are two types of Datatypes:-

**1]Primitive Datatype**

**2]Non-Primitive Datatype**

**Note:** JavaScript data types are divided into primitive and non-primitive types.

* **Primitive Data Types:-** They can hold a single simple value. String, Number, BigInt, Boolean, undefined, null, and Symbol are primitive data types.
* **Non-Primitive Data Types:-** They can hold multiple values. Objects are non-primitive data types.

**1]Primitive Datatype**

-The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types.

-Primitive values are immutable

-Primitive values compared by value

|  |  |  |
| --- | --- | --- |
| Type | typeof return value | Object wrapper |
| [Null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#null_type) | "object" | N/A |
| [Undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#undefined_type) | "undefined" | N/A |
| [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#boolean_type) | "boolean" | [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) |
| [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#number_type) | "number" | [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) |
| [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#bigint_type) | "bigint" | [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) |
| [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#string_type) | "string" | [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) |
| [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#symbol_type) | "symbol" | [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) |

-All primitive types, except [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null), can be tested by the [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) operator. typeof null returns "object", so one has to use === null to test for null.

**1.STRING**

-String Are Immutable In Nature.

-String Is Primitive Datatype in JavaScript.

-The sequence of character is called as String.

-Strings are for **storing text**

-Strings are written **with quotes** ( **“ “** )

**-We write String in following ways: -**

**1." String "**

**2.' String '**

**3.` String `**

**e.g.: -**

|  |  |
| --- | --- |
| Single Quotes | 'Hello' |
| Double Quotes | "Hello" |
| Backticks | `Hello` |

**-We Can Write Also**

**1." ' String ' "**

**2." ` String ` "**

-Not Allowed To String in Following Ways: -

1. " " " " It is Not Allowed because same Sign is Dominated to each other

2. ` ` ` ` It is Not Allowed because same Sign is Dominated to each other

3. ' ' ' ' It is Not Allowed because same Sign is Dominated to each other

**e.g.: -" " String " " it is not allowed**

-We can create the String using Two Ways

**1.Using Literals**

**2.Using New Keyword**

**1.Using Literals**

*<!--! STRING -->*

   <script>

      let str1 = "Hello World!"; *// Using double quotes*

      let str2 = 'Hello World!'; *// Using single quotes*

      let str3 = `Hello World!`; *// Using backticks*

      document.write(str1 + "<br>");

      document.write(str2 + "<br>");

      document.write(str3 + "<br>");

      console.log(typeof str1);

      console.log(typeof str2);

      console.log(typeof str3);

   </script>

-In this we have create variable and directly pass the Values.

-We can pass in three ways as shown in above example.

**2.By Using New Keyword**

<script>

        let str = new String("GANGADAR");

        let str1 = new String('Hey');

        let str2 = new String(`Hello`)

        console.log(str)

        console.log(str.length);

    </script>

-In this case we have to create the variable.

-Then we have write the New Keyword.

-After that call the String Constructor.

-Then pass the String value in the String Class Constructor.

**Length Property**

- To find the length of a string, use the built-in length property:

<script>

        let str = new String("GANGADAR");

        console.log(str.length)

   </script>

**Escape Characters**

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

|  |  |  |
| --- | --- | --- |
| Code | Result | Description |
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

**1.Insert A Single Coat.**

<script>

        let text= 'It\'s alright.';

        document.write(text);

    </script>

**O/P:-** It's alright.

**2.Insert Double Coat in String**

 <script>

        let text = "We are the So called \"Vikings\" From the North.";

        document.write(text);

    </script>

**O/P:-**We are the So called "Vikings" From the North.

**3.Insert BackSlash in String**

    <script>

        let text = "The character \\ is called backslash.";

        document.write(text);

    </script>

**O/P:-** The character \ is called backslash.

Six other escape sequences are valid in JavaScript:

|  |  |
| --- | --- |
| Code | Result |
| \b | Backspace |
| \f | Form Feed |
| \n | New Line |
| \r | Carriage Return |
| \t | Horizontal Tabulator |
| \v | Vertical Tabulator |

-These other Escape Sequence character works only Browsers Console.

**Access String Characters**

-One way is to treat strings as an array and access the character at the specified index. For example,

<script>

        let message = "hello";

*// use index 1 to access*

*// 2nd character of message*

        console.log(message[1]);  *// e*

    </script>

**Features of JavaScript Strings**

**1. JavaScript Strings are Immutable**

*<!--! String are immutable in Nature  -->*

    <script>

        let message = "hello";

        message[0] = "H";

        console.log(message);  *// hello*

    </script>

-You cannot change the String because it is Immutable in Nature.

-But you achieve that output by assigning new Value.

*<!--\* you cannot change value of String But you can assign new Value in String -->*

    <script>

        let message = "hello";

        message = "Hello";

        console.log(message);  *// Hello*

    </script>

**2. JavaScript Strings are Case-Sensitive**

In JavaScript., the lowercase and uppercase letters are treated as different values. For example,

<script>

        let value1 = "a";

        let value2 = "A"

        console.log(value1 == value2);  *// false*

    </script>

**STRING BASIC METHODS**

**There are 4 methods for extracting string characters:**

* The at(position) Method
* The charAt(position) Method
* The charCodeAt(position) Method
* Using property access [] like in arrays

**1.charAt()**

-The charAt() method returns the character at a specified index (position) in a string:

  <script>

        let str = "Amol";

        console.log(str.charAt(0)); *// A*

    </script>

## Note

The at() method is a new addition to JavaScript.

It allows the use of negative indexes while charAt() do not.

**2.charCodeAt()**

-The charCodeAt() method returns the code of the character at a specified index in a string:

 <script>

        let text = "Amol";

        console.log(text.charCodeAt(0)); *// 65*

    </script>

*<!--\* It cannot accepts the Negative Values.  -->*

    <script>

        let text = "HelloWorld";

        console.log(text.charAt(-2)); *//Not Works With Negative Values*

    </script>

-Note: It cannot accepts the Negative Values

**3.at()**

-The at() method returns the character at a specified index (position) in a string.

-It accepts **Negative** Values.

-[ES2022](https://www.w3schools.com/js/js_2022.asp) introduced the string method at():

   <script>

        let text = "Amol";

        console.log(text.at(2)); *// o*

        console.log(text.at(-1)); *// l*

        console.log(text.at(length-3))*//m*

    </script>

**4.[0]**

* -It makes strings look like arrays (but they are not)
* If no character is found, [ ] returns undefined, while charAt() returns an empty string.
* It is read only. str[0] = "A" gives no error (but does not work!)

<script>

        let message = "Hello "

        console.log(message[0]);*//h*

    </script>

**Extracting String Parts**

There are 3 methods for extracting a part of a string:

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

**5.slice(start,end)**

-slice() extracts a part of a string and returns the extracted part in a new string.

-The method takes 2 parameters: start position, and end position (end not included).

**Note: The slice() method does not change the original string.**

    <script>

        let text = "Let's Learn JavaScript."

        console.log(text.slice(6));*//Learn JavaScript.*

        console.log(text.slice(12,22)); *//JavaScript*

        console.log(text.slice(-11)); *//JavaScript.*

        console.log(text.slice(-11,-1)) *//JavaScript*

    </script>

-It accepts the Negative Values As well.

**6.subStr()**

-substr() is similar to slice().

-The difference is that the second parameter specifies the **length** of the extracted part.

Syntax:

subStr(start-index, define-length)

* **start-index:-**Define the Start index when you start.
* **Define-length:-**After define the length of string that from starting point you want to extracted.

 <script>

        let text = "Let's Learn JavaScript."

        console.log(text.substr(6))

        console.log(text.substr(6,5))

        console.log(text.substr(-11,11))

    </script>

**7.subString()**

**Notes:**

* Any **argument value < 0** is treated as **0**.
* Any **argument value > str.length** is treated as **str.length**.
* Any NaN argument value is treated as **0**.
* If indexStart is greater than indexEnd, the two arguments are swapped, i.e. str.substring(a, b) will be str.substring(b, a).

<script>

        let text = "Lets's Start JavaScript."

        console.log(text.substring(5));*//s Start JavaScript.*

        console.log(text.substring(-5));*//Lets's Start JavaScript.*

        console.log(text.substring(35,5))*//s Start JavaScript.*

    </script>

**8.toUpperCase**

-The toUpperCase() method returns the [string](https://www.programiz.com/javascript/string) converted to uppercase.

-The toUpperCase() method does not take in any parameters.

    <script>

        let text = "hey hello..."

        document.write(text+"</br>");*//hey hello...*

        document.write(text.toUpperCase())*//HEY HELLO...*

    </script>

<script>

        let str=undefined;

        console.log(strt.toUpperCase());*//TypeError:Cannot read property*

    </script>

**Notes:**

* The toUpperCase() method raises TypeError when called on null or undefined.
* The toUpperCase() method does not change the original string.

**9.toLowerCase()**

   <script>

        let text = "LET'S START PROGRAMMING.";

        console.log(text.toLowerCase())//let's start programming.

    </script>

**Notes**

* The toLowerCase() method raises TypeError when called on null or undefined.
* The toLowerCase() method does not change the original string.

**10.concat()**

-concat() joins two or more strings:

-It takes two parameter:

concat(“ “,stringAsAnArgument)

* **First(optinal):-**Inside this if you want space or any sign to be for string separation you can pass as an First Argument.
* **String:-**inside this pass the String Variable as an argument want to be concatenated.

    <script>

        let str1 = "Hello";

        let str2 = "World";

        console.log(str1.concat(str2));*//HelloWorld*

        console.log(str1.concat(",",str2));*//Hello,World*

        console.log(str1.concat("...",str2));*//Hello...World*

    </script>

**11.trim()**

- The trim() method removes whitespace from both ends of a [string](https://www.programiz.com/javascript/string).

- The trim() method does not take in any parameters.

    <script>

        let message = "    JAVASCRIPT IS FUN    ";

        console.log(message);*//    JAVASCRIPT IS FUN*

        console.log(message.trim());*//JAVASCRIPT IS FUN*

    </script>

**12.trimStart()**

[ECMAScript 2019](https://www.w3schools.com/js/js_2019.asp) added the String method trimStart() to JavaScript.

The trimStart() method works like trim(), but removes whitespace only from the start of a string.

    <script>

        let text = "     Hello World!     ";

        console.log(text);*//     Hello World!     ;*

        console.log(text.trimStart());*//Hello World!    ;*

    </script>

**13.trimEnd()**

-[ECMAScript 2019](https://www.w3schools.com/js/js_2019.asp) added the string method trimEnd() to JavaScript.

-The trimEnd() method works like trim(), but removes whitespace only from the end of a string.

    <script>

        let text = "     Hello World!     ";

        console.log(text);*//     Hello World!     ;*

        console.log(text.trimEnd())*//     Hello World!;*

    </script>

**14.spilt()**

-A string can be converted to an array with the split() method:

-If you want to work with a string as an array, you can convert it to an array.

**Note:**The split() method does not change the original string.

**split() Parameter**

**The split() method takes in:**

* separator (optional) - The pattern (string or regular expression) describing where each split should occur.
* limit (optional) - A non-negative integer limiting the number of pieces to split the given string into.

-If the separator is "", the returned array will be an array of single characters as an Array.

### **Example**

1.text.split(",")    // Split on commas  
2.text.split(" ")    // Split on spaces  
3.text.split("|")    // Split on pipe

    <script>

        const text = "Java is awesome. Java is fun.";

        console.log(text.split(""));//['J', 'a', 'v', 'a', ' ', 'i', 's', ' ', 'a', 'w', 'e', 's', 'o', 'm', 'e', '.', ' ', 'J', 'a', 'v', 'a', ' ', 'i', 's', ' ', 'f', 'u', 'n', '.']

        console.log(text.split("."));// ['Java is awesome', ' Java is fun', '']

        console.log(text.split(".", 2))//['Java is awesome', ' Java is fun']

    </script>

**15.repeat()**

-The repeat() method returns a string with a number of copies of a string.

-The repeat() method returns a new string.

-The repeat() method does not change the original string.

-The repeat() method creates a new [string](https://www.programiz.com/javascript/string) by repeating the given string a specified number of times and returns it.

**The repeat() method takes in :**

* count - An integer between 0 and +Infinity, indicating the number of times to repeat the string.

<script>

        const holiday = "Happy holiday!";

        console.log(holiday.repeat(7));

    </script>

**Note:**repeat() raises RangeError if repeat count is negative, infinity, or overflows maximum string size.

**16.replace()**

-The replace() method replaces a specified value with another value in a string:

-The replace() method does not change the string it is called on.

-The replace() method returns a new string.

-The replace() method replaces only the first match

-If you want to replace all matches, use a regular expression with the /g flag set.

The syntax of replace() is:

str.replace(pattern, replacement)

**The replace() method takes in:**

* pattern - either a string or a regex that is to be replaced
* replacement - the pattern is replaced with this replacement (can be either a string or a function)

1.It will replace first caught string after that i will not replace.

   <script>

        const text = "Java is awesome. Java is fun."

        console.log(text.replace("Java","JavaScript"));*//JavaScript is awesome. Java is fun.*

        console.log()

    </script>

2.By default, the replace() method is case sensitive. Writing MICROSOFT (with upper-case) will not work:

 <script>

        let message = "Please visit Microsoft!";

        let newText = message.replace("MICROSOFT", "W3Schools");

        console.log(newText);

    </script>

3**.**To replace case insensitive, use a regular expression with an /i flag (insensitive):

 <script>

        let message1 = "Please visit Microsoft!";

        let newText1 = message.replace(/MICROSOFT/i, "W3Schools");

        console.log(newText1);

    </script>

4.To replace all matches, use a regular expression with a /g flag (global match)

 <script>

        const text2 = "Java is awesome. Java is fun."

        console.log(text2.replace("/Java/g","JavaScript"));*//JavaScript is awesome. Java is fun.*

        console.log()

    </script>

**17.replaceAll()**

-The replaceAll() method returns a new [string](https://www.programiz.com/javascript/string) with all matches of a pattern replaced by a replacement.

The syntax of replaceAll() is:

str.replaceAll(pattern, replacement)

**replaceAll() Parameter**

* **pattern** - either a substring or a [regex](https://www.programiz.com/javascript/regex) that is to be replaced
* **replacement** - the pattern is replaced with this replacement (can be either a string or a [function](https://www.programiz.com/javascript/function))

1.The replaceAll() method returns a new string with all matches of a pattern replaced by a replacement.

    <script>

        const message = "ball bat";

        console.log(message.replaceAll("b","c"));

    </script>

2.The replaceAll() method is case sensitive

 <script>

    const text = "Java is awesome. Java is fun.";

    console.log(text.replaceAll("JAVA","JavaScript"))

    console.log(text.replaceAll(/JAVA/gi,"JavaScript"))

    </script>

**Note:**A RegExp without the global ("g") flag will throw a TypeError

**18.padStart()**

-The padStart() method pads a string from the start.

It pads a string with another string (multiple times) until it reaches a given length.

The syntax of the padStart() method is:

str.padStart(targetLength, padString)

* targetLength - The length of the final string after the current string has been padded.
* padString (optional) - The string to pad the current string with. Its default value is " ".

  <script>

        let string1 = "CODE";

        console.log(string1.padStart(10,"$"))*//$$$$$$CODE*

    </script>

**19.padEnd**

-The padEnd() method pads the current [string](https://www.programiz.com/javascript/string) with another string to the end.

The syntax of the padEnd() method is:

str.padEnd(targetLength, padString)

The padEnd() method takes two parameters:

* targetLength - The length of the final string after the current string has been padded.
* padString (optional) - The string to pad the current string with. Its default value is " ".

   <script>

        let string1 = "CODE";

        console.log(string1.padEnd(10,"$"))*//CODE$$$$$$*

    </script>

**STRING SEARCH METHODS**

**1.indexOf()**

-The indexOf() method returns the **index** (position) of the **first** occurrence of a string in a string, or it returns -1 if the string is not found:

The syntax of the indexOf() method is:

str.indexOf(searchValue, fromIndex)

The indexOf() method takes in:

* searchValue - The value to search for in the string. If no string is provided explicitly, "undefined" will be searched.
* fromIndex (optional) - The index to start the search at. By default it is 0. If fromIndex < 0, the search starts at index 0.

1.Returns the first index of the value in the string if it is present at least once.

<script>

        let text = "Please locate where 'locate' occurs!";

        console.log(text.indexOf("a"));*//3*

        console.log(text.indexOf("locate"));*//7*

    </script>

2. Returns -1 if the value is not found in the string.

  <script>

        let str = "Apple, Banana, Kiwi";

        console.log(str.indexOf("orange"));*//-1*

    </script>

3. The indexOf() method is case sensitive

    <script>

        let str1 = "Apple, Banana, Kiwi";

        console.log(str.indexOf("apple"));*//-1*

    </script>

4. For empty string searchValue and fromIndex less than the string's length, indexOf returns the value the same as fromIndex.

    <script>

        let str2 = "Apple, Banana, Kiwi";

        console.log(str2.indexOf("", 10));*//10*

    </script>

5. Similarly, for empty string searchValue and fromIndex greater than the string's length, indexOf returns the string's length.

 <script>

        let str3 = "Apple, Banana, Kiwi";

        console.log(str3.indexOf("", 55));*//19*

    </script>

**2.lastIndexOf()**

-The lastIndexOf() method returns the **index** of the last occurrence of a specified text in a string:

The syntax of the lastIndexOf() method is:

str.lastIndexOf(searchValue, fromIndex)

## lastIndexOf() Parameters

The lastIndexOf() method takes in:

* substr- The value to search for in the given string.
* fromIndex (optional) - The index to start searching the string backwards. By default it is +Infinity.

<script>

        let str = "I love Java.The Best language is Java.";

        console.log(str.lastIndexOf("Java"));

   </script>

2.Both indexOf(), and lastIndexOf() return -1 if the text is not found:

 <script>

        let string = "Hey..."

        console.log(string.lastIndexOf("P"))

   </script>

**3.search()**

-The search() method searches a string for a string (or a regular expression) and returns the position of the match:

 <script>

        let text = "The rain in SPAIN stays mainly in the plain";

        console.log(text.search("ain"));

    </script>

The syntax of the search() method is:

str.search(regexp)

The search() method takes a **single** parameter:

* regExp - A regular expression object (Argument is implicitly converted to regExp if it is a non-regExp object)

-The two methods are **NOT** equal. These are the differences:

* The search() method cannot take a second start position argument.
* The indexOf() method cannot take powerful search values (regular expressions).

**4.match()**

-The match() method returns an array containing the results of matching a string against a string (or a regular expression).

1.Perform a search for "ain":

<script>

        let text = "The rain in SPAIN stays mainly in the plain";

        console.log(text.match("ain"))

</script>

2.Perform a search for "ain":

<script>

        let text1 = "The rain in SPAIN stays mainly in the plain";

        console.log(text1.match("ain"))

</script>

3.Perform a global search for "ain":

<script>

        let text2 = "The rain in SPAIN stays mainly in the plain";

        console.log(text2.match(/ain/g));

 </script>

4.Perform a global, case-insensitive search for "ain":

<script>

        let text3 = "The rain in SPAIN stays mainly in the plain";

        console.log(text3.match(/ain/gi));

    </script>

**5.matchAll()**

-The matchAll() method returns an iterator containing the results of matching a string against a string (or a regular expression).

    <script>

        let text = "I love cats. Cats are very easy to love. Cats are very popular."

        const iterator = text.matchAll("Cats");

        console.log(Array.from(iterator));

    </script>

-If the parameter is a regular expression, the global flag (g) must be set, otherwise a TypeError is thrown.(i) it checks both Uppercase or lowercase.

    <script>

        let text1 = "I love cats. Cats are very easy to love. Cats are very popular."

        const iterator1 = text.matchAll(/Cats/gi);

        console.log(Array.from(iterator1));

    </script>

**6.includes()**

-The includes() method returns true if a string contains a specified value.

Otherwise it returns false.

-includes() is case sensitive.

 <script>

        let text = "Hello world, welcome to the universe.";

        console.log(text.includes("world"));*//true*

    </script>

The includes() method takes in:

* searchString - A string to be searched for within str.
* position (optional) - The position within str to begin searching for searchString. By default, it is **0**.

<script>

        let text1 = "Hello world, welcome to the universe.";

        console.log(text1.includes("world",13));*//false*

</script>

* Returns true if searchString is found anywhere within str.
* Returns false if searchString is not found anywhere within str.

**7.startsWith()**

-The startsWith() method returns true if a string begins with a specified value.Otherwise it returns false:

-The startsWith() method is case sensitive.

The syntax of the startsWith() method is:

str.startsWith(searchString, position)

   <script>

        let text = "Hello world, welcome to the universe.";

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

    </script>

    <script>

        let text1 = "Hello world, welcome to the universe.";

        console.log(text1.startsWith("Hello"));

    </script>

-Here, str is a string.

The startsWith() method takes in :

* searchString - The characters to be searched for at the beginning of str.
* position (optional) - The position in str to start searching for searchString. Default value is 0.

<script>

        let sentence = "Java is to JavaScript what Car is to Carpet.";

        let check3 = sentence.startsWith("JavaScript", 11);

        console.log(check3); *// true*

    </script>

**8.endsWith()**

-The endsWith() method returns true if a string ends with a specified value.Otherwise it returns false:

 <script>

        let text = "John Doe";

        console.log(text.endsWith("Doe"));

    </script>

The endsWith() method takes two parameters:

* searchString - The characters to be searched for at the end of str.
* length (optional) - It is used as the length of str where searchString is searched. Default value is [str.length](https://www.programiz.com/javascript/library/string/length).

    <script>

        let text1 = "Hello world, welcome to the universe.";

        console.log(text1.endsWith("world", 11));

    </script>

**9.String()**

Description

The String() method converts a value to a string.

Note

The String() method returns the same as the toString() method for any value.

**Syntax**

String(*value*)

Parameters

|  |
| --- |
|  |
| Parameter | Description |
| *value* | Required. A JavaScript value. |

Return Value

|  |
| --- |
|  |
| Type | Description |
| A string. | The value converted to a string. |

**2.Number**

-In JavaScript, the **number** type represents numeric values (both integers and floating-point numbers).Integers - Numeric values without any decimal parts.

* **Integers** - Numeric values without any decimal parts. Example: **3**, **-74**, etc.
* **Floating-Point** - Numeric values with decimal parts. Example: **3.15**, **-1.3**, etc.

*<!--! Number -->*

*<!--? Example1 -->*

   <script>

      let num1 = 10; *// Integer*

      let num2 = 10.22; *// Floating point number*

      document.write("The value of num1 is " + num1 + "<br/>");

      document.write("The value of num2 is " + num2 + "<br/>");

      console.log(typeof num1);

      console.log(typeof num2);

   </script>

*<!--? Example2 -->*

    <script>

        let num3 = 98e4;    *// 980000*

        let num4 = 98e-4;   *// 0.0098*

        document.write("The value of num3 is: " + num3 + "<br/>");

        document.write("The value of num4 is: " + num4);

        console.log(typeof num3);

        console.log(typeof num4);

    </script>

255; // two-hundred and fifty-five

255.0; // same number

255 === 255.0; // true

255 === 0xff; // true (hexadecimal notation)

255 === 0b11111111; // true (binary notation)

255 === 0.255e3; // true (decimal exponential notation)

**JavaScript Number Methods and Properties**

|  |  |
| --- | --- |
| Name | Description |
| [constructor](https://www.w3schools.com/jsref/jsref_constructor_number.asp) | Returns the function that created JavaScript's Number prototype |
| [EPSILON](https://www.w3schools.com/jsref/jsref_number_epsilon.asp) | Returns the difference between 1 and the smallest number greater than 1 |
| [isFinite()](https://www.w3schools.com/jsref/jsref_isfinite_number.asp) | Checks whether a value is a finite number |
| [isInteger()](https://www.w3schools.com/jsref/jsref_isinteger.asp) | Checks whether a value is an integer |
| [isNaN()](https://www.w3schools.com/jsref/jsref_isnan_number.asp) | Checks whether a value is Number.NaN |
| [isSafeInteger()](https://www.w3schools.com/jsref/jsref_issafeinteger.asp) | Checks whether a value is a safe integer |
| [MAX\_SAFE\_INTEGER](https://www.w3schools.com/jsref/jsref_max_safe_integer.asp) | Returns the maximum safe integer in JavaScript. |
| [MIN\_SAFE\_INTEGER](https://www.w3schools.com/jsref/jsref_min_safe_integer.asp) | Returns the minimum safe integer in JavaScript |
| [MAX\_VALUE](https://www.w3schools.com/jsref/jsref_max_value.asp) | Returns the largest number possible in JavaScript |
| [MIN\_VALUE](https://www.w3schools.com/jsref/jsref_min_value.asp) | Returns the smallest number possible in JavaScript |
| [NaN](https://www.w3schools.com/jsref/jsref_number_nan.asp) | Represents a "Not-a-Number" value |
| [NEGATIVE\_INFINITY](https://www.w3schools.com/jsref/jsref_negative_infinity.asp) | Represents negative infinity (returned on overflow) |
| [POSITIVE\_INFINITY](https://www.w3schools.com/jsref/jsref_positive_infinity.asp) | Represents infinity (returned on overflow) |
| [parseFloat()](https://www.w3schools.com/jsref/jsref_number_parsefloat.asp) | Parses a string an returns a number |
| [parseInt()](https://www.w3schools.com/jsref/jsref_number_parseint.asp) | Parses a string an returns a whole number |
| [prototype](https://www.w3schools.com/jsref/jsref_prototype_num.asp) | Allows you to add properties and methods to an object |
| [toExponential(x)](https://www.w3schools.com/jsref/jsref_toexponential.asp) | Converts a number into an exponential notation |
| [toFixed(x)](https://www.w3schools.com/jsref/jsref_tofixed.asp) | Formats a number with x numbers of digits after the decimal point |
| [toLocaleString()](https://www.w3schools.com/jsref/jsref_tolocalestring_number.asp) | Converts a number into a string, based on the locale settings |
| [toPrecision(x)](https://www.w3schools.com/jsref/jsref_toprecision.asp) | Formats a number to x length |
| [toString()](https://www.w3schools.com/jsref/jsref_tostring_number.asp) | Converts a number to a string |
| [valueOf()](https://www.w3schools.com/jsref/jsref_valueof_number.asp) | Returns the primitive value of a number |

**Note**

-Most programming languages have many number types:

-Whole numbers (integers):  
byte (8-bit), short (16-bit), int (32-bit), long (64-bit)

-Real numbers (floating-point):  
float (32-bit), double (64-bit).

-Javascript numbers are always one type:  
double (64-bit floating point).

**JAVASCRIPT NUMBER CONSTRUCTOR**

-The constructor property returns the function that created the Number prototype.

-For JavaScript numbers the constructor property returns:

**function Number() { [native code] }**

    <script>

        let num1 = new Number(123);

        console.log(num1);

        console.log(typeof num1);*//Object*

        let num2 = 123;

        console.log(num1 == num2);*//true*

        console.log(num1 === num2);*//false*

    </script>

**1.Number()**

-The Number() method converts a value to a number.

-If the value cannot be converted, NaN is returned.

<body>

*<!--! Number()  -->*

*<!--todo : - It converts String,Boolean into Number format  -->*

    <script>

        let boolean = true;

        console.log(Number(boolean));*//1*

        let String = "123";

        console.log(Number(String));*//123*

        let a;

        console.log(Number(undefined));*//NaN*

    </script>

## Syntax

Number(value)

## Parameters

|  |
| --- |
|  |
| Parameter | Description |
| value | Optional. A JavaScript value (variable). |

## Return Value

|  |  |
| --- | --- |
| Type | Description |
| A number | Returns the value as a number. If the value cannot be converted to a number, NaN is returned. If no value is provided, 0 is returned. |

## Notes

-For **booleans**, Number() returns 0 or 1.

-For **dates**, Number() returns milliseconds since January 1, 1970 00:00:00.

-For **strings**, Number() returns a number or NaN.

**2.Number.isFinite()**

Syntax

Number.isFinite(*value*)

Return Value

|  |
| --- |
|  |
| Type | Description |
| A boolean | true if the value is a finite Number, otherwise false. |

<script>

        let num1 = 123;

        console.log(Number.isFinite(num1))*//true*

        console.log(Number.isFinite(5-5));*//true*

        console.log(Number.isFinite("123"));*//false*

        console.log(Number.isFinite(undefined));*//false*

        console.log(Number.isFinite(true));;*//false*

    </script>

**3.isInteger()**

-The Number.isInteger() method returns true if a value is an integer of the datatype Number.

-Otherwise it returns false.

    <script>

        let num = 10;

        console.log(Number.isInteger(num)); *// true*

        let decimal = 10.5;

        console.log(Number.isInteger(decimal)); *// false*

    </script>

**4.isNaN()**

-In JavaScript, NaN is short for "Not-a-Number".

-In JavaScript, NaN is a number that is not a legal number.

-The Number.isNaN() method returns true if the value is NaN, and the type is a Number.

**Synatx:-**

Number.isNan(value);

## Return Value

|  |
| --- |
|  |
| Type | Description |
| A boolean. | true if the value is Number.NaN, otherwise false. |

  <script>

        let num1 = "Hello";

        console.log(Number.isNaN(num1));//false

        let num2 = 12;

        console.log(Number.isNaN(num2));//false

    </script>

**5.toFixed()**

The toFixed() method rounds the string to a specified number of decimals.

## Syntax

number.toFixed(x)

    <script>

        let num1 = 4554554.33443;

        console.log(num1.toFixed(0));

        let num2 = 2332;

        console.log(num2.toFixed(4));

    </script>

## Return Value

|  |
| --- |
|  |
| Type | Description |
| A string | The representation of a number with (or without) decimals. |

**6.parseInt()**

-The Number.parseInt method parses a value as a string and returns the first integer.

-A radix parameter specifies the number system to use:

-2 = binary, 8 = octal, 10 = decimal, 16 = hexadecimal.

-If radix is omitted, JavaScript assumes radix 10. If the value begins with "0x", JavaScript assumes radix 16.

**Syntax**

Number.parseInt(*string, radix*)

## Return Value

|  |
| --- |
|  |
| Type | Description |
| Number | NaN if no integer is found. |

  <script>

        console.log(Number.parseInt("10"));

        console.log(Number.parseInt("10"));

        console.log(Number.parseInt("10.00"));

        console.log(Number.parseInt("10.33"));

        console.log(Number.parseInt("34 45 66"));

        console.log(Number.parseInt(" 60 "));

        console.log(Number.parseInt("40 years"));

        console.log(Number.parseInt("He was 40"));

    </script>

**7.parseFloat()**

-The Number.parseFloat() method parses a value as a string and returns the first number.

**Syntax**

Number.parseFloat(*value*)

**Parameters**

|  |
| --- |
|  |
| Parameter | Description |
| *value* | Required. The value to parse. |

**Return Value**

|  |
| --- |
|  |
| Type | Description |
| Number | NaN if no number is found. |

   <script>

        console.log(Number.parseFloat(10));

        console.log(Number.parseFloat("10") );

        console.log(Number.parseFloat("10.33"));

        console.log(Number.parseFloat("34 45 66"));

        console.log(Number.parseFloat("He was 40"));

   </script>

**Notes**

**-If the first character cannot be converted, NaN is returned.**

**8.toLocalString()**

-The toLocaleString() returns a number as a string, using local language format.

-The language format depends on the locale setup on your computer.

    <script>

        let num = 1000000;

        let text = num.toLocaleString();

        console.log(text);

    </script>

Syntax

*number*.toLocaleString(*locales*, *options*)

**3.JavaScript BigInt**

**-**BigInt is a type of number that can represent very large or very small integers beyond the range of the regular number data type.

-Note: The regular number data type can handle values less than (2^53 - 1) and greater than -(2^53 - 1).

-A BigInt number is created by appending **n** to the end of an integer.

<script>

        let largeNum = 1245646564515635412348923448234842842343546576876789n;

        document.write("The value of the largeNum is " + largeNum + "<br/>");

</script>

-A Boolean data can only have one of two values: true or false. For

**4.Boolean**

example,

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! Boolean  -->*

*<!--? In JavaScript true and false has values  -->*

*<!--\* 1.true = 1  -->*

*<!--\* 2.false = 0  -->*

    <script>

        let bool1 = true;

        let bool2 = false;

        document.write("The value of the bool1 is " + bool1 + "<br/>");

        document.write("The value of the bool2 is " + bool2 + "<br/>");

     </script>

</body>

</html>

# **JavaScript Boolean constructor**

   <script>

        let bool = false;

        let text = bool.constructor;

        consol.log(text)

    </script>

## Description

The constructor property returns the function that created the Boolean prototype.

**Syntax**

*boolean*.constructor

**Return Value**

|  |
| --- |
| function Boolean() { [native code] } |

**5.Symbol**

-The Symbol data type is introduced in the ES6 version of JavaScript. It is used to create unique primitive, and immutable values.

-The **Symbol()** constructor can be used to create a unique symbol, and you may pass the string as a parameter of the Symbol() constructor.

### **Example**

-In the example below, we created the *sym1* and *sym2* symbols for the same string. After that, we compared the value of *sym1* and *sym2*, and it gave a false output. It means both symbols are unique.

*<!--! Symbol  -->*

<script>

        let sym1 = Symbol("123");

        let sym2 = Symbol("123");

        let res = sym1 == sym2;

        console.log("Is sym1 is Equal To sym2 : "+ res);

*// Output: Is sym1 is Equal To sym2 : false*

</script>

**6.Undefined**

-The undefined type is a primitive type that has only one value undefined. By default, when a variable is declared but not initialized, it defaults to undefined.

    <script>

        let x;

        console.log(x);

    </script>

    <script>

        let counter;

        console.log(counter);        *// undefined*

        console.log(typeof counter); *// undefined*

    </script>

**7.Null**

-The null type is the second primitive data type that also has only one value null. For example:

   <script>

        let obj = null;

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

    </script>

-The typeof null returns object is a known bug in JavaScript. A proposal to fix was rejected due to the potential to break many existing sites.

JavaScript defines that null is equal to undefined as follows:

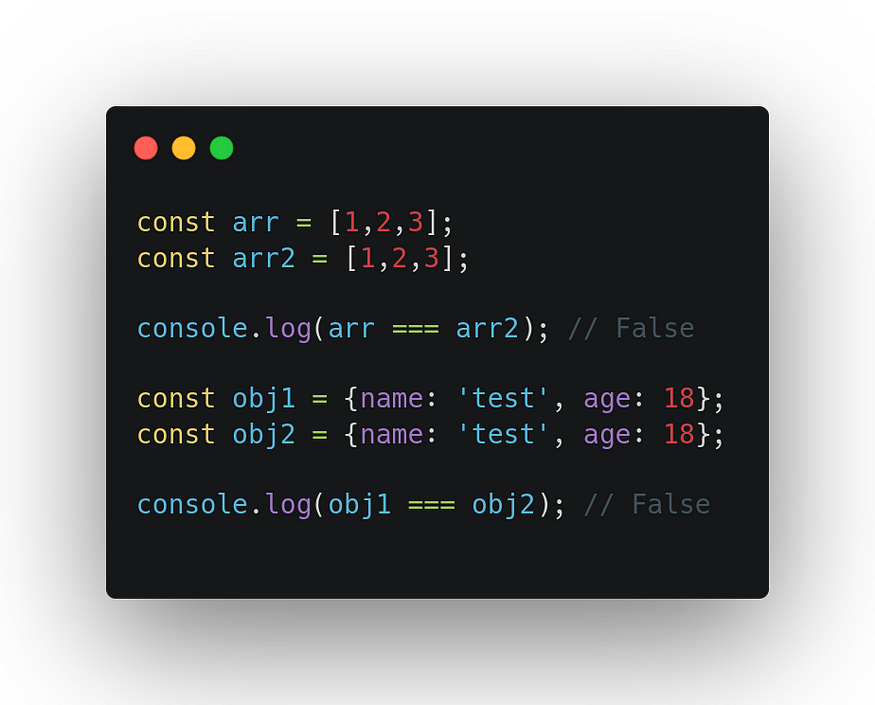
 <script>

        console.log(a == b);

 </script>

**2]Non-Primitive Datatype**

-The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.



-Non-primitive values are mutable

**-**Non-primitive compare by reference not value

**Example :-**

**Object , Array, Functions, Date ,Regex**

**1.Object**

-Object is the Realtime entity

-Anything which has the Existence in the World is called Object

-Object some have Properties And Behaviour.

-In JavaScript, an [object](https://www.javascripttutorial.net/home/javascript-objects/) is a collection of [properties](https://www.javascripttutorial.net/home/javascript-object-properties/), where each property is defined as a key-value pair,where the keys are strings (or symbols) and the values can be any data type (including other objects).

-Objects are one of the fundamental building blocks of JavaScript, allowing you to represent real-world entities and organize data in a structured way.

**Key Concepts:**

* **Properties**:-

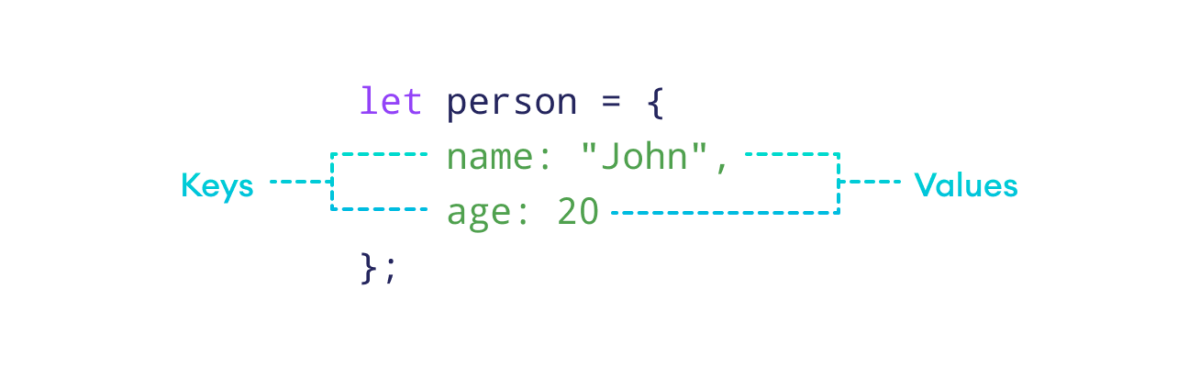
-The key-value pairs within an object are called properties. The key represents the name of the property, and the value represents its associated data.

* **Methods**:-

-Functions that are stored as properties of an object are called methods. They allow objects to perform actions or operations.

Here is a **car object** example:

|  |  |  |
| --- | --- | --- |
| **Car Object** | **Properties** | **Methods** |
|  | car.name = Fiat  car.model = 500  car.weight = 850kg  car.color = white | car.start()  car.drive()  car.brake()  car.stop() |



-Object does not remember the order of properties.

**There are several ways to create objects in JavaScript:**

**1]BY USING LITERALS**

**2]BY USING NEW KEYWORD**

**3]BY USING FUNCTION CONSTRUCTOR**

**4]BY USING SINGLTON PATTERN**

**5]BY USING ES6 CLASSES**

**6]BY USING CREATE() METHOD**

**1.Object Literals:-**

-This is the most common way to create objects. You use curly braces {} to define the object, with each property separated by a comma.

    <script>

        let person = {

             firstName: "John",

             lastName: "Doe",

             age: 30,

             greet: function() {

             console.log("Hello, my name is " + this.firstName + " " + this.lastName);

            }

        };

        console.log(person);

        console.log(person.greet())

    </script>

-An object literal is a list of property names:values inside curly braces {}.

The syntax of JavaScript object is:

const objectName = {

key1: value1,

key2: value2,

...,

keyN: valueN

};

Here,

* objectName - Name of the object.
* key1: value1 - The first key-value pair.
* key2: value2 - The second key-value pair.
* keyN: valueN - The Nth key-value pair.

Each key-value pair has a colon : between them and is separated by a comma ,.

Note:

An object literal is also called an object initializer.

**Access Object Properties**

-You can access the value of a property by using its key.

**I. Using Dot Notation**

const dog = {

name: "Rocky",

};

// access property

console.log(dog.name);

// Output: Rocky

**II. Using Bracket Notation**

const cat = {

name: "Luna",

};

// access property

console.log(cat["name"]);

// Output: Luna

**2.Using New Keyword**

 <script>

        let obj = new Object();

        console.log(obj);

        obj.name="Amol";

        obj.id=123;

        console.log(obj);

    </script>

-Here, the new [keyword](https://www.programiz.com/javascript/keywords-identifiers) is used with the Object() instance to create an object.

-After that the instance is created then you have to manually add the Properties inside the Object Using assigning.

**3.Using Function Constructor**

    <script>

        function Emp(*ename*,*eid*,*esal*) {

            this.ename=*ename*;

            this.eid=*eid*;

            this.esal=*esal*;

        }

        let emp = new Emp("Amol",123,35000);

        console.log(emp);

    </script>

-In this example we have to take the help to assigning value odf properties.

-In the constructor function, this has no value.

-The value of this will become the new object when a new object is created.

**4.BY USING OBJECT CREATE () METHOD**

*<!--! USING OBJECT CREATE() METHOD  -->*

*<!--todo : - For that you have Create one Variable and assign to Object.create( ).  -->*

*<!--todo : - Inside that you have pass the null is mandatory  -->*

    <script>

        let obj = Object.create(null);

        obj.name = "Amol"

        console.log(obj);

    </script>

-In this example we have to use create method with **null** datatype.

It will create the Object but it will null mandatory neither the browser will throw the error.

**5.BY USING ES6 CLASS**

 <script>

        class myObject {

                constructor(*name*) {

                this.name = *name*;

            }

         }

        var obj = new myObject("hello");

        console.log(obj);

    </script>

**6.BY USING SINGLTON PATTERN**

<script>

        var obj = new function(){

            this.name = "hello";

        }

        console.log(obj);

    </script>

**JavaScript Object Operations**

**C = CREATE**

**R = RETRIVE**

**U = UPADATE**

**D = DELETE**

**CRUD OPERATION**

-In JavaScript, we can perform various operations on object properties like modifying, adding, deleting, and so on.

**1. Modify Object Properties**

-We can modify object properties by assigning a new value to an existing key. For example,

const person = {

name: "Bobby",

hobby: "Dancing",

};

// modify property

person.hobby = "Singing";

// display the object

console.log(person);

// Output: { name: 'Bobby', hobby: 'Singing' }

-In the above example, we used the dot notation to change the value of the hobby key from Dancing to Singing.

**2. Add Object Properties**

const student = {

name: "John",

age: 20,

};

// add properties

student.rollNo = 14;

student.faculty = "Science";

// display the object

console.log(student);

// Output: { name: 'John', age: 20, rollNo: 14, faculty: 'Science' }

**-** In the above example, the keys rollNo and faculty do not exist within the object. Hence, when we assign values to these keys, new properties are added to the object.

**3. Delete Object Properties**

-We can remove properties from an object using the delete operator. For example,

const employee = {

name: "Tony",

position: "Officer",

salary: 30000,

};

// delete object property

delete employee.salary

// display the object

console.log(employee);

// Output: { name: 'Tony', position: 'Officer' }

**JavaScript Nested Objects**

A nested object contains another object as a property. For example,

// outer object student

const student = {

name: "John",

age: 20,

// contains another object marks

marks: {

science: 70,

math: 75

}

};

// display student

console.log(student);

// Output: { name: 'John', age: 20, marks: { science: 70, math: 75 } }

Access Properties of Nested Objects

const student = {

name: "John",

age: 20,

marks: {

science: 70,

math: 75

}

};

// use dot notation

console.log(student.marks.science); // 70

// use bracket notation

console.log(student["marks"]["math"]); // 75

**2.ARRAY**

-Array is continuous Block of Memory which is used to store the Heterogeneous (Different Type) Type of data in JavaScript

-In JavaScript Array is used to store the Heterogeneous (Different Types Of Values.) Type of data.

-An array is an [object](https://www.programiz.com/javascript/object) that can store multiple values at once.

-An array is a special variable, which can hold more than one value.

-Array is type of object.

### **Why Use Arrays?**

-Arrays allow us to organize related data by grouping them within a single variable.

-Suppose you want to store a list of fruits. Using only variables, this process might look like this:

let fruit1 = "Apple";

let fruit2 = "Banana";

let fruit3 = "Orange";

Here, we've only listed a few fruits. But what if we need to store 100 fruits?

For such a case, the easiest solution is to store them in an array.

let fruits = ["Apple", "Banana", "Orange", ...];

**Basic Terminologies of JavaScript Array**

* **Array:** A data structure in JavaScript that allows you to store multiple values in a single variable.
* **Array Reference variable**: - In the array the Array reference Variable is created which holds the Address of the Array
* **Array Element:** Each value within an array is called an element. Elements are accessed by their index.
* **Array Index value**: - Array index value is always start with **0** [**Zero**] with the help of Array Index value we store or manipulate the data.
* **Array Length:** The number of elements in an array. It can be retrieved using the length property. The Length of the Array is always starts with **1.**

**The length Property**

-The length property of an array returns the length of an array (the number of array elements).

-The length always starts with **1**.

**Example : -**

const fruits = ["Banana", "Orange", "Apple", "Mango"];

let length = fruits.length;

-The length property is always one more than the highest array index.

## Create an Array

Using an array literal is the easiest way to create a JavaScript Array.

**Syntax: -**

const array\_name = [item1, item2, ...];

It is a common practice to declare arrays with the const keyword.

**For Example : -**

const **array1** = [“Amol”,”Ganesh”,”Onkar”,”Rohit”, true,24,null,undefined]

**LENGTH**

**array1**

**INDEX VALUE**

**1**

**0**

Amol

**8**

**7**

**6**

**5**

**3**

**2**

**4**

**1**

Ganesh

**2**

Onkar

**3**

Rohit

**4**

true

**5**

24

**6**

null

**7**

undefined

-In JavaScript, you cannot directly print the memory address of an array (or any object) like in languages like C/C++. JavaScript abstracts memory management from the developer.

However, if you need a way to uniquely identify an array, you can use its object reference:

const arr = [1, 2, 3];  
  
console.log(arr); // This prints the array content, not the address

-In JavaScript we can create an array in three ways:

**1.By Using Literals**

**2.By Using Array Of Object**

**3.By Using New Keyword /Array Constructor**

**4.By Using Array.of() Method**

**5.By Using Array.from() Method**

**1.By Using Literals**

We can create an array by placing elements inside an array literal [], separated by commas**( , ).** For example,

const numbers = [10, 30, 40, 60, 80];

Here,

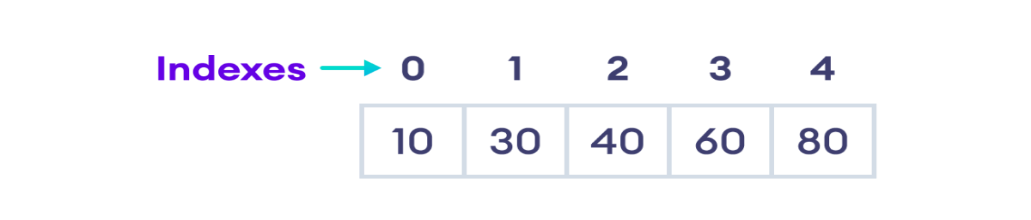
* numbers - Name of the array.
* [10, 30, 40, 60, 80] - Elements of the array.

## Access Elements of an Array

-Each element of an array is associated with a number called an index, which specifies its position inside the array.

Consider the following array:

let numbers = [10, 30, 40, 60, 80];

Here is the indexing of each element:s

-We can use an array index to access the elements of the array.

|  |  |
| --- | --- |
| Code | Description |
| numbers[0] | Accesses the first element 10. |
| numbers[1] | Accesses the second element 30. |
| numbers[2] | Accesses the third element 40. |
| numbers[3] | Accesses the fourth element 60. |
| numbers[4] | Accesses the fifth element 80. |

// Example 1: Empty Array

let emptyArray = [];

// Example 2: Array with elements

let numbers = [1, 2, 3, 4, 5]; let fruits = ["apple", "banana", "cherry"];

**2.By Using Array Of Object**

**<!--! ARRAY USING OBJECT -->**

    <script>

        let obj={

            name:"Amol",

            Age:24,

            Proff:"Software Developer",

            array:[10,20,30,40,50]

        }

        console.log(obj);

        console.log(obj.name);

        console.log(obj.array);

        console.log(obj.array[0]);

    </script>

-By Using Object you can create the Array.

-For that you have define one Object. Inside that you have define Name of the array as key and Values in the Pair.

<script>

*// Create an array of objects*

            const students = [

            { name: "Alice", age: 20 },

            { name: "Bob", age: 21 },

            { name: "Charlie", age: 19 }

            ];

*// Access properties of an object in the array*

            console.log(students[1]); *// Output: "Bob"*

 </script>

**3.By Using New Keyword / Array Constructor**

<script>

        let array = new Array();

        console.log(array);

        console.log(array[0]);

        array[0]=10;

        array[1]="Amol";

        array[2]=true;

        array[3]=null;

        console.log(array)

        console.log(array.length)

    </script>

-In this we have to use the new keyword.

You can use the Array constructor to create an array. This method is particularly useful if you want to create an array of a specific length.

// Example 1: Empty Array

let emptyArray = new Array();

// Example 2: Array with specified length

let tenElementsArray = new Array(10);

// Example 3: Array with elements

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

**4. Using the Array.of( ) Method**

-The Array.of() method creates a new Array instance with a variable number of arguments, regardless of the number or type of the arguments.

Example : -

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

let fruits = Array.of("apple", "banana", "cherry");

**5. Using the Array.from( ) Method**

-The Array.from() method creates a new, shallow-copied Array instance from an array-like or iterable object.

  <script>

*// Array.from Method*

        let arrayFromString = Array.from("hello");

        console.log(arrayFromString); *// Output: ["h", "e", "l", "l", "o"]*

    </script>

**ARRAY BASIC METHODS**

**1.Array.toString()**

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

<script>

    const fruits = ["Banana", "Orange", "Apple", "Mango"];

    console.log(fruits);

    console.log(fruits.toString()); *// Outputs: Banana,Orange,Apple,Mango*

    </script>

**2.Array.at()**

-The at() method returns an indexed element from an array that mentioned inside the Parenthesis( ).

-It is a Parameterized method it accepts the Parameter.

<script>

        let numbers = [10,20,30,40,50];

        console.log(numbers);

        console.log(numbers.at(2));

    </script>

**3. Array.join()**

-The join method in JavaScript is used to combine all the elements of an array into a single string. The elements are separated by a specified separator string. If no separator is provided, a comma is used by default.

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

let result = fruits.join();

console.log(result); // Output: "apple,banana,cherry"

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

let result = fruits.join(" - ");

console.log(result); // Output: "apple - banana - cherry"

**4. Array.pop()**

- The pop() method removes the last element from an array.

-The pop() method returns the value that was "popped out":

 <script>

        let arr = ["Amol",24,null,true,10,20,30];

        console.log(arr.pop());

    </script>

    <script>

        const fruits = ["Banana", "Orange", "Apple", "Mango"];

        console.log(fruits.pop())

    </script>

**5.Array.push()**

-It is Parameterized method accepts the arguments.

-You can add Multiple elements in the Array.

-The push() adds the elements from Last Index.

**Syntax:-**

**array.push(element1, ..., elementN)**

-Adding single elements in the array

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

fruits.push("cherry");

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

-Adding Multiple Elements.

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

let newLength = fruits.push("cherry", "date", "fig");

console.log(fruits); // Output: ["apple", "banana", "cherry", "date", "fig"]

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

**6.Array.shift()**

-The shift() method removes the first array element and "shifts" all other elements to a lower index.

-It is Non-Parameterized method

 <script>

        let arr = [1,2,3,4,5,6,7,8,9]

        console.log(arr.shift());//1

        console.log(arr);*//2,3,4,5,6,7,8,9*

    </script>

**7.Array.unshift()**

-The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements

-The unshift() method returns the new array length:

    <script>

        const fruits = ["Banana", "Orange", "Apple", "Mango"];

        fruits.unshift("Lemon");

        console.log(fruits)

    </script>

**Note:We cannot use the shift() and unshift() more because it changes the array index value**

**8.Array.splice()**

-The splice() method can be used to add new items to an array

    <script>

        const fruits = ["Banana", "Orange", "Apple", "Mango"];

        console.log(fruits)

        fruits.splice(2, 0, "Lemon", "Kiwi");

        console.log(fruits);// Banana,Orange,Lemon,Kiwi,Apple,Mango

    </script>

-The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

-The second parameter (0) defines **how many** elements should be **removed**.

-The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

**Example NO2:-**

    <script>

        const fruits1 = ["Banana", "Orange", "Apple", "Mango"];

        console.log(fruits1);

        fruits1.splice(2, 2, "Lemon", "Kiwi");

        console.log(fruits1)

    </script>

**Original Array:**  
Banana,Orange,Apple,Mango

**New Array:**Banana,Orange,Lemon,Kiwi

**Removed Items:**Apple,Mango

**Note : Splice method changes the original array**

**9.Array.slice()**

-The slice() method slices out a piece of an array into a new array:

<script>

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

        const citrus = fruits.slice(1);

        console.log(citrus)

    </script>

Note

-The slice() method creates a new array.

-The slice() method does not remove any elements from the source array.

-The slice() method can take two arguments like slice(1, 3).

-The method then selects elements from the start argument, and up to (but not including) the end argument.

**ARRAY ITERATIONS METHODS**

**1.map()**

-Map is array method in Javascript.

-Map method used to perform the operation on the array.

-Map method does not return undefined unlike the forEach method.

-Map method is used to perform the operation(Logical) on the array.

-When we applied map on the array the size of the array is same like original array.

**2.filter**

**3.reduce()**

-The reduce() method runs a function on each array element to produce (reduce it to) a single value.

-The reduce() method works from left-to-right in the array. See also reduceRight().

-The reduce() method does not reduce the original array.

-Note that the function takes 4 arguments:

**Parameters:**

**1.callback**: A function to execute on each element in the array, taking four arguments:

* **Total /Accumulator**: It is a required parameter, also called accumulator, that holds the initialValue in the beginning and then the last returned value of the function.
* **curValue**: This is a required argument that holds the value of the current element being executed.
* **curIndex**: It is an optional parameter holding the index of the current value.
* **Array**: This is also an optional parameter that holds the complete array object on which the operation is performed.

**2. initialValue (optional):**

**-**This is an optional parameter and holds the initial value passed to the function.

    <script>

        let num = [10,20,30,40,50];

        let sum =num.reduce((*accu*,*cValue*)=>{

*return* *accu* + *cValue*;

        })

        console.log(sum)

    </script>

**When to Use the reduce() Method**

-As shown above, the reduce() method is recommended when you need to have a single value returned from iterating over your array.

**Difference Between forEach, Map(), filter(), reduce() Methods**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SrNo | forEach | Map | Filter | Reduce |
| 1 | .forEach(), is used to execute the same code on every element in an array but does not change the array and it returns undefined. | The map method is used to create a new array with the result of a callback function called on each element in the original array | The name itself conveys what this method can do. It just filters the array and returns only the needed elements out of it. | The reduce method is used to reduce an array to a single value. The reduce method takes a callback function as its argument, which is called on each element in the array. |
| 2 | The For Each also accepts Function as an argument and that Function also take array as argument as an variable. | The callback function takes one argument, the current element, and returns the new value for that element. | The filter method takes a callback function as its argument, which is called on each element in the array | The callback function takes two arguments, the accumulator and the current element, and returns the new value for the accumulator. |
| 3 | It does not return new array | It Also Return New Array | It Also Return New Array | It Does Not Return Array |
| 4 | The Array Size is Similar | The Array Size is Similar | Array Size is not Similar | It does not return array |
| 5 | It return Modified Elements | It return Modified Elements | It return Filtered Elements | It return Only Single Element |
| 6 | The return values is always undefined | return value is not undefine | return value is not undefine | return value is not undefine |

**4.Array.keys()**

-The Array.keys() method returns an Array Iterator object with the keys of an array.

-keys() method only return the Index value not value inside the index.

**5.Array.entries()**

-Create an Array Iterator, and then iterate over the key/value pairs

-The entries() method returns an Array Iterator object with key/value pairs.

-The entries() method does not change the original array.

**FUNCTIONS**

-Function is a set of instruction to perform the set of instructions

-A JavaScript function is a block of code designed to perform a particular task.

-Function have some functionality to perform some Action

-Function have some instruction according to that instruction it performs some Action.

-Function have their own scope.

-A JavaScript function is executed when "something" invokes it (calls it).

-The **()** Operator

-The **()** operator invokes (calls) the function:

**-There are Five Types of Functions in JavaScript.**

1]Function Declaration

2]Function with Expression

3]Anonymous Function(No Name)

4]Arrow Function 🡺ES06 Function

5]IIFE Function (IMMEDIATE INVOKING FUNCTION)🡺ES Function

**-Functions are categorized into Two Types JavaScript**

1]Parametrized Functions.

2]Non-Parameterized Functions.

**Function contains the following parts:**

**Function Keyword**

- The function keyword is used to create the function.

**Function Name**

 – We have to give the Function Name to identified, followed by parentheses **()**.

**Function Body**

-Inside the Function body we can write the instruction.

-Instruction means to required steps to perform that action.

**Function Call Statement**

-Function call statement is used to call the Function.

-For Execution of that function calling is mandatory.

-So we have that as an Function Call Statement.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <script>

       //Keyword *FunctionName*

        function     myFunction() {

*//Function Body*

            console.log("Hello This Is Function Block")

        }

        myFunction()  *// Function Call Statement*

    </script>

</body>

</html>

**Syntax: -**

Keyword FunctionName

function functionName(){

//Function Body

-----------------------------

INSTRUCTIONS

----------------------------

-----------------------------

}

functionName();**//FunctionCall Statement**

**The () Operator**

-The () operator invokes (calls) the function:

**Why Functions?**

-With functions you can reuse code

-You can write code that can be used many times.

-You can use the same code with different arguments, to produce different results.

**DATE OBJECT**

-Date objects are static. The "clock" is not "running".

-The computer clock is ticking, date objects are not.

## JavaScript Date Output

By default, JavaScript will use the browser's time zone and display a date as a full text string:

Thu Jul 18 2024 16:27:41 GMT+0530 (India Standard Time)

-Creating Date Objects

-Date objects are created with the new Date() constructor.

-There are **9 ways** to create a new date object:

new Date()  
new Date(*date string*)  
new Date(*year,month*)  
new Date(*year,month,day*)  
new Date(*year,month,day,hours*)  
new Date(*year,month,day,hours,minutes*)  
new Date(*year,month,day,hours,minutes,seconds*)  
new Date(*year,month,day,hours,minutes,seconds,ms*)  
new Date(*milliseconds*)

**1.JavaScript new Date()**

-new Date() creates a date object with the **current date and time**:

**1.** **new Date()**

Example

const d = new Date();

console.log(d);

**2.new Date(*date string*)**

-new Date(*date string*) creates a date object from a **date string**:

Example1:-

const d = new Date("October 13, 2014 11:13:00");

console.log(d);

Example2:-

const d = new Date("2022-03-25");

console.log(d);

**3.new Date(*year, month,day,hour,minute,second,millisecond*)**

-new Date(*year, month, day, hour, minute, second, millisecond)* creates a date object with a **specified date and time**.

-7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

Example:-

const d = new Date(2018, 11, 24, 10, 33, 30, 0);

console.log(d);

-Specifying a month higher than 11, will not result in an error but add the overflow to the next year:

## Note

JavaScript counts months from **0** to **11**:

**January = 0**.

**December = 11**.

Example:-

const d = new Date(2018, 15, 24, 10, 33, 30);

console.log(d);

## Using 6, 4, 3, or 2 Numbers

6 numbers specify year, month, day, hour, minute, second:

### **Example**

const d = new Date(2018, 11, 24, 10, 33, 30);

console.log(d);

5 numbers specify year, month, day, hour, and minute:

### **Example**

const d = new Date(2018, 11, 24, 10, 33);  
console.log(d);

4 numbers specify year, month, day, and hour:

### **Example**

const d = new Date(2018, 11, 24, 10);  
console.log(d);

3 numbers specify year, month, and day:

### **Example**

const d = new Date(2018, 11, 24);  
console.log(d);

2 numbers specify year and month:

### **Example**

const d = new Date(2018, 11);  
console.log(d);

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

### **Example**

const d = new Date(2018);

console.log(d);

## Previous Century

One and two digit years will be interpreted as 19xx:

### **Example**

const d = new Date(99, 11, 24);

console.log(d);

### **Example**

const d = new Date(9, 11, 24);

console.log(d);

## new Date(milliseconds)

new Date(milliseconds) creates a new date object as **milliseconds** plus zero time:

### **Examples**

01 January 1970 plus 100 000 000 000 milliseconds is:

const d = new Date(100000000000);

console.log(d);

## Date Methods

When a date object is created, a number of methods allow you to operate on it.

Date methods allow you to get and set the year, month, day, hour, minute, second, and millisecond of date objects, using either local time or UTC (universal, or GMT) time.

## Displaying Dates

JavaScript will (by default) output dates using the **toString()** method. This is a string representation of the date, including the time zone. The format is specified in the ECMAScript specification:

### **Example**

Thu Jul 18 2024 17:15:53 GMT+0530 (India Standard Time)

-When you display a date object in HTML, it is automatically converted to a string, with the toString() method.

### **Example**

const d = new Date();  
d.toString();

-The toDateString() method converts a date to a more readable format:

### **Example**

const d = new Date();  
d.toDateString();

-The toUTCString() method converts a date to a string using the UTC standard:

### **Example**

const d = new Date();  
d.toUTCString();

The toISOString() method converts a date to a string using the ISO standard:

### **Example**

const d = new Date();  
d.toISOString();

JavaScript Date Input

-There are generally 3 types of JavaScript date input formats:

|  |  |
| --- | --- |
| **Type** | **Example** |
| ISO Date | "2015-03-25" (The International Standard) |
| Short Date | "03/25/2015" |
| Long Date | "Mar 25 2015" or "25 Mar 2015" |

The ISO format follows a strict standard in JavaScript.

The other formats are not so well defined and might be browser specific.

Date Get Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| getFullYear() | Get **year** as a four digit number (yyyy) |
| getMonth() | Get **month** as a number (0-11) |
| getDate() | Get **day** as a number (1-31) |
| getDay() | Get **weekday** as a number (0-6) |
| getHours() | Get **hour** (0-23) |
| getMinutes() | Get **minute** (0-59) |
| getSeconds() | Get **second** (0-59) |
| getMilliseconds() | Get **millisecond** (0-999) |
| getTime() | Get **time** (milliseconds since January 1, 1970) |
|  |  |

Note 1

The get methods above return **Local time**.

<script>

        let a = new Date();

        console.log(a);

        console.log(a.getMilliseconds());

        console.log(a.getSeconds());

        console.log(a.getMinutes());

        console.log(a.getHours());

        console.log(a.getDay())

        console.log(a.getDate());

        console.log(a.getMonth());

        console.log(a.getFullYear());

        console.log(a.getTimezoneOffset());

        console.log(a.getUTCSeconds());

        console.log(a.getUTCMinutes());

        console.log(a.getUTCHours());

        console.log(a.getUTCDate());

        console.log(a.getUTCDay());

        console.log(a.getUTCMonth());

        console.log(a.getUTCFullYear());

    </script>

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <script>

        const months = ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"];

        const d = new Date("2021-03-25");

        let month = months[d.getMonth()];

        document.write(month)

    </script>

</body>

</html>

# **JavaScript Set Date Methods**

-Set Date methods let you set date values (years, months, days, hours, minutes, seconds, milliseconds) for a Date Object.

## Set Date Methods

Set Date methods are used for setting a part of a date:

|  |  |
| --- | --- |
| **Method** | **Description** |
| setDate() | Set the day as a number (1-31) |
| setFullYear() | Set the year (optionally month and day) |
| setHours() | Set the hour (0-23) |
| setMilliseconds() | Set the milliseconds (0-999) |
| setMinutes() | Set the minutes (0-59) |
| setMonth() | Set the month (0-11) |
| setSeconds() | Set the seconds (0-59) |
| setTime() | Set the time (milliseconds since January 1, 1970) |

## The setFullYear() Method

-The setFullYear() method sets the year of a date object. In this example to 2020:

### **Example**

const d = new Date();  
d.setFullYear(2020);

The setFullYear() method can optionally set month and day:

### **Example**

const d = new Date();  
d.setFullYear(2020, 11, 3);

## The setMonth() Method

The setMonth() method sets the month of a date object (0-11):

### **Example**

const d = new Date();  
d.setMonth(11);

## The setDate() Method

The setDate() method sets the day of a date object (1-31):

### **Example**

const d = new Date();  
d.setDate(15);

The setDate() method can also be used to add days to a date:

### **Example**

const d = new Date();  
d.setDate(d.getDate() + 50);

If adding days shifts the month or year, the changes are handled automatically by the Date object.

## The setHours() Method

The setHours() method sets the hours of a date object (0-23):

### **Example**

const d = new Date();  
d.setHours(22);

## The setMinutes() Method

The setMinutes() method sets the minutes of a date object (0-59):

### **Example**

const d = new Date();  
d.setMinutes(30);

## The setSeconds() Method

The setSeconds() method sets the seconds of a date object (0-59):

### **Example**

const d = new Date();  
d.setSeconds(30);

**Spread Operator**

-The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

-Basically It is used to spread(uncompress/unpack) the values.

**const** numbersOne **=** **[1,** **2,** **3];**

**const** numbersTwo **=** **[4,** **5,** **6];**

**const** numbersCombined **=** **[...**numbersOne**,** **...**numbersTwo**];**

-We can use the More Than Spread Operator

**syntax** : ...variableName

-The spread operator in JavaScript, represented by three dots (**...**), allows you to expand iterable objects, such as arrays, strings, and objects, into their individual elements.

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

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

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

function sum(a, b, c) {

return a + b + c;

}

const numbers = [1, 2, 3];

const result = sum(...numbers);

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

**Rest Operator**

-In JavaScript, the rest operator, represented by three dots (**...**), allows a function to accept an indefinite number of arguments as an array.

-Rest mean remaining values pack in one parameter.

function sum(...numbers) {

let total = 0;

for (let number of numbers) {

total += number;

}

return total;

}

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

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

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

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

console.log(second); // Output: 2

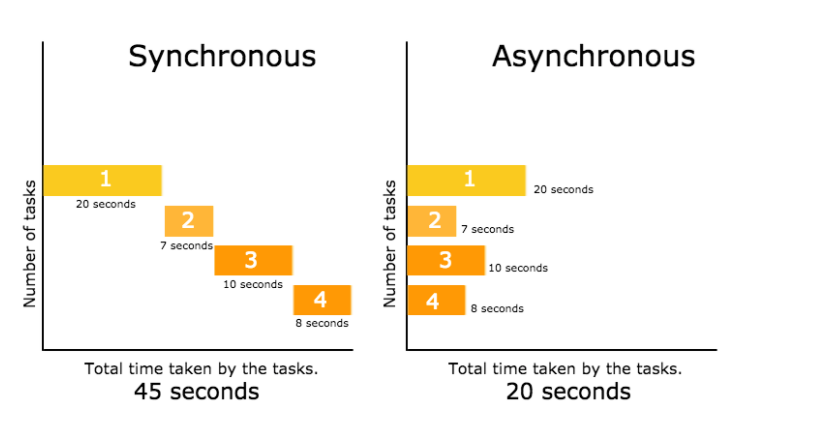
console.log(rest); // Output: [3, 4, 5]

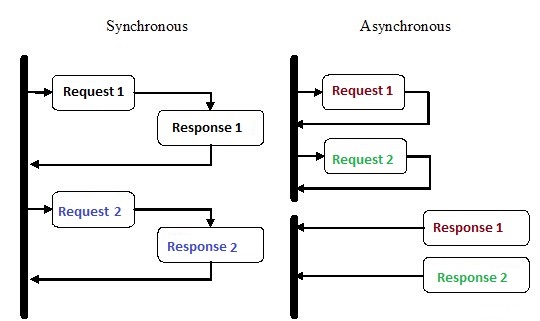
**OOPS IN JAVASCRIPT**

**Synchronous JavaScript**

-JavaScript is, by default, **single-threade**d and **synchronous**. This means it executes code line by line, one operation at a time.

-Functions running in **parallel** with other functions are called **asynchronous**





-In  a synchronous way the first operation is start first and complete its execution then the next operation is start after complete the first one. Next operation is blocked by the previous operation until complete it.

-But in asynchronous programming one task never wait for another task for execute. The task execute first which completed first. It never block one task input or output for another task.

**What is Synchronous Programming?**

-Synchronous programming is a way for computers to do things one step at a time, in the order they are given the instructions.

-Imagine you're cooking dinner and have a list of tasks, like boiling water for pasta, frying chicken, and making a salad.



-You would do these tasks one at a time and wait for each one to finish before moving to the next.

console.log("Start");  
console.log("Middle");  
console.log("End");

**Synchronous JavaScript**:

* Code executes in a predictable order, one line after the other.
* If a long-running operation is encountered, the entire program is blocked until it completes.

-Here's an example of synchronous code in JavaScript:

**// Define three functions**

**function** **firstTask()** **{**

console**.log("Task 1");**

**}**

**function** **secondTask()** **{**

console**.log("Task 2");**

**}**

**function** **thirdTask()** **{**

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

**}**

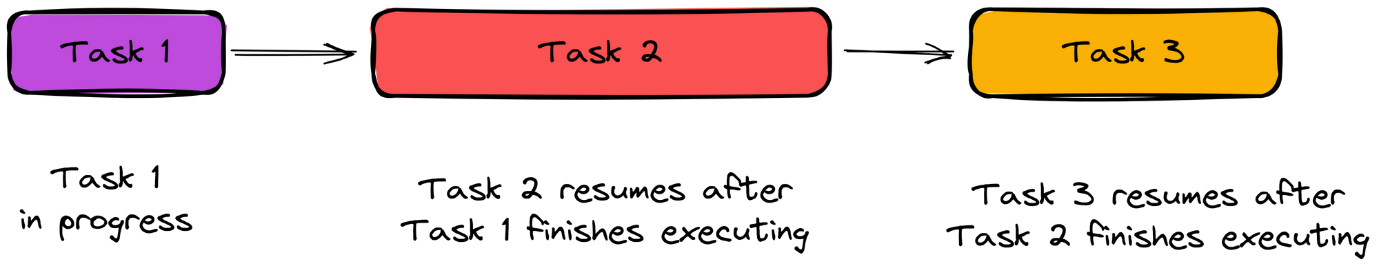
**// Execute the functions**

**firstTask();**

**secondTask();**

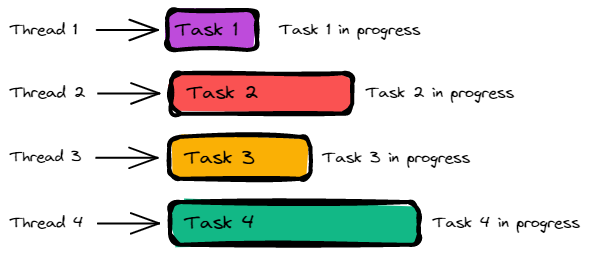
**thirdTask();**

-The code will execute the tasks in the order you see them and wait for each task to be completed before moving on to the next one.

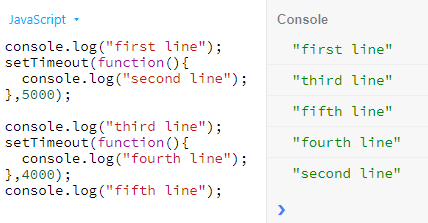


**WHAT IS ASYNCHRONOUS PROGRAMMING?**

-Asynchronous programming is a way for a computer program to handle multiple tasks simultaneously rather than executing them one after the other.

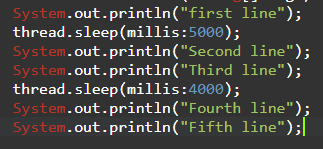
-Diagram showing how asynchronous programming works.

-Here's an example of an asynchronous program using the setTimeout method:



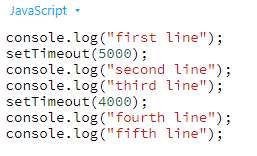
-JavaScript is, by default, single-threaded and synchronous. This means it executes code line by line, one operation at a time.

**Example 1: Synchronous programming java**



-In this example after execute first line it will wait for 5 sec, then second and third line will be executed. Also wait 4 second after third line and execute fourth and fifth line.

**Example 2: Asynchronous programming JavaScript.**



-In this example using JavaScript no line wait for another line. Here second line does not wait for first line and forth line does not wait after third line. These all lines are execute at run time.   No line blocked another line. That is non-blocking of asynchronous programming.

-There are different way to implement Asynchronous in JavaScript.

1. **Callback function**
2. **Promise**
3. **Generators**
4. **Async & Await.**

**1.Callback Function**

*"I will call back later!"*

-A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished.

// function

function greet(name, callback) {

console.log('Hi' + ' ' + name);

callback();

}

// callback function

function callMe() {

console.log('I am callback function');

}

// passing function as an argument

greet('Peter', callMe);

Output

Hi Peter

I am callback function



-Calling Multiple Functions using Set Timeout Function

<script>

    function person1(*name*,*person2*,*person3*,*person4*,*person5*,*person6*,*person7*,*person8*, *person9*,*person10*) {

            console.log("Person 1 is Busy with "+*name*);

            setTimeout(*person2*,4000)

            setTimeout(*person3*,6000)

            setTimeout(*person4*,7000)

            setTimeout(*person5*,5000)

            setTimeout(*person6*,5000)

            setTimeout(*person7*,5000)

            setTimeout(*person8*,5000)

            setTimeout(*person9*,5000)

            setTimeout(*person10*,5000)

        }

        function person2() {

            console.log("person 2 is Called");

        }

        function person3() {

            console.log("person 3 is Called" );

         }

        function person4() {

            console.log("person 4 is Called" );

        }

        function person5() {

            console.log("person 5 is Called" );

        }

        function person6() {

            console.log("person 6 is Called" );

        }

        function person7() {

            console.log("person 7 is Called" );

        }

        function person8() {

            console.log("person 8 is Called" );

        }

        function person9() {

            console.log("person 9 is Called" );

        }

        function person10() {

            console.log("person 10 is Called" );

        }

    person1("Amol",person2,person3,person4,person5,person6,person7,person8,person9,person10);

    </script>

**2.PROMISES**

- In JavaScript, a Promise is a built-in object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises are used to **handle asynchronous operations** more effectively compared to traditional callback-based approaches.

**Key Concepts of Promises:**

1. **States**:
   * **Pending**: The initial state of a promise. The operation has not yet completed.
   * **Fulfilled/Resolve/Successful**: The state when the operation completes successfully, and the promise has a resolved value.
   * **Rejected/NotFulfilled**: The state when the operation fails, and the promise has a reason for the failure (an error).
   * **Settled**:A promise is said to be *settled* if it is either fulfilled or rejected, but not pending.
2. **Promise Creation**: A promise is created using the Promise constructor,which takes an executor function with two arguments: resolve and reject.

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

// Asynchronous operation

let success = true;

if (success) {

resolve("Operation successful");

} else {

reject("Operation failed");

}

});

1. **Handling Promises**: You handle the results of a promise using .then() for successful completions and .catch() for errors.

myPromise.then(result => {

console.log(result); // "Operation successful"

}).catch(error => {

console.error(error); // "Operation failed"

});

1. **Chaining Promises**: Promises can be chained, allowing for multiple asynchronous operations to be performed in sequence.

myPromise

.then(result => {

console.log(result); // "Operation successful"

return anotherAsyncOperation(); // Returns a new promise

})

.then(nextResult => {

console.log(nextResult); // Result of the next async operation

})

.catch(error => {

console.error(error); // Error handling for any of the promises

});

1. **Promise.all()**: You can run multiple promises in parallel and wait for all of them to complete.

Promise.all([promise1, promise2, promise3])

.then(results => {

console.log(results); // Array of results

})

.catch(error => {

console.error(error); // If any of the promises fail

});

1. **Promise.race()**: You can run multiple promises in parallel and get the result of the first one that completes.

Promise.race([promise1, promise2, promise3])

.then(result => {

console.log(result); // Result of the first promise to resolve or reject

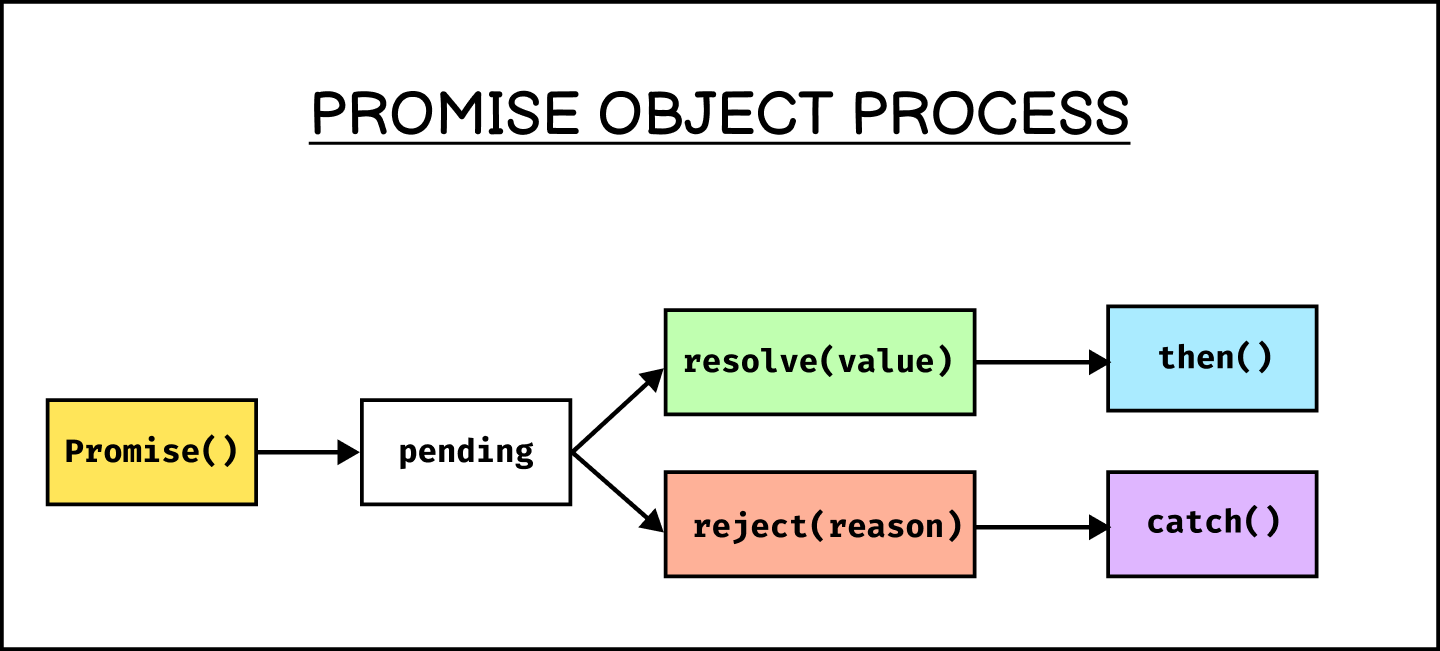
})

.catch(error => {

console.error(error); // Error from the first promise to reject

});

-Promises provide a cleaner, more manageable way to handle asynchronous operations compared to nested callbacks, making code more readable and easier to maintain.



**Promise()**

-The **Promise()** constructor creates [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) objects. It is primarily used to wrap callback-based APIs that do not already support promises.

<script>

        let promises = new Promise(function(*resolve*,*reject*){

            resolve();

            reject();

        })

        promises.then(

            function(*value*){

                console.log("Promise Resolved");

            }

        )

        promises.catch(()=>{

        console.log("Promise Rejected")

        })

    </script>

**Note:** Promise() can only be constructed with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new). Attempting to call it without new throws a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError).

JavaScript Promise Methods and Properties

|  |  |
| --- | --- |
| Name | Description |
| [Promise.all()](https://www.w3schools.com/jsref/jsref_promise_all.asp) | Returns a single Promise from a list of promises When all promises fulfill |
| [Promise.allSettled()](https://www.w3schools.com/jsref/jsref_promise_allsettled.asp) | Returns a single Promise from a list of promises When all promises sette |
| [Promise.any()](https://www.w3schools.com/jsref/jsref_promise_any.asp) | Returns a single Promise from a list of promises When any promise fulfills |
| [Promise.race()](https://www.w3schools.com/jsref/jsref_promise_race.asp) | Returns a single Promise from a list of promises When the faster promise settles |
| [Promise.reject()](https://www.w3schools.com/jsref/jsref_promise_reject.asp) | Returns a Promise object rejected with a value |
| [Promise.resolve()](https://www.w3schools.com/jsref/jsref_promise_resolve.asp) | Returns a Promise object resolved with a value |
| [catch()](https://www.w3schools.com/jsref/jsref_promise_catch.asp) | Provides a function to be called when a promise is rejected |
| [finally()](https://www.w3schools.com/jsref/jsref_promise_finally.asp) | Provides a function to be called when a promise is fulfilled or rejected |
| [then()](https://www.w3schools.com/jsref/jsref_promise_then.asp) | Provide two functions to be called when a promise is fulfilled or rejected |

**Promise.all()**

Syntax:-

Promise.all(iterable)

    <script>

        const recordVideoOne = new Promise((*resolve*,*reject*) =>{

*// resolve("Video 1 Recorded")*

            reject("Video 1 is Not Recorded");

        })

        const recordVideoTwo = new Promise((*resolve*,*reject*) =>{

*// resolve("Video 2 Recorded")*

            reject("Video 3 is Not Recorded");

        })

        const recordVideoThree = new Promise((*resolve*,*reject*) =>{

*// resolve("Video 3 Recorded")*

            reject("Video 3 is Not Recorded");

        })

        Promise.all([

            recordVideoOne,

            recordVideoTwo,

            recordVideoThree

        ]).then((*message*) =>{

            console.log(*message*)

        }).catch((*error*) =>{

            console.log(*error*)

        })

    </script>

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all#parameters)

[iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all#iterable)

An [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols" \l "the_iterable_protocol) (such as an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)) of promises.

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all#return_value)

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that is:

* **Already fulfilled**, if the iterable passed is empty.

# **JavaScript async/await**

*"async and await make promises easier to write"*

**async** makes a function return a Promise

**await** makes a function wait for a Promise

-We use the async keyword with a [function](https://www.programiz.com/javascript/function) to represent that the function is an asynchronous function. The async function returns a [promise](https://www.programiz.com/javascript/promise).

async function name(parameter1, parameter2, ...paramaterN) {

// statements

}

Here,

* **name** - name of the function
* **parameters** - parameters that are passed to the function

## JavaScript await Keyword

The await keyword is used inside the async function to wait for the asynchronous operation.

**DOM [Document Object Model] in JavaScript**

**Section 1. Getting started**

**What is Document Object Model (DOM)**

-The Document Object Model (DOM) is an application programming interface (API) for manipulating HTML documents.

-The DOM represents an HTML document as a tree of nodes. The DOM provides functions that allow you to add, remove, and modify parts of the document effectively.

-A document as a hierarchy of nodes

-The DOM represents an HTML document as a hierarchy of nodes. Consider the following HTML document:

-With the object model, JavaScript gets all the power it needs to create dynamic HTML:

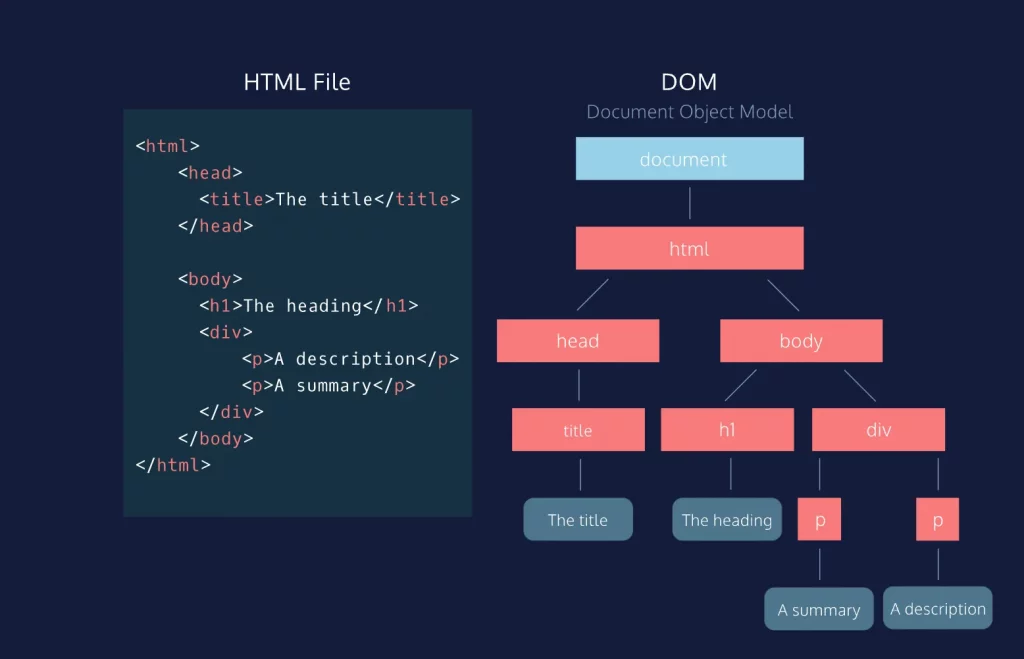
* JavaScript can change all the HTML attributes in the page
* JavaScript can change all the CSS styles in the page
* JavaScript can remove existing HTML elements and attributes
* JavaScript can add new HTML elements and attributes
* JavaScript can react to all existing HTML events in the page
* JavaScript can create new HTML events in the page

## Why is DOM required?

* Access HTML elements
* Replace HTML elements
* Add New HTML elements
* Delete HTML elements
* Change CSS of HTML elements
* Change attributes of HTML elements
* Add animation to HTML elements
* Add events to HTML elements

### **The HTML DOM Tree of Objects**





-In this DOM tree, the **document** is the root node. The root node has one child node which is the <html> element. The <html> element is called the *document element*.

-Each document can have only one document element. In an HTML document, the document element is the <html> element. Each markup can be represented by a node in the tree.

**Document Object Properties and Methods**

**The following properties and methods can be used on HTML documents:**

|  |  |
| --- | --- |
| Property / Method | Description |
| [activeElement](https://www.w3schools.com/jsref/prop_document_activeelement.asp) | Returns the currently focused element in the document |
| [addEventListener()](https://www.w3schools.com/jsref/met_document_addeventlistener.asp) | Attaches an event handler to the document |
| [adoptNode()](https://www.w3schools.com/jsref/met_document_adoptnode.asp) | Adopts a node from another document |
| [anchors](https://www.w3schools.com/jsref/coll_doc_anchors.asp) | [Deprecated](https://www.w3schools.com/jsref/coll_doc_anchors.asp) |
| [applets](https://www.w3schools.com/jsref/coll_doc_applets.asp) | [Deprecated](https://www.w3schools.com/jsref/coll_doc_applets.asp) |
| [baseURI](https://www.w3schools.com/jsref/prop_doc_baseuri.asp) | Returns the absolute base URI of a document |
| [body](https://www.w3schools.com/jsref/prop_doc_body.asp) | Sets or returns the document's body (the <body> element) |
| [charset](https://www.w3schools.com/jsref/prop_document_charset.asp) | [Deprecated](https://www.w3schools.com/jsref/prop_document_charset.asp) |
| [characterSet](https://www.w3schools.com/jsref/prop_document_characterset.asp) | Returns the character encoding for the document |
| [close()](https://www.w3schools.com/jsref/met_doc_close.asp) | Closes the output stream previously opened with document.open() |
| [cookie](https://www.w3schools.com/jsref/prop_doc_cookie.asp) | Returns all name/value pairs of cookies in the document |
| [createAttribute()](https://www.w3schools.com/jsref/met_document_createattribute.asp) | Creates an attribute node |
| [createComment()](https://www.w3schools.com/jsref/met_document_createcomment.asp) | Creates a Comment node with the specified text |
| [createDocumentFragment()](https://www.w3schools.com/jsref/met_document_createdocumentfragment.asp) | Creates an empty DocumentFragment node |
| [createElement()](https://www.w3schools.com/jsref/met_document_createelement.asp) | Creates an Element node |
| [createEvent()](https://www.w3schools.com/jsref/event_createevent.asp) | Creates a new event |
| [createTextNode()](https://www.w3schools.com/jsref/met_document_createtextnode.asp) | Creates a Text node |
| [defaultView](https://www.w3schools.com/jsref/prop_document_defaultview.asp) | Returns the window object associated with a document, or null if none is available. |
| [designMode](https://www.w3schools.com/jsref/prop_document_designmode.asp) | Controls whether the entire document should be editable or not. |
| [doctype](https://www.w3schools.com/jsref/prop_document_doctype.asp) | Returns the Document Type Declaration associated with the document |
| [documentElement](https://www.w3schools.com/jsref/prop_document_documentelement.asp) | Returns the Document Element of the document (the <html> element) |
| [documentMode](https://www.w3schools.com/jsref/prop_doc_documentmode.asp) | [Deprecated](https://www.w3schools.com/jsref/prop_doc_documentmode.asp) |
| [documentURI](https://www.w3schools.com/jsref/prop_document_documenturi.asp) | Sets or returns the location of the document |
| [domain](https://www.w3schools.com/jsref/prop_doc_domain.asp) | Returns the domain name of the server that loaded the document |
| [domConfig](https://www.w3schools.com/jsref/prop_document_domconfig.asp) | [Deprecated](https://www.w3schools.com/jsref/prop_document_domconfig.asp) |
| [embeds](https://www.w3schools.com/jsref/coll_doc_embeds.asp) | Returns a collection of all <embed> elements the document |
| [execCommand()](https://www.w3schools.com/jsref/met_document_execcommand.asp) | [Deprecated](https://www.w3schools.com/jsref/met_document_execcommand.asp) |
| [forms](https://www.w3schools.com/jsref/coll_doc_forms.asp) | Returns a collection of all <form> elements in the document |
| [getElementById()](https://www.w3schools.com/jsref/met_document_getelementbyid.asp) | Returns the element that has the ID attribute with the specified value |
| [getElementsByClassName()](https://www.w3schools.com/jsref/met_document_getelementsbyclassname.asp) | Returns an [HTMLCollection](https://www.w3schools.com/jsref/dom_obj_htmlcollection.asp) containing all elements with the specified class name |
| [getElementsByName()](https://www.w3schools.com/jsref/met_doc_getelementsbyname.asp) | Returns an live [NodeList](https://www.w3schools.com/jsref/dom_obj_html_nodelist.asp) containing all elements with the specified name |
| [getElementsByTagName()](https://www.w3schools.com/jsref/met_document_getelementsbytagname.asp) | Returns an [HTMLCollection](https://www.w3schools.com/jsref/dom_obj_htmlcollection.asp) containing all elements with the specified tag name |
| [hasFocus()](https://www.w3schools.com/jsref/met_document_hasfocus.asp) | Returns a Boolean value indicating whether the document has focus |
| [head](https://www.w3schools.com/jsref/prop_doc_head.asp) | Returns the <head> element of the document |
| [images](https://www.w3schools.com/jsref/coll_doc_images.asp) | Returns a collection of all <img> elements in the document |
| [implementation](https://www.w3schools.com/jsref/prop_document_implementation.asp) | Returns the DOMImplementation object that handles this document |
| [importNode()](https://www.w3schools.com/jsref/met_document_importnode.asp) | Imports a node from another document |
| [inputEncoding](https://www.w3schools.com/jsref/prop_document_inputencoding.asp) | [Deprecated](https://www.w3schools.com/jsref/prop_document_inputencoding.asp) |
| [lastModified](https://www.w3schools.com/jsref/prop_doc_lastmodified.asp) | Returns the date and time the document was last modified |
| [links](https://www.w3schools.com/jsref/coll_doc_links.asp) | Returns a collection of all <a> and <area> elements in the document that have a href attribute |
| [normalize()](https://www.w3schools.com/jsref/met_document_normalize.asp) | Removes empty Text nodes, and joins adjacent nodes |
| [normalizeDocument()](https://www.w3schools.com/jsref/met_document_normalizedocument.asp) | [Deprecated](https://www.w3schools.com/jsref/met_document_normalizedocument.asp) |
| [open()](https://www.w3schools.com/jsref/met_doc_open.asp) | Opens an HTML output stream to collect output from document.write() |
| [querySelector()](https://www.w3schools.com/jsref/met_document_queryselector.asp) | Returns the first element that matches a specified CSS selector(s) in the document |
| [querySelectorAll()](https://www.w3schools.com/jsref/met_document_queryselectorall.asp) | Returns a static [NodeList](https://www.w3schools.com/jsref/dom_obj_html_nodelist.asp) containing all elements that matches a specified CSS selector(s) in the document |
| [readyState](https://www.w3schools.com/jsref/prop_doc_readystate.asp) | Returns the (loading) status of the document |
| [referrer](https://www.w3schools.com/jsref/prop_doc_referrer.asp) | Returns the URL of the document that loaded the current document |
| [removeEventListener()](https://www.w3schools.com/jsref/met_document_removeeventlistener.asp) | Removes an event handler from the document (that has been attached with the [addEventListener()](https://www.w3schools.com/jsref/met_document_addeventlistener.asp) method) |
| [renameNode()](https://www.w3schools.com/jsref/met_document_renamenode.asp) | [Deprecated](https://www.w3schools.com/jsref/met_document_renamenode.asp) |
| [scripts](https://www.w3schools.com/jsref/coll_doc_scripts.asp) | Returns a collection of <script> elements in the document |
| [strictErrorChecking](https://www.w3schools.com/jsref/prop_document_stricterrorchecking.asp) | [Deprecated](https://www.w3schools.com/jsref/prop_document_stricterrorchecking.asp) |
| [title](https://www.w3schools.com/jsref/prop_doc_title.asp) | Sets or returns the title of the document |
| [URL](https://www.w3schools.com/jsref/prop_doc_url.asp) | Returns the full URL of the HTML document |
| [write()](https://www.w3schools.com/jsref/met_doc_write.asp) | Writes HTML expressions or JavaScript code to a document |
| [writeln()](https://www.w3schools.com/jsref/met_doc_writeln.asp) | Same as write(), but adds a newline character after each statement |

**Properties and Methods**

The following properties and methods can be used on all HTML elements:

|  |  |
| --- | --- |
| **Property / Method** | **Description** |
| [accessKey](https://www.w3schools.com/jsref/prop_html_accesskey.asp) | Sets or returns the accesskey attribute of an element |
| [addEventListener()](https://www.w3schools.com/jsref/met_element_addeventlistener.asp) | Attaches an event handler to an element |
| [appendChild()](https://www.w3schools.com/jsref/met_node_appendchild.asp) | Adds (appends) a new child node to an element |
| [attributes](https://www.w3schools.com/jsref/prop_node_attributes.asp) | Returns a [NamedNodeMap](https://www.w3schools.com/jsref/dom_obj_attributes.asp) of an element's attributes |
| [blur()](https://www.w3schools.com/jsref/met_html_blur.asp) | Removes focus from an element |
| [childElementCount](https://www.w3schools.com/jsref/prop_element_childelementcount.asp) | Returns an elements's number of child elements |
| [childNodes](https://www.w3schools.com/jsref/prop_node_childnodes.asp) | Returns a [NodeList](https://www.w3schools.com/jsref/dom_obj_html_nodelist.asp) of an element's child nodes |
| [children](https://www.w3schools.com/jsref/prop_element_children.asp) | Returns an [HTMLCollection](https://www.w3schools.com/jsref/dom_obj_htmlcollection.asp) of an element's child elements |
| [classList](https://www.w3schools.com/jsref/prop_element_classlist.asp) | Returns the class name(s) of an element |
| [className](https://www.w3schools.com/jsref/prop_html_classname.asp) | Sets or returns the value of the class attribute of an element |
| [click()](https://www.w3schools.com/jsref/met_html_click.asp) | Simulates a mouse-click on an element |
| [clientHeight](https://www.w3schools.com/jsref/prop_element_clientheight.asp) | Returns the height of an element, including padding |
| [clientLeft](https://www.w3schools.com/jsref/prop_element_clientleft.asp) | Returns the width of the left border of an element |
| [clientTop](https://www.w3schools.com/jsref/prop_element_clienttop.asp) | Returns the width of the top border of an element |
| [clientWidth](https://www.w3schools.com/jsref/prop_element_clientwidth.asp) | Returns the width of an element, including padding |
| [cloneNode()](https://www.w3schools.com/jsref/met_node_clonenode.asp) | Clones an element |
| [closest()](https://www.w3schools.com/jsref/met_element_closest.asp) | Searches the DOM tree for the closest element that matches a CSS selector |
| [compareDocumentPosition()](https://www.w3schools.com/jsref/met_node_comparedocumentposition.asp) | Compares the document position of two elements |
| [contains()](https://www.w3schools.com/jsref/met_node_contains.asp) | Returns true if a node is a descendant of a node |
| [contentEditable](https://www.w3schools.com/jsref/prop_html_contenteditable.asp) | Sets or returns whether the content of an element is editable or not |
| [dir](https://www.w3schools.com/jsref/prop_html_dir.asp) | Sets or returns the value of the dir attribute of an element |
| [firstChild](https://www.w3schools.com/jsref/prop_node_firstchild.asp) | Returns the first child node of an element |
| [firstElementChild](https://www.w3schools.com/jsref/prop_element_firstelementchild.asp) | Returns the first child element of an element |
| [focus()](https://www.w3schools.com/jsref/met_html_focus.asp) | Gives focus to an element |
| [getAttribute()](https://www.w3schools.com/jsref/met_element_getattribute.asp) | Returns the value of an element's attribute |
| [getAttributeNode()](https://www.w3schools.com/jsref/met_element_getattributenode.asp) | Returns an attribute node |
| [getBoundingClientRect()](https://www.w3schools.com/jsref/met_element_getboundingclientrect.asp) | Returns the size of an element and its position relative to the viewport |
| [getElementsByClassName()](https://www.w3schools.com/jsref/met_element_getelementsbyclassname.asp) | Returns a collection of child elements with a given class name |
| [getElementsByTagName()](https://www.w3schools.com/jsref/met_element_getelementsbytagname.asp) | Returns a collection of child elements with a given tag name |
| [hasAttribute()](https://www.w3schools.com/jsref/met_element_hasattribute.asp) | Returns true if an element has a given attribute |
| [hasAttributes()](https://www.w3schools.com/jsref/met_node_hasattributes.asp) | Returns true if an element has any attributes |
| [hasChildNodes()](https://www.w3schools.com/jsref/met_node_haschildnodes.asp) | Returns true if an element has any child nodes |
| [id](https://www.w3schools.com/jsref/prop_html_id.asp) | Sets or returns the value of the id attribute of an element |
| [innerHTML](https://www.w3schools.com/jsref/prop_html_innerhtml.asp) | Sets or returns the content of an element |
| [innerText](https://www.w3schools.com/jsref/prop_node_innertext.asp) | Sets or returns the text content of a node and its descendants |
| [insertAdjacentElement()](https://www.w3schools.com/jsref/met_node_insertadjacentelement.asp) | Inserts a new HTML element at a position relative to an element |
| [insertAdjacentHTML()](https://www.w3schools.com/jsref/met_node_insertadjacenthtml.asp) | Inserts an HTML formatted text at a position relative to an element |
| [insertAdjacentText()](https://www.w3schools.com/jsref/met_node_insertadjacenttext.asp) | Inserts text into a position relative to an element |
| [insertBefore()](https://www.w3schools.com/jsref/met_node_insertbefore.asp) | Inserts a new child node before an existing child node |
| [isContentEditable](https://www.w3schools.com/jsref/prop_html_iscontenteditable.asp) | Returns true if an element's content is editable |
| [isDefaultNamespace()](https://www.w3schools.com/jsref/met_node_isdefaultnamespace.asp) | Returns true if a given namespaceURI is the default |
| [isEqualNode()](https://www.w3schools.com/jsref/met_node_isequalnode.asp) | Checks if two elements are equal |
| [isSameNode()](https://www.w3schools.com/jsref/met_node_issamenode.asp) | Checks if two elements are the same node |
| [isSupported()](https://www.w3schools.com/jsref/met_node_issupported.asp) | [Deprecated](https://www.w3schools.com/jsref/met_node_issupported.asp) |
| [lang](https://www.w3schools.com/jsref/prop_html_lang.asp) | Sets or returns the value of the lang attribute of an element |
| [lastChild](https://www.w3schools.com/jsref/prop_node_lastchild.asp) | Returns the last child node of an element |
| [lastElementChild](https://www.w3schools.com/jsref/prop_element_lastelementchild.asp) | Returns the last child element of an element |
| [matches()](https://www.w3schools.com/jsref/met_element_matches.asp) | Returns true if an element is matched by a given CSS selector |
| [namespaceURI](https://www.w3schools.com/jsref/prop_node_namespaceuri.asp) | Returns the namespace URI of an element |
| [nextSibling](https://www.w3schools.com/jsref/prop_node_nextsibling.asp) | Returns the next node at the same node tree level |
| [nextElementSibling](https://www.w3schools.com/jsref/prop_element_nextelementsibling.asp) | Returns the next element at the same node tree level |
| [nodeName](https://www.w3schools.com/jsref/prop_node_nodename.asp) | Returns the name of a node |
| [nodeType](https://www.w3schools.com/jsref/prop_node_nodetype.asp) | Returns the node type of a node |
| [nodeValue](https://www.w3schools.com/jsref/prop_node_nodevalue.asp) | Sets or returns the value of a node |
| [normalize()](https://www.w3schools.com/jsref/met_node_normalize.asp) | Joins adjacent text nodes and removes empty text nodes in an element |
| [offsetHeight](https://www.w3schools.com/jsref/prop_element_offsetheight.asp) | Returns the height of an element, including padding, border and scrollbar |
| [offsetWidth](https://www.w3schools.com/jsref/prop_element_offsetwidth.asp) | Returns the width of an element, including padding, border and scrollbar |
| [offsetLeft](https://www.w3schools.com/jsref/prop_element_offsetleft.asp) | Returns the horizontal offset position of an element |
| [offsetParent](https://www.w3schools.com/jsref/prop_element_offsetparent.asp) | Returns the offset container of an element |
| [offsetTop](https://www.w3schools.com/jsref/prop_element_offsettop.asp) | Returns the vertical offset position of an element |
| [outerHTML](https://www.w3schools.com/jsref/prop_html_outerhtml.asp) | Sets or returns the content of an element (including the start tag and the end tag) |
| [outerText](https://www.w3schools.com/jsref/prop_node_outertext.asp) | Sets or returns the outer text content of a node and its descendants |
| [ownerDocument](https://www.w3schools.com/jsref/prop_node_ownerdocument.asp) | Returns the root element (document object) for an element |
| [parentNode](https://www.w3schools.com/jsref/prop_node_parentnode.asp) | Returns the parent node of an element |
| [parentElement](https://www.w3schools.com/jsref/prop_node_parentelement.asp) | Returns the parent element node of an element |
| [previousSibling](https://www.w3schools.com/jsref/prop_node_previoussibling.asp) | Returns the previous node at the same node tree level |
| [previousElementSibling](https://www.w3schools.com/jsref/prop_element_previouselementsibling.asp) | Returns the previous element at the same node tree level |
| [querySelector()](https://www.w3schools.com/jsref/met_element_queryselector.asp) | Returns the first child element that matches a CSS selector(s) |
| [querySelectorAll()](https://www.w3schools.com/jsref/met_element_queryselectorall.asp) | Returns all child elements that matches a CSS selector(s) |
| [remove()](https://www.w3schools.com/jsref/met_element_remove.asp) | Removes an element from the DOM |
| [removeAttribute()](https://www.w3schools.com/jsref/met_element_removeattribute.asp) | Removes an attribute from an element |
| [removeAttributeNode()](https://www.w3schools.com/jsref/met_element_removeattributenode.asp) | Removes an attribute node, and returns the removed node |
| [removeChild()](https://www.w3schools.com/jsref/met_node_removechild.asp) | Removes a child node from an element |
| [removeEventListener()](https://www.w3schools.com/jsref/met_element_removeeventlistener.asp) | Removes an event handler that has been attached with the addEventListener() method |
| [replaceChild()](https://www.w3schools.com/jsref/met_node_replacechild.asp) | Replaces a child node in an element |
| [scrollHeight](https://www.w3schools.com/jsref/prop_element_scrollheight.asp) | Returns the entire height of an element, including padding |
| [scrollIntoView()](https://www.w3schools.com/jsref/met_element_scrollintoview.asp) | Scrolls the an element into the visible area of the browser window |
| [scrollLeft](https://www.w3schools.com/jsref/prop_element_scrollleft.asp) | Sets or returns the number of pixels an element's content is scrolled horizontally |
| [scrollTop](https://www.w3schools.com/jsref/prop_element_scrolltop.asp) | Sets or returns the number of pixels an element's content is scrolled vertically |
| [scrollWidth](https://www.w3schools.com/jsref/prop_element_scrollwidth.asp) | Returns the entire width of an element, including padding |
| [setAttribute()](https://www.w3schools.com/jsref/met_element_setattribute.asp) | Sets or changes an attribute's value |
| [setAttributeNode()](https://www.w3schools.com/jsref/met_element_setattributenode.asp) | Sets or changes an attribute node |
| [style](https://www.w3schools.com/jsref/prop_html_style.asp) | Sets or returns the value of the style attribute of an element |
| [tabIndex](https://www.w3schools.com/jsref/prop_html_tabindex.asp) | Sets or returns the value of the tabindex attribute of an element |
| [tagName](https://www.w3schools.com/jsref/prop_element_tagname.asp) | Returns the tag name of an element |
| [textContent](https://www.w3schools.com/jsref/prop_node_textcontent.asp) | Sets or returns the textual content of a node and its descendants |
| [title](https://www.w3schools.com/jsref/prop_html_title.asp) | Sets or returns the value of the title attribute of an element |
| toString() | Converts an element to a string |

**DOM DOCUMENT PROPERTIES**

**1.activeElement**

-The activeElement property returns the HTML element that have focus.

-The activeElement property is read-only.

Return Value

|  |
| --- |
|  |
| Type | Description |
| Element | The HTML element that has focus |

Syntax: -

document.activeElement

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body onclick="myFunction()">

    <form action="">

        <input type="text" placeholder="input text">

    </form>

    <button>BUTTON</button>

    <h4 id="Head4"></h4>

    <script>

        function myFunction(){

            const element = document.activeElement.tagName;

            document.getElementById("Head4").innerHTML=element;

        }

    </script>

    </body>

</html>

**2.baseURI**

-The baseURI property returns the base URI of the document.

-The baseURI property is read-only.

Return Value

|  |  |
| --- | --- |
| Type | Description |
| String | The Base URI of the document. |

Syntax: -

document.baseURI

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! baseURI  -->*

    <h3 id="demo"></h3>

    <script>

*// Get the base URL of the document*

        var baseURL = document.baseURI;

*// Display the base URL*

        document.getElementById('demo').textContent = baseURL;

    </script>

</body>

</html>

**3.body**

-The body property sets or returns a document's <body> element.Return Value.

Return Value

|  |
| --- |
|  |
| Type | Description |
| Object | The body element of the document. |

Syntax: -

document.body

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! body  -->*

    <script>

        document.body.style.backgroundColor="black";

        document.body.style.color="white";

        document.body.innerHTML="Hello This Welcome To Javascript"

    </script>

</body>

</html>

**DOM DOCUMENT METHODS**

**1.** **addEventListener**

-The addEventListener() method attaches an event handler to a document.

Syntax: -

**document.addEventListener(*event*, *function*, *Capture*)**

Parameters

|  |  |
| --- | --- |
| Parameter | Description |
| *event* | Required. The event name. Do not use the "on" prefix. Use "click" instead of "onclick". All HTML DOM events are listed in the: [HTML DOM Event Object Reference](https://www.w3schools.com/jsref/dom_obj_event.asp). |
| *function* | Required. The function to run when the event occurs.  When the event occurs, an event object is passed to the function as the first parameter. The type of the event object depends on the specified event. For example, the "click" event belongs to the MouseEvent object. |
| *capture* | Optional (default = false). true - The handler is executed in the capturing phase. false - The handler is executed in the bubbling phase. |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>The Document Object</h1>

    <h2>The addEventListener() Method</h2>

    <p>Click anywhere in the document to display "Hello World!".</p>

        <p id="demo"></p>

    <div id="Circle">

    </div>

    <script>

        document.addEventListener("click",myFunction)

        function myFunction(){

            document.getElementById("demo").innerHTML = "Hello World!";

        }

    </script>

    <script>

        document.addEventListener("click",myFunction2)

        function myFunction2(){

          let Circle = document.getElementById("Circle");

          Circle.style.backgroundColor = "red";

          Circle.style.height = "200px";

          Circle.style.width = "200px";

          Circle.style.border = "1px solid";

          Circle.style.borderRadius = "50%";

        }

    </script>

</body>

</html>

**2.open()**

-The open() method opens a document for writing.

-All existing document content will be cleared.

Syntax: -

**document.open();**

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Document</title>

  </head>

  <body>

    <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Aperiam sed ex quis odio ducimus atque necessitatibus, officiis aliquam. Mollitia rem quidem velit dolore expedita!

    </p>

    <button onclick="myFunction()">OPEN</button>

    <script>

        function myFunction(){

            document.open();

            document.write("<h1>Hello This Is Heading<h1>");

        }

    </script>

  </body>

</html>

**3.close()**

-The close() method closes a window previously opened with the open() method.

Syntax: -

**document.close();**

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Document</title>

  </head>

  <body>

    <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Aperiam sed ex quis odio ducimus atque necessitatibus, officiis aliquam. Mollitia rem uidem velit dolore expedita!

    </p>

    <button onclick="myFunction()">OPEN</button>

    <script>

        function myFunction(){

            document.open();

            document.write("<h1>Hello This Is Heading<h1>");

            document.close();

        }

    </script>

  </body>

</html>

**4.createAttribute(name)**

-The createAttribute() method creates an attribute and returns the attribute as an Attr object.

Syntax: -

**document.createAttribute(name);**

Parameters

|  |  |
| --- | --- |
| Parameter | Description |
| *name* | Required. The name of the attribute to create. |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <style>

        .demo{

            color: red;

        }

    </style>

</head>

<body>

    <h1>The Document Object</h1>

    <h2>The createAttribute() and setAttributeNode() Methods</h2>

    <p>Click "Add" to create a class attribute and add it to first h1 element.</p>

    <button onclick="myFunction()">Add</button>

    <script>

        function myFunction(*params*) {

*// Create a class attribute:*

            let att = document.createAttribute("class");

*// Set a value of the class attribute*

            att.value = "demo";

*// Add the class attribute to the first h1;*

            let h1 = document.getElementsByTagName("h1")[0];

            h1.setAttributeNode(att);

        }

    </script>

</body>

</html>

**5.createElement(name)**

-The createElement() method creates an element node.

Syntax: -

**document.createElement(type);**

Parameters

|  |
| --- |
|  |
| Parameter | Description |
| *type* | Required. The type of element to create. |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! createElement()  -->*

    <script>

        var a = document.createElement("h1");

        console.log(a);

        a.innerHTML = "Hello, World!";

*// Append to body:*

        document.body.appendChild(a);

    </script>

</body>

</html>

**6.createComment(text)**

-The createComment() method creates a comment and returns the comment node.

Syntax: -

**document.createComment(text);**

|  |  |
| --- | --- |
| Parameter | Description |
| *text* | Optional. The comment text. |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! createComments()  -->*

*<!--? The createComment() method creates a comment and returns the comment node. -->*

    <script>

        var a = document.createComment("This Is Comments");

        console.log(a);

    </script>

</body>

</html>

**7.createComment(text)**

-The createComment() method creates a comment and returns the comment node.

Syntax: -

**document.createComment(text);**

|  |  |
| --- | --- |
| Parameter | Description |
| *text* | Optional. The comment text. |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

*<!--! createComments()  -->*

*<!--? The createComment() method creates a comment and returns the comment node. -->*

    <script>

        var a = document.createComment("This Is Comments");

        console.log(a);

    </script>

</body>

</html>

**8.contains()**

The contains() method returns true if a node is a descendant of a node.

The contains() method returns false if not.

Syntax: -

**node.contains(node)**

**Parameters**

|  |
| --- |
|  |
| Parameter | Description |
| *node* | Required. The node that may be a descendant of the node. |

**Return Value**

|  |  |  |
| --- | --- | --- |
| Type |  | Description |
| Boolean |  | true - The node is a descendant false - The node is NOT a descendant |

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <div id="myDIV">

        <p>I am a p element inside "myDIV", and I have a <span id="mySPAN"> <b>span</b></span> element inside of me.</p>

    </div>

    <h3 id="demo"></h3>

    <script>

    const span = document.getElementById("mySPAN");

    let answer = document.getElementById("myDIV").contains(span);

    document.getElementById("demo").innerHTML = answer;

    </script>

</body>

</html>

**HTML Nodes vs Elements**

-In the HTML DOM (Document Object Model), an HTML document is a collection of nodes with (or without) child nodes.

-Nodes are element nodes, text nodes, and comment nodes.

-Whitespace between elements are also text nodes.

-Elements are only element nodes.

**childNodes vs children**

-childNodes returns child nodes (element nodes, text nodes, and comment nodes).

-children returns child elements (not text and comment nodes).

**Siblings vs Element Siblings**

-Siblings are "brothers" and "sisters".

-Siblings are nodes with the same parent (in the same childNodes list).

-Element Siblings are elements with the same parent (in the same children list).

**Section 2. Selecting elements**

* [getElementById()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementbyid/) – select an element by id.
* [getElementsByName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbyname/) – select elements by name.
* [getElementsByTagName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbytagname/)  – select elements by a tag name.
* [getElementsByClassName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbyclassname/) – select elements by one or more class names.
* [querySelector()](https://www.javascripttutorial.net/javascript-dom/javascript-queryselector/)  – select elements by CSS selectors.

**HTML DOM EVENTS**

- HTML events are **"things"** that happen to HTML elements.

-When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

-In JavaScript, an event is an occurrence such as a user clicking a button, hovering over an element, or a page finishing loading. These events trigger JavaScript code to execute, allowing developers to create interactive and dynamic web applications.

Key points about JavaScript events:

* Trigger:

An event is triggered by an action, like a user interaction or a system occurrence.

* Event Object:

When an event occurs, JavaScript creates an event object containing information about the event, such as the type of event, the element that triggered it, and other relevant details.

* Event Listener:

To respond to an event, you attach an event listener to an element. This listener is a function that gets executed when the specified event happens on that element.

* Event Handling:

The process of responding to an event is called event handling.

Common examples of JavaScript events:

* Mouse events: click, dblclick, mouseover, mouseout, mousemove
* Keyboard events: keydown, keyup, keypress
* Form events: submit, change, focus, blur
* Window events: load, resize, scroll

## JavaScript Event Handlers

Event handlers can be used to handle and verify user input, user actions, and browser actions:

* Things that should be done every time a page loads
* Things that should be done when the page is closed
* Action that should be performed when a user clicks a button
* Content that should be verified when a user inputs data
* And more ...

Many different methods can be used to let JavaScript work with events:

* HTML event attributes can execute JavaScript code directly
* HTML event attributes can call JavaScript functions
* You can assign your own event handler functions to HTML elements
* You can prevent events from being sent or being handled
* And more ...

## JavaScript Event Listeners

-In JavaScript, an event listener is a function that "listens" for a specific event to occur on a particular HTML element or the document itself. When the event happens, the associated function is executed.

How it works:

* Attaching an event listener: You use the addEventListener() method to attach an event listener to an element.

JavaScript

element.addEventListener(eventName, functionToExecute);

* eventName: The name of the event you want to listen for (e.g., click, mouseover, keydown).
* functionToExecute: The function that will be called when the event occurs.

Event triggering: When the specified event happens on the element, the browser triggers the event listener and executes the associated function.

Example:

JavaScript

const button = document.querySelector('button');

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

console.log('Button clicked!');

});

**Event Bulbling**

-Bottom To Top

-When we want to stop propagation child to parent then we have write the Stoppropagation child.

-When we write false.

**Event Capturing**

-Top to Bottom

-When we write true.

-When we want to stop propagation parent to child then we have write stopPropagation in Parent.

|  |  |  |
| --- | --- | --- |
| Event | Occurs When | Belongs To |
| [abort](https://www.w3schools.com/jsref/event_onabort_media.asp) | The loading of a media is aborted | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [afterprint](https://www.w3schools.com/jsref/event_onafterprint.asp) | A page has started printing | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [animationend](https://www.w3schools.com/jsref/event_animationend.asp) | A CSS animation has completed | [AnimationEvent](https://www.w3schools.com/jsref/obj_animationevent.asp) |
| [animationiteration](https://www.w3schools.com/jsref/event_animationiteration.asp) | A CSS animation is repeated | [AnimationEvent](https://www.w3schools.com/jsref/obj_animationevent.asp) |
| [animationstart](https://www.w3schools.com/jsref/event_animationstart.asp) | A CSS animation has started | [AnimationEvent](https://www.w3schools.com/jsref/obj_animationevent.asp) |
| [beforeprint](https://www.w3schools.com/jsref/event_onbeforeprint.asp) | A page is about to be printed | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [beforeunload](https://www.w3schools.com/jsref/event_onbeforeunload.asp) | Before a document is about to be unloaded | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [blur](https://www.w3schools.com/jsref/event_onblur.asp) | An element loses focus | [FocusEvent](https://www.w3schools.com/jsref/obj_focusevent.asp) |
| [canplay](https://www.w3schools.com/jsref/event_oncanplay.asp) | The browser can start playing a media (has buffered enough to begin) | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [canplaythrough](https://www.w3schools.com/jsref/event_oncanplaythrough.asp) | The browser can play through a media without stopping for buffering | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [change](https://www.w3schools.com/jsref/event_onchange.asp) | The content of a form element has changed | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [click](https://www.w3schools.com/jsref/event_onclick.asp) | An element is clicked on | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [contextmenu](https://www.w3schools.com/jsref/event_oncontextmenu.asp) | An element is right-clicked to open a context menu | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [copy](https://www.w3schools.com/jsref/event_oncopy.asp) | The content of an element is copied | [ClipboardEvent](https://www.w3schools.com/jsref/obj_clipboardevent.asp) |
| [cut](https://www.w3schools.com/jsref/event_oncut.asp) | The content of an element is cut | [ClipboardEvent](https://www.w3schools.com/jsref/obj_clipboardevent.asp) |
| [dblclick](https://www.w3schools.com/jsref/event_ondblclick.asp) | An element is double-clicked | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [drag](https://www.w3schools.com/jsref/event_ondrag.asp) | An element is being dragged | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [dragend](https://www.w3schools.com/jsref/event_ondragend.asp) | Dragging of an element has ended | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [dragenter](https://www.w3schools.com/jsref/event_ondragenter.asp) | A dragged element enters the drop target | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [dragleave](https://www.w3schools.com/jsref/event_ondragleave.asp) | A dragged element leaves the drop target | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [dragover](https://www.w3schools.com/jsref/event_ondragover.asp) | A dragged element is over the drop target | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [dragstart](https://www.w3schools.com/jsref/event_ondragstart.asp) | Dragging of an element has started | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [drop](https://www.w3schools.com/jsref/event_ondrop.asp) | A dragged element is dropped on the target | [DragEvent](https://www.w3schools.com/jsref/obj_dragevent.asp) |
| [durationchange](https://www.w3schools.com/jsref/event_ondurationchange.asp) | The duration of a media is changed | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [ended](https://www.w3schools.com/jsref/event_onended.asp) | A media has reach the end ("thanks for listening") | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [error](https://www.w3schools.com/jsref/event_onerror.asp) | An error has occurred while loading a file | [ProgressEvent](https://www.w3schools.com/jsref/obj_progressevent.asp), [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [focus](https://www.w3schools.com/jsref/event_onfocus.asp) | An element gets focus | [FocusEvent](https://www.w3schools.com/jsref/obj_focusevent.asp) |
| [focusin](https://www.w3schools.com/jsref/event_onfocusin.asp) | An element is about to get focus | [FocusEvent](https://www.w3schools.com/jsref/obj_focusevent.asp) |
| [focusout](https://www.w3schools.com/jsref/event_onfocusout.asp) | An element is about to lose focus | [FocusEvent](https://www.w3schools.com/jsref/obj_focusevent.asp) |
| [fullscreenchange](https://www.w3schools.com/jsref/event_fullscreenchange.asp) | An element is displayed in fullscreen mode | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [fullscreenerror](https://www.w3schools.com/jsref/event_fullscreenerror.asp) | An element can not be displayed in fullscreen mode | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [hashchange](https://www.w3schools.com/jsref/event_onhashchange.asp) | There has been changes to the anchor part of a URL | [HashChangeEvent](https://www.w3schools.com/jsref/obj_hashchangeevent.asp) |
| [input](https://www.w3schools.com/jsref/event_oninput.asp) | An element gets user input | [InputEvent](https://www.w3schools.com/jsref/obj_inputevent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [invalid](https://www.w3schools.com/jsref/event_oninvalid.asp) | An element is invalid | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [keydown](https://www.w3schools.com/jsref/event_onkeydown.asp) | A key is down | [KeyboardEvent](https://www.w3schools.com/jsref/obj_keyboardevent.asp) |
| [keypress](https://www.w3schools.com/jsref/event_onkeypress.asp) | A key is pressed | [KeyboardEvent](https://www.w3schools.com/jsref/obj_keyboardevent.asp) |
| [keyup](https://www.w3schools.com/jsref/event_onkeyup.asp) | A key is released | [KeyboardEvent](https://www.w3schools.com/jsref/obj_keyboardevent.asp) |
| [load](https://www.w3schools.com/jsref/event_onload.asp) | An object has loaded | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [loadeddata](https://www.w3schools.com/jsref/event_onloadeddata.asp) | Media data is loaded | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [loadedmetadata](https://www.w3schools.com/jsref/event_onloadedmetadata.asp) | Meta data (like dimensions and duration) are loaded | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [loadstart](https://www.w3schools.com/jsref/event_onloadstart.asp) | The browser starts looking for the specified media | [ProgressEvent](https://www.w3schools.com/jsref/obj_progressevent.asp) |
| [message](https://www.w3schools.com/jsref/event_onmessage_sse.asp) | A message is received through the event source | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [mousedown](https://www.w3schools.com/jsref/event_onmousedown.asp) | The mouse button is pressed over an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mouseenter](https://www.w3schools.com/jsref/event_onmouseenter.asp) | The pointer is moved onto an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mouseleave](https://www.w3schools.com/jsref/event_onmouseleave.asp) | The pointer is moved out of an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mousemove](https://www.w3schools.com/jsref/event_onmousemove.asp) | The pointer is moved over an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mouseover](https://www.w3schools.com/jsref/event_onmouseover.asp) | The pointer is moved onto an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mouseout](https://www.w3schools.com/jsref/event_onmouseout.asp) | The pointer is moved out of an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| [mouseup](https://www.w3schools.com/jsref/event_onmouseup.asp) | A user releases a mouse button over an element | [MouseEvent](https://www.w3schools.com/jsref/obj_mouseevent.asp) |
| mousewheel | Deprecated. Use the [wheel](https://www.w3schools.com/jsref/event_onwheel.asp) event instead | [WheelEvent](https://www.w3schools.com/jsref/obj_wheelevent.asp) |
| [offline](https://www.w3schools.com/jsref/event_onoffline.asp) | The browser starts working offline | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [online](https://www.w3schools.com/jsref/event_ononline.asp) | The browser starts working online | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [open](https://www.w3schools.com/jsref/event_onopen_sse.asp) | A connection with the event source is opened | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [pagehide](https://www.w3schools.com/jsref/event_onpagehide.asp) | User navigates away from a webpage | [PageTransitionEvent](https://www.w3schools.com/jsref/obj_pagetransitionevent.asp) |
| [pageshow](https://www.w3schools.com/jsref/event_onpageshow.asp) | User navigates to a webpage | [PageTransitionEvent](https://www.w3schools.com/jsref/obj_pagetransitionevent.asp) |
| [paste](https://www.w3schools.com/jsref/event_onpaste.asp) | Some content is pasted in an element | [ClipboardEvent](https://www.w3schools.com/jsref/obj_clipboardevent.asp) |
| [pause](https://www.w3schools.com/jsref/event_onpause.asp) | A media is paused | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [play](https://www.w3schools.com/jsref/event_onplay.asp) | The media has started or is no longer paused | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [playing](https://www.w3schools.com/jsref/event_onplaying.asp) | The media is playing after being paused or buffered | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| popstate | The window's history changes | [PopStateEvent](https://www.w3schools.com/jsref/obj_popstateevent.asp) |
| [progress](https://www.w3schools.com/jsref/event_onprogress.asp) | The browser is downloading media data | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [ratechange](https://www.w3schools.com/jsref/event_onratechange.asp) | The playing speed of a media is changed | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [resize](https://www.w3schools.com/jsref/event_onresize.asp) | The document view is resized | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [reset](https://www.w3schools.com/jsref/event_onreset.asp) | A form is reset | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [scroll](https://www.w3schools.com/jsref/event_onscroll.asp) | An scrollbar is being scrolled | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [search](https://www.w3schools.com/jsref/event_onsearch.asp) | Something is written in a search field | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [seeked](https://www.w3schools.com/jsref/event_onseeked.asp) | Skipping to a media position is finished | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [seeking](https://www.w3schools.com/jsref/event_onseeking.asp) | Skipping to a media position is started | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [select](https://www.w3schools.com/jsref/event_onselect.asp) | User selects some text | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [show](https://www.w3schools.com/jsref/event_onshow.asp) | A <menu> element is shown as a context menu | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [stalled](https://www.w3schools.com/jsref/event_onstalled.asp) | The browser is trying to get unavailable media data | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| storage | A Web Storage area is updated | [StorageEvent](https://www.w3schools.com/jsref/obj_storageevent.asp) |
| [submit](https://www.w3schools.com/jsref/event_onsubmit.asp) | A form is submitted | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [suspend](https://www.w3schools.com/jsref/event_onsuspend.asp) | The browser is intentionally not getting media data | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [timeupdate](https://www.w3schools.com/jsref/event_ontimeupdate.asp) | The playing position has changed (the user moves to a different point in the media) | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [toggle](https://www.w3schools.com/jsref/event_ontoggle.asp) | The user opens or closes the <details> element | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [touchcancel](https://www.w3schools.com/jsref/event_touchcancel.asp) | The touch is interrupted | [TouchEvent](https://www.w3schools.com/jsref/obj_touchevent.asp) |
| [touchend](https://www.w3schools.com/jsref/event_touchend.asp) | A finger is removed from a touch screen | [TouchEvent](https://www.w3schools.com/jsref/obj_touchevent.asp) |
| [touchmove](https://www.w3schools.com/jsref/event_touchmove.asp) | A finger is dragged across the screen | [TouchEvent](https://www.w3schools.com/jsref/obj_touchevent.asp) |
| [touchstart](https://www.w3schools.com/jsref/event_touchstart.asp) | A finger is placed on a touch screen | [TouchEvent](https://www.w3schools.com/jsref/obj_touchevent.asp) |
| [transitionend](https://www.w3schools.com/jsref/event_transitionend.asp) | A CSS transition has completed | [TransitionEvent](https://www.w3schools.com/jsref/obj_transitionevent.asp) |
| [unload](https://www.w3schools.com/jsref/event_onunload.asp) | A page has unloaded | [UiEvent](https://www.w3schools.com/jsref/obj_uievent.asp), [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [volumechange](https://www.w3schools.com/jsref/event_onvolumechange.asp) | The volume of a media is changed (includes muting) | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [waiting](https://www.w3schools.com/jsref/event_onwaiting.asp) | A media is paused but is expected to resume (e.g. buffering) | [Event](https://www.w3schools.com/jsref/obj_event.asp) |
| [wheel](https://www.w3schools.com/jsref/event_onwheel.asp) | The mouse wheel rolls up or down over an element | [WheelEvent](https://www.w3schools.com/jsref/obj_wheelevent.asp) |

JavaScript Window Event Attributes

JavaScript *Window* events triggered for a window object and apply in <body> tag

|  |  |  |  |
| --- | --- | --- | --- |
| Attributes | Value | Description | In HTML5? |
|  |  |  |  |
| onafterprint | *js\_script* | Script is run after the document is printed | NEW |
| onbeforeprint | *js\_script* | Script is run before the document is printed | NEW |
| onbeforeunload | *js\_script* | Script is run before the document is unloaded | NEW |
| onerror | *js\_script* | Script is run when any error occur | NEW |
| onhaschange | *js\_script* | Script is run when document has changed | NEW |
| onload | *js\_script* | Event fires after the page loading finished |  |
| onmessage | *js\_script* | Script is run when document goes in offline | NEW |
| onoffline | *js\_script* | Script is run when document comes in online | NEW |
| onpagehide | *js\_script* | Script is run when document window is hidden | NEW |
| onpageshow | *js\_script* | Script is run when document window become visible | NEW |
| onpopstate | *js\_script* | Script is run when document window history changes | NEW |
| onredo | *js\_script* | Script is run when document perform redo | NEW |
| onresize | *js\_script* | Event fires when browser window is resized | NEW |
| onstorage | *js\_script* | Script is run when web storage area is updated | NEW |
| onundo | *js\_script* | Script is run when document performs undo | NEW |
| onunload | *js\_script* | Event fires when browser window has been closed |  |

JavaScript Form Events

JavaScript *Form* events triggered by perform some action inside HTML form elements.

|  |  |  |  |
| --- | --- | --- | --- |
| Attributes | Value | Description | In HTML5? |
| onblur | *js\_script* | Event fire when element loses focus |  |
| onchange | *js\_script* | Event fire when element value is changed |  |
| oncontextmenu | *js\_script* | Event fire when context menu is triggered | NEW |
| onfocus | *js\_script* | Event fire when element gets focus |  |
| onformchange | *js\_script* | Event fire when form changes | NEW |
| onforminput | *js\_script* | Event fire when form get input field |  |
| oninput | *js\_script* | Event fire when element get input field | NEW |
| oninvalid | *js\_script* | Event fire when element is invalid | NEW |
| onreset | *js\_script* | Event fire when clicked on form reset button | REMOVE |
| onselect | *js\_script* | Event fire after allow to select text in an element |  |
| onsubmit | *js\_script* | Event fire when form is submitted |  |

JavaScript Keyboard Events

|  |  |  |  |
| --- | --- | --- | --- |
| Attributes | Value | Description | In HTML5? |
| onkeydown | *js\_script* | Event fire when pressing a key |  |
| onkeypress | *js\_script* | Event fire when press a key |  |
| onkeyup | *js\_script* | Event fire when releases a key |  |

JavaScript Mouse Events

JavaScript *Mouse* events triggered by mouse action.

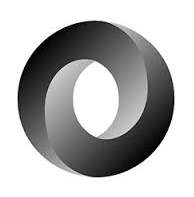
|  |  |  |  |
| --- | --- | --- | --- |
| Attributes | Value | Description | In HTML5? |
| onclick | *js\_script* | Event fire when mouse click on element |  |
| ondblclick | *js\_script* | Event fire when mouse double click on element |  |
| ondrag | *js\_script* | Script is run when element is dragged | NEW |
| ondragend | *js\_script* | Script is run at end of drag operation | NEW |
| ondragenter | *js\_script* | Script is run when element has dragged to a valid drop target | NEW |
| ondragleave | *js\_script* | Script is run when element leaves valid drop target | NEW |
| ondragover | *js\_script* | Script is run when element is dragged over on valid drop target | NEW |
| ondragstart | *js\_script* | Script is run at start of drag operation | NEW |
| ondrop | *js\_script* | Script is run when dragged element is dropped | NEW |
| onmousedown | *js\_script* | Event fire when mouse button is pressed down on element |  |
| onmousemove | *js\_script* | Event fire when mouse pointer moves over an element |  |
| onmouseout | *js\_script* | Event fire when mouse pointer moves out an element |  |
| onmouseover | *js\_script* | Event fire when mouse pointer moves over on element |  |
| onmouseup | *js\_script* | Event fire when mouse button is released over an element |  |
| onmousewheel | *js\_script* | Event fire when mouse wheel being rotated | NEW |
| onscroll | *js\_script* | Event fire when element scrollbar being scrolled | NEW |

JavaScript Media Events

JavaScript *Media* events triggered by common media elements like <img>, <audio>, <embed>, <object>, and <video>.

|  |  |  |  |
| --- | --- | --- | --- |
| Attributes | Value | Description | In HTML5? |
| onabort | *js\_script* | Script is run when element is abort |  |
| oncanplay | *js\_script* | Script is run when file is ready for start playing | NEW |
| oncanplaythrough | *js\_script* | Script is run when file is played all way without pausing for buffering | NEW |
| ondurationchange | *js\_script* | Script is run when media length changes | NEW |
| onemptied | *js\_script* | Script is run when something unavailable/disconnects | NEW |
| onended | *js\_script* | Script is run when media has reach to end position | NEW |
| onerror | *js\_script* | Script is run when error occurs file loaded time | NEW |
| onloadeddata | *js\_script* | Script is run when media is loaded | NEW |
| onloadedmetadata | *js\_script* | Script is run when meta data are loaded | NEW |
| onloadstart | *js\_script* | Script is run when file being loaded | NEW |
| onpause | *js\_script* | Script is run when media is paused | NEW |
| onplay | *js\_script* | Script is run when media is ready to start playing | NEW |
| onplaying | *js\_script* | Script is run when media is actually start for playing | NEW |
| onprogress | *js\_script* | Script is run when browser is process of getting media data | NEW |
| onratechange | *js\_script* | Script is run when playback rate changes | NEW |
| onreadystatechange | *js\_script* | Script is run when ready state changes for each time | NEW |
| onseeked | *js\_script* | Script is run when seeking attribute value set to false, that indicate seeking has ended | NEW |
| onseeking | *js\_script* | Script is run when seeking attribute value set to true, that indicate seeking has active | NEW |
| onstalled | *js\_script* | Script is run when browser is unable to fetch media data for any reason | NEW |
| onsuspend | *js\_script* | Script is run when fetching media data is stopped before it is completely loaded for any reason | NEW |
| ontimeupdate | *js\_script* | Script is run when playing position has changed | NEW |
| onvolumechange | *js\_script* | Script is run each time volume is changed | NEW |
| onwaiting | *js\_script* | Script is run when media has paused(for buffer more data) | NEW |

**JSON[JAVASCRIPT OBJECT NOTATION]**



**JSON**

JSON stands for **J**ava**S**cript **O**bject **N**otation

JSON is a **text format** for storing and transporting data

JSON is "self-describing" and easy to understand

**What is JSON?**

* JSON stands for **J**ava**S**cript **O**bject **N**otation
* JSON is a lightweight data-interchange format
* JSON is plain text written in JavaScript object notation
* JSON is used to send data between computers
* JSON is language independent **\***

-The JSON syntax is derived from JavaScript object notation, but the JSON format is text only.

**Why Use JSON?**

-The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.

-Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JSON.parse()

-JavaScript has a built in function for converting JSON strings into JavaScript objects

JSON.stringify()

-JavaScript also has a built in function for converting an object into a JSON string

-JSON makes it possible to store JavaScript objects as text.

## JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

* Data is in name/value pairs
* Data is separated by commas
* Curly braces hold objects
* Square brackets hold arrays

## JSON Data - A Name and a Value

-JSON data is written as name/value pairs (aka key/value pairs).

-A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

### Example

"name":"John"

*<!--! NORMAL OBJECT  -->*

    <script>

        let obj1 = {

            name : "Amol",

            age : 24,

            proff : "Software Engineer"

        }

        console.log(obj1);

    </script>

*<!--! JSON OBJECT  -->*

    <script>

        let obj2 = {

            "Key1" : "Value1",

            "Key2" : "Value2",

            "Key3" : "Value3"

        }

        console.log(obj2)

    </script>

// JSON syntax

{

"name": "John",

"age": 22,

"gender": "male",

}

-Below is a simple example −

{

"book": [

{

"id": "01",

"language": "Java",

"edition": "third",

"author": "Herbert Schildt"

},

{

"id": "07",

"language": "C++",

"edition": "second",

"author": "E.Balagurusamy"

}

]

}

-We have to create the JSON file with filename extension **.json**.

-In JSON File comments are not permitted(We cannot write the comments in JSON file).

JSON format supports the following data types −

|  |  |
| --- | --- |
| **Sr.No.** | **Type & Description** |
| 1 | **Number**  double- precision floating-point format in JavaScript |
| 2 | **String**  double-quoted Unicode with backslash escaping |
| 3 | **Boolean**  true or false |
| 4 | **Array**  an ordered sequence of values |
| 5 | **Value**  it can be a string, a number, true or false, null etc |
| 6 | **Object**  an unordered collection of key:value pairs |
| 7 | **Whitespace**  can be used between any pair of tokens |
| 8 | **null**  empty |

Try to find all the data types in this JSON example:

{

"student": {

"name": "Rumaisa Mahoney",

"age": 30,

"fullTime": true,

"languages": [ "JavaScript", "HTML", "CSS" ],

"GPA": 3.9,

"favoriteSubject": null

}

}

## JSON Strings

Strings in JSON must be written in double quotes.

### **Example**

{"name":"John"}

## JSON Numbers

Numbers in JSON must be an integer or a floating point.

### **Example**

{"age":30}

## JSON Objects

Values in JSON can be objects.

### **Example**

{  
"employee":{"name":"John", "age":30, "city":"New York"}  
}

Objects as values in JSON must follow the JSON syntax.

## JSON Arrays

Values in JSON can be arrays.

### **Example**

{  
"employees":["John", "Anna", "Peter"]  
}

## JSON Booleans

Values in JSON can be true/false.

### **Example**

{"sale":true}

## JSON null

Values in JSON can be null.

### **Example**

{"middlename":null}

## JSON Data

JSON data consists of **key/value** pairs similar to JavaScript object properties. The key and values are written in double quotes separated by a colon :. For example,

// JSON data

"name": "John"

**Note**: JSON data requires double quotes for the key.

## JSON Object

The JSON object is written inside curly braces { }. JSON objects can contain multiple **key/value** pairs. For example,

// JSON object

{ "name": "John", "age": 22 }

## JSON Array

JSON array is written inside square brackets [ ]. For example,

// JSON array

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

// JSON array containing objects

[

{ "name": "John", "age": 22 },

{ "name": "Peter", "age": 20 }.

{ "name": "Mark", "age": 23 }

]

**Note**: JSON data can contain objects and [arrays](https://www.programiz.com/javascript/array). However, unlike JavaScript objects, JSON data cannot contain [functions](https://www.programiz.com/javascript/function) as values.

## Accessing JSON Data

You can access JSON data using the dot notation. For example,

// JSON object

const data = {

"name": "John",

"age": 22,

"hobby": {

"reading" : true,

"gaming" : false,

"sport" : "football"

},

"class" : ["JavaScript", "HTML", "CSS"]

}

// accessing JSON object

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

console.log(data.hobby); // { gaming: false, reading: true, sport: "football"}

console.log(data.hobby.sport); // football

console.log(data.class[1]); // HTML

[Run Code](https://www.programiz.com/javascript/online-compiler)

We use the . notation to access JSON data. Its syntax is: variableName.key

You can also use square bracket syntax [] to access JSON data. For example,

// JSON object

const data = {

"name": "John",

"age": 22

}

// accessing JSON object

console.log(data["name"]); // John

[Run Code](https://www.programiz.com/javascript/online-compiler)

**JSON METHODS**

* **JSON.parse()** turns JSON into JavaScript
* **JSON.stringify()** turns JavaScript into JSON

**1.stringify(Normal ObjectName)**

-It is used to convert the normal object into the JSON object.

-It is the parameterized method it will accept the Normal object.

-Syntax:-

JSON.stringify(Normal ObjectName);

-For calling we write like **JSON.stringify(Normal ObjectName);**

## Converting JavaScript Object to JSON

You can also convert JavaScript objects to JSON format using the JavaScript built-in JSON.stringify() function. For example,

// JavaScript object

const jsonData = { "name": "John", "age": 22 };

// converting to JSON

const obj = JSON.stringify(jsonData);

// accessing the data

console.log(obj); // "{"name":"John","age":22}"

**2.parse(JSON ObjectName)**

-It is used to convert the JSON object into the Normal object.

-It is the parameterized method it will accept the JSON object.

-Syntax:-

JSON.stringify(Normal ObjectName);

-For calling we write like **JSON.parse(JSON ObjectName);**

## Converting JSON to JavaScript Object

You can convert JSON data to a JavaScript object using the built-in JSON.parse() function. For example,

// json object

const jsonData = '{ "name": "John", "age": 22 }';

// converting to JavaScript object

const obj = JSON.parse(jsonData);

// accessing the data

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

1. Check if what you are passing to JSON.parse() is a string

// ❌ ERROR!

const result = { name: "my object" };

console.log(typeof result);

// "object" ❌

const obj = JSON.parse(result);

// Error: "[object Object]" is not valid JSON

// ✅ WORKS!

const result = `{ "name": "my object" }`;

console.log(typeof result);

// "string" ✅

const obj = JSON.parse(result);

**JavaScript Objects VS JSON**

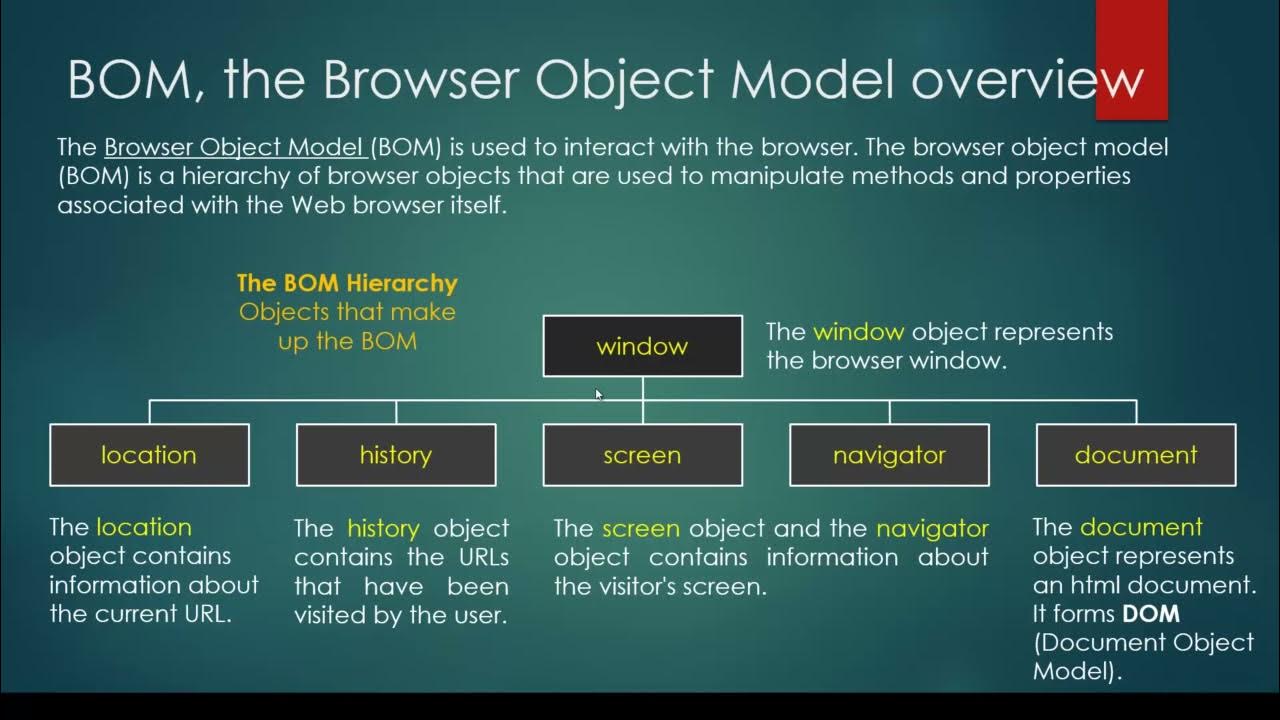
Though the syntax of JSON is similar to the JavaScript object, JSON is different from JavaScript objects.

|  |  |
| --- | --- |
| JSON | JavaScript Object |
| The key in key/value pair should be in double quotes. | The key in key/value pair can be without double quotes. |
| JSON cannot contain functions. | JavaScript objects can contain functions. |
| JSON can be created and used by other programming languages. | JavaScript objects can only be used in JavaScript. |

**JAVSCRIPT WINDOW – BROWSERS OBJECT MODEL**

-The **Browser Object Model** (BOM) is used to interact with the browser.

-The default object of browser is window means you can call all the functions of window by specifying window or directly.



-The Browser Object Model (BOM) in JavaScript refers to the objects provided by the browsers to interact with them. By using these objects, you can manipulate the browser's functionality. For example, you can get the browser history and window size, navigate to different URLs, etc.

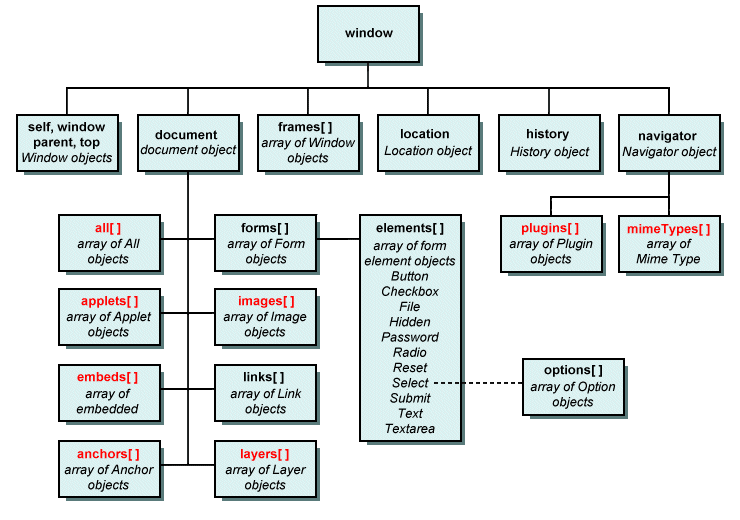
-The Browser Object Model (BOM) is the core of JavaScript on the web. The BOM provides you with objects that expose the web browser’s functionality.

-The BOM includes objects such as Window, Navigator, Location, History, and Screen. The Window object serves as the entry point to the BOM and provides access to the other objects within it.

Here, we have listed all objects of the Browser Object Model with descriptions −

* **Window** − The 'window' object represents the current browser window. You can use it to manipulate the browser window.
* **Document** − The 'document' object represents the currently opened web page in the browser window. You can use it to customize the property of the document.
* **Screen** − It provides information about the user's device's screen.
* **History** − It provides the browser's session history.
* **Navigator** − It is used to get the browser's information like default language, etc.
* **Location** − The Location object is used to get the URL information, such as the hostname of the current web page.
* **Console** − The console object allows developers to access the browser's console.

**1.JavaScript Window**



-The window object is global.The window object is supported by all browsers. It represents the browser's window.

The Window object serves as the entry point to the BOM and provides access to the other objects within it.

-All global JavaScript objects, functions, and variables automatically become members of the window object.

-Global variables are properties of the window object.

-Global functions are methods of the window object.

-Even the document object (of the HTML DOM) is a property of the window object.

-The [global object of JavaScript](https://www.javascripttutorial.net/es-next/javascript-globalthis/) in the web browser is the window object.

**The window object is global**

-The [global object of JavaScript](https://www.javascripttutorial.net/es-next/javascript-globalthis/) in the web browser is the window object. It means that all [variables](https://www.javascripttutorial.net/javascript-variables/) and [functions](https://www.javascripttutorial.net/javascript-function/) declared globally with the var keyword become the [properties](https://www.javascripttutorial.net/javascript-object-properties/) and methods of the window object. For example:

    <script>

       var counter = 1;

        var showCounter = () => console.log("counter "+counter);

        console.log(window.counter);

        window.showCounter();

    </script>

1

counter 1

-Because the counter variable and the showCounter() function are declared globally with the var keyword, they are automatically added to the window object.

-If you don’t want to pollute the window object, you can use the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword to declare variables and functions.

**Window Object Properties**

Below, we have covered all properties of the 'window' object with a description. You may use the 'window' as a reference to access these properties.

|  |  |
| --- | --- |
| Property Name | Property Description |
| closed | When the particular window is closed, it returns true. |
| console | It returns the window's console object. |
| customElements | It is used to define and access the custom elements in the browser window. |
| devicePixelRatio | It returns the physical pixel ratio of the device divided by CSS pixel ratio. |
| document | It is used to access the HTML document opened in the current window. |
| frames | It is used to get the window items like iframes, which are opened in the current window. |
| frameElement | It returns the current frame of the window. |
| history | It is used to get the history object of the window. |
| innerHeight | It returns the inner height of the window without including the scroll bar, toolbar, etc. |
| innerWidth | It returns the inner width of the window without including the scroll bar, toolbar, etc. |
| length | It returns the total number of iframes in the current window. |
| localStorage | It is used to access the local storage of the current window. |
| location | It is used to access the location object of the current window. |
| name | It is used to get or set the name of the window. |
| navigator | It is used to get the Navigator object of the browser. |
| opener | It returns a reference to the window from where the current window is opened. |
| outerHeight | It returns the total height of the window. |
| outerWidth | It returns the total width of the window. |
| pageXOffset | It returns the number of pixels you have scrolled the web page horizontally. |
| pageYOffset | It returns the number of pixels you have scrolled the web page vertically. |
| parent | It contains the reference to the parent window of the current window. |
| scheduler | It is entry point for using the prioritized task scheduling. |
| screen | It returns the 'screen' object of the current window. |
| screenLeft | It returns the position of the x-coordinate of the current window relative to the screen in pixels. |
| screenTop | It returns the position of the y-coordinate of the current window relative to the screen in pixels. |
| screenX | It is similar to the screenLeft property. |
| screenY | It is similar to the screenTop property. |
| scrollX | It is similar to the pageXOffset. |
| scrollY | It is similar to the pageYOffset. |
| self | It is used to get the current state of the window. |
| sessionStorage | It lets you access the 'sessionStorage' object of the current window. |
| speechSynthesis | It allows you to use the web speech API. |
| visualViewPort | It returns the object containing the viewport of the current window. |
| top | It contains a reference to the topmost window. |

**Window Object Methods**

-The 'window' object also contains methods like properties to manipulate the current browser window.

|  |  |
| --- | --- |
| Method Name | Method Description |
| alert() | It is used to show the alert message to the visitors. |
| atob() | It converts the string into the base-64 string. |
| blur() | It removes the focus from the window. |
| btoa() | It decodes the base-64 string in the normal string. |
| cancelAnimationFrame() | It cancels the animation frame scheduled using the requestAnimationFrame() method. |
| cancelIdleCallback() | It cancels a callback scheduled with the requestIdCallback() method. |
| clearImmediate() | It is used to clear actions specified using the setImmediate() method. |
| clearInterval() | It resets the timer you have set using the setInterval() method. |
| clearTimeout() | It stops the timeout you have set using the setTimeOut() method. |
| close() | It is used to close the current window. |
| confirm() | It shows the confirm box to get the confirmation from users. |
| focus() | It focuses on the current active window. |
| getComputedStyle() | It returns the current window's computed CSS style. |
| getSelection() | It returns the selection object based on the selected text range. |
| matchMedia() | It returns a new MediaQueryList object, which you can use to check whether the document matches the media queries. |
| moveBy() | It changes the position of the window relative to the current position. |
| moveTo() | It changes the position of the window absolutely. |
| open() | It opens a new window. |
| postMessage() | It is used to send a message to a window. |
| print() | It lets you print the window. |
| prompt() | It allows you to show a prompt box to get user input. |
| requestAnimationFrame() | It helps you to tell the browser that you want to perform an animation so the browser can update the animation before the next repaint. |
| requestIdleCallback() | It sets the callback functions to be called when the browser is Idle. |
| resizeBy() | It resizes the window by a particular number of pixels. |
| resizeTo() | It changes the size of the window. |
| scrollTo() | It scrolls the window to the absolute position. |
| scrollBy() | It scrolls the window relative to the current position. |
| setImmediate() | It breaks up long-running operations and runs the callback function instantly when the browser completes other operations. |
| setInterval() | It is used to execute a particular action after every interval. |
| setTimeout() | It is used to execute a particular action after a particular time. |
| stop() | It stops the loading of window. |

**ES6 FEATURES**

ECMAScript 2015 was the second major revision to JavaScript.

ECMAScript 2015 is also known as ES6 and ECMAScript 6.

## New Features in ES6

* [The let keyword](https://www.w3schools.com/js/js_es6.asp#mark_let)
* [The const keyword](https://www.w3schools.com/js/js_es6.asp#mark_const)
* [Arrow Functions](https://www.w3schools.com/js/js_es6.asp#mark_arrow)
* [The {a,b} = Operator](https://www.w3schools.com/js/js_es6.asp#mark_object_destructuring)
* [The [a,b] = Operator](https://www.w3schools.com/js/js_es6.asp#mark_array_destructuring)
* [The ... Operator](https://www.w3schools.com/js/js_es6.asp#mark_spread)[Spread]
* [For/of](https://www.w3schools.com/js/js_es6.asp#mark_forof)
* [Map Objects](https://www.w3schools.com/js/js_es6.asp#mark_map)
* [Set Objects](https://www.w3schools.com/js/js_es6.asp#mark_set)
* [Classes](https://www.w3schools.com/js/js_es6.asp#mark_class)
* Template Sttring
* [Promises](https://www.w3schools.com/js/js_es6.asp#mark_promise)
* [Symbol](https://www.w3schools.com/js/js_es6.asp#mark_symbol)
* [Default Parameters](https://www.w3schools.com/js/js_es6.asp#mark_param)
* [Function Rest Parameter](https://www.w3schools.com/js/js_es6.asp#mark_rest)
* [String.includes()](https://www.w3schools.com/js/js_es6.asp#mark_includes)
* [String.startsWith()](https://www.w3schools.com/js/js_es6.asp#mark_startswith)
* [String.endsWith()](https://www.w3schools.com/js/js_es6.asp#mark_endswith)
* [Array entries()](https://www.w3schools.com/js/js_es6.asp#mark_entries)
* [Array.from()](https://www.w3schools.com/js/js_es6.asp#mark_array_from)
* [Array keys()](https://www.w3schools.com/js/js_es6.asp#mark_array_keys)
* [Array find()](https://www.w3schools.com/js/js_es6.asp#mark_array_find)
* [Array findIndex()](https://www.w3schools.com/js/js_es6.asp#mark_array_findIndex)
* [Math.trunc](https://www.w3schools.com/js/js_es6.asp#mark_math_trunc)
* [Math.sign](https://www.w3schools.com/js/js_es6.asp#mark_math_sign)
* [Math.cbrt](https://www.w3schools.com/js/js_es6.asp#mark_math_cbrt)
* [Math.log2](https://www.w3schools.com/js/js_es6.asp#mark_math_log2)
* [Math.log10](https://www.w3schools.com/js/js_es6.asp#mark_math_log10)
* [Number.EPSILON](https://www.w3schools.com/js/js_es6.asp#mark_number_properties)
* [Number.MIN\_SAFE\_INTEGER](https://www.w3schools.com/js/js_es6.asp#mark_number_properties)
* [Number.MAX\_SAFE\_INTEGER](https://www.w3schools.com/js/js_es6.asp#mark_number_properties)
* [Number.isInteger()](https://www.w3schools.com/js/js_es6.asp#mark_number_isinteger)
* [Number.isSafeInteger()](https://www.w3schools.com/js/js_es6.asp#mark_number_issafeinteger)
* [New Global Methods](https://www.w3schools.com/js/js_es6.asp#mark_global_methods)
* [JavaScript Modules](https://www.w3schools.com/js/js_es6.asp#mark_modules)

**CLOSURE IN JAVSCRIPT**

What is Closure?

-The concept of **closures** in JavaScript allows nested functions to access variables defined in the scope of the parent function, even if the execution of the parent function is finished.

OR

-In JavaScript, closure provides access to the outer scope of a function from inside the inner function, even after the outer function has closed

-A JavaScript **closure** is basically a combination of the function and its **lexical environment**. This allows an inner function to access the outer function scope. A closure is created every time a function is -created at the function creation time.

The closure has three scope chains listed as follows:

* Access to its own scope.
* Access to the variables of the outer function.
* Access to the global variables.

**Example**

    <script>

*// javascript closure example*

*// outer function*

        function greet() {

*// variable defined outside the inner function*

        let name = 'John';

*// inner function*

            function displayName() {

*// accessing name variable*

*return* 'Hi' + ' ' + name;

            }

*return* displayName;

        }

        const g1 = greet();

        console.log(g1); *// returns the function definition*

        console.log(g1()); *// returns the value*

    </script>

**Output**

function displayName() {

// accessing name variable

return 'Hi' + ' ' + name;

}

Hi John

-In the above example, when greet() function is called, it returns the function definition of displayName.

-Here, g1 is a reference to the displayName() function.

-When g1() is called, it still has access to the greet() function.

-When we run console.log(g1), it returns the function definition.

**COERCION IN JAVSCRIPT**

-In JavaScript, coercion refers to the automatic or implicit conversion of a value from one data type to another. This happens when you perform operations on values of different types, and JavaScript attempts to make them compatible.

-There are two types of coercion in Javascript.

1]Implicit Coercion

2]Explicit Coercion.

**1]Implicit Coercion**

-Implicit Coercion: This is when JavaScript automatically converts a value without you explicitly asking for it.

-Interpreter automatically Converts.

 <script>

        let x = 5 + "10"; *// Output: "510" (number coerced to string)*

    </script>

**2]Explicit Coercion.**

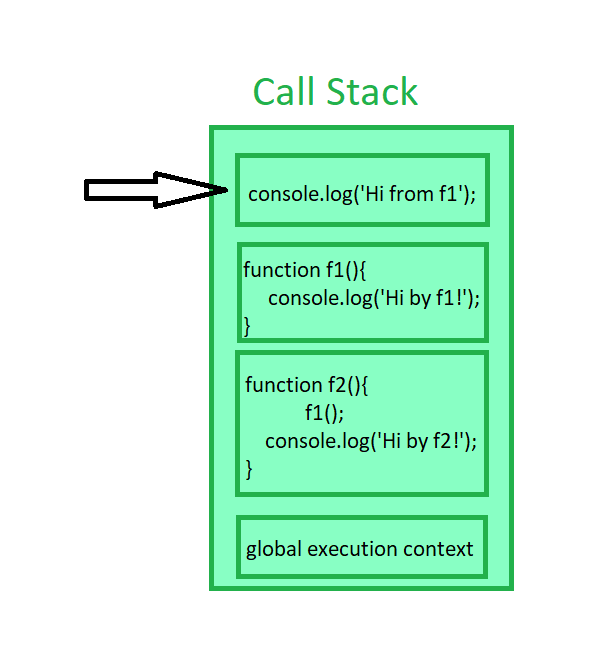
-Explicit Coercion: This is when you intentionally convert a value to a different type using built-in functions.

-This conversion you have do it by own to taking help of methods or operartors.

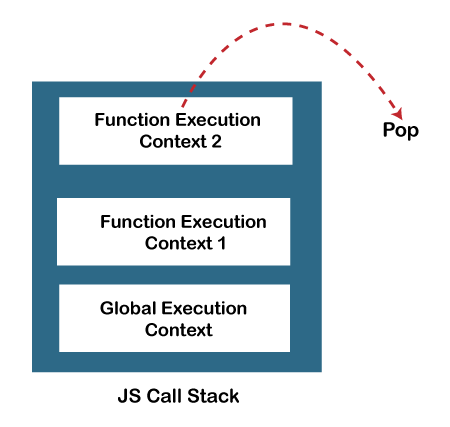
    <script>

             let y = Number("10"); *// Output: 10 (string coerced to number)*

    </script>



**CALL STACK**



-In JavaScript, the call stack is a data structure that keeps track of the execution context of functions. It operates on a Last-In-First-Out (LIFO) principle, meaning the last function called is the first one to finish executing.

    <script>

        function foo() {

            console.log("Inside foo");

            bar();

        }

        function bar() {

            console.log("Inside bar");

        }

        foo();

    </script>

Call Stack Progression:

* foo() is called, and its execution context is pushed onto the call stack.
* console.log("Inside foo") executes.
* bar() is called, and its execution context is pushed onto the stack.
* console.log("Inside bar") executes.
* bar() completes, its context is popped off the stack.
* foo() completes, its context is popped off the stack.

**-Why it's important:**

-Function Execution Order: The call stack ensures that functions are executed in the correct order.

-Error Tracking: When an error occurs, the call stack provides a trace of the function calls leading to the error, making debugging easier.

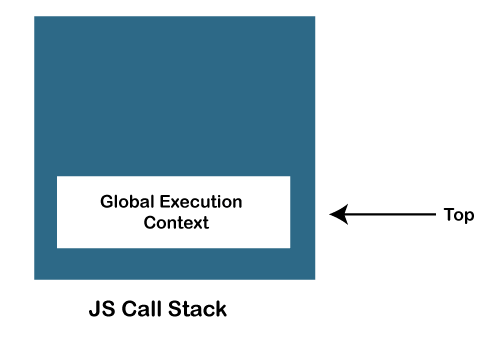
-Recursion: The call stack enables recursive function calls, where a function calls itself.

**Stack Overflow:**

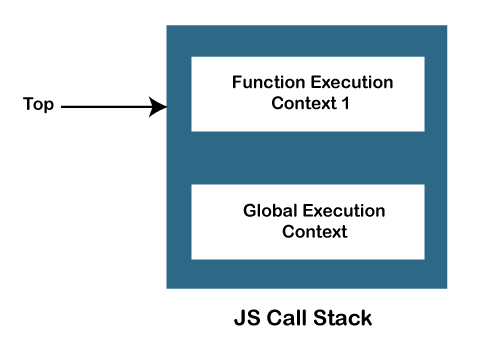
-If the call stack grows too large, it can lead to a "stack overflow" error. This often happens when a recursive function calls itself indefinitely without a proper base case to stop the recursion.

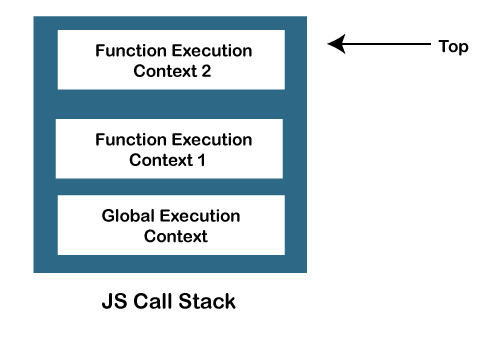
Role of JavaScript Call Stack

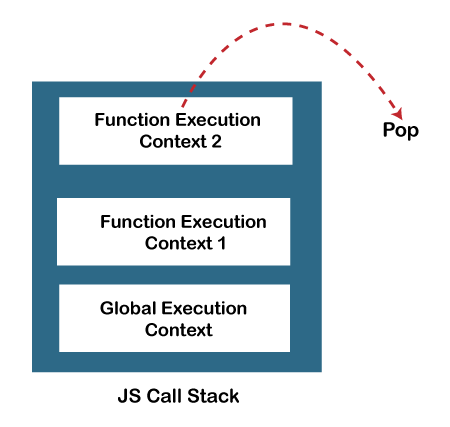
There are the following points where the call stack is being used by the JS engine:

-When any script is executed by the user, the JS engine creates a Global execution context and then adds it on the call stack and at the top of the stack so that it may get executed.  


-When any function is invoked, the JS engine creates a Function execution context and adds it on the stack and at the top of the stack so that the invoked function may get executed.



-In case a function invokes another function, the JS engine creates a Function execution context for the invoked function, adds it to the top of the stack, and begins the execution.  


When any function execution gets completed, the JS engine pops it out of the stack and continues the execution of the other functions stored in the stack.  


If no space is left in the stack and we try to push more functions, it throws a "stack overflow" error, and if no further execution context is present in the call stack, it throws a "Stack Underflow" error.

**JavaScript call stack example**

Let’s start with the following example:

function add(a, b) {

return a + b;

}

function average(a, b) {

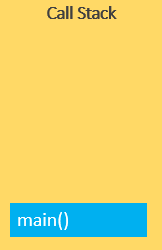
return add(a, b) / 2;

}

let x = average(10, 20);

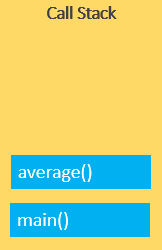
Code language: JavaScript (javascript)

When the JavaScript engine executes this script, it places the global execution context (denoted by main() or global() function on the call stack.



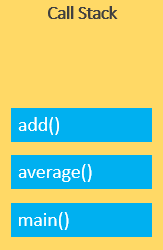
The global execution context enters the creation phase and moves to the execution phase.

-The JavaScript engine executes the call to the average(10, 20) function and creates a function execution context for the average() function and pushes it on top of the call stack:

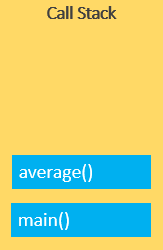


-The JavaScript engine starts executing the average() since because the average() function is on the top of the call stack.

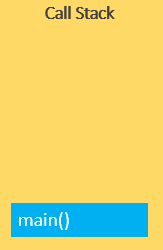
-The average() function calls add() function. At this point, the JavaScript engine creates another function execution context for the add() function and places it on the top of the call stack:



-JavaScript engine executes the add() function and pops it off the call stack:



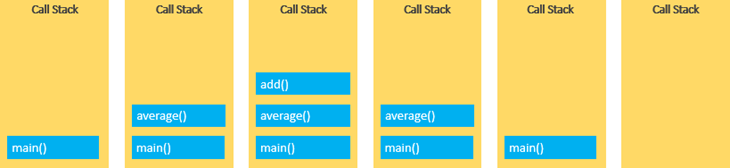
-At this point, the average() function is on the top of the call stack, the JavaScript engine executes and pops it off the call stack.



-Now, the call stack is empty so the script stops executing:



-The following picture illustrates the overall status of the Call Stack in all steps:



**JAVSCRIPT MODULES**

-JavaScript modules allow you to break up your code into separate files.

-This makes it easier to maintain a code-base.

-Modules are imported from external files with the import statement.

-Modules also rely on type="module" in the <script> tag.

-Before importing the module we have take care of it the module first exported.

**Export**

-Modules with **functions** or **variables** can be stored in any external file.

-There are two types of exports: **Named Exports** and **Default Exports**.

**Named Exports**

-Let us create a file named person.js, and fill it with the things we want to export.

-You can create named exports two ways. In-line individually, or all at once at the bottom.

**In-line individually:**

**person.js**

*export* const name = "Amol";

*export* const age = 24;

**All at once at the bottom:**

**person1.js**

const name = "Jesse";

const age = 40;

*export* {name, age};

**Default Exports**

-Let us create another file, named message.js, and use it for demonstrating default export.

-You can only have one default export in a file.

**message.js**

const message = () => {

        const name = "Amol";

        const age = 24;

*return* name + ' is ' + age + 'years old.';

    };

*export* *default* message;

**Import**

-You can import modules into a file in two ways, based on if they are named exports or default exports.

-Named exports are constructed using curly braces. Default exports are not.

**Import from named exports**

Import named exports from the file person.js:

**person.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1 id="demo"></h1>

    <script type="module">

*import* { name, age } *from* "../EXPORTS/person.js";

        let text = "My name is " + name + ", I am " + age + ".";

        document.getElementById("demo").innerHTML = text;

    </script>

</body>

</html>

**Import from default exports**

Import a default export from the file message.js:

**meassge.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>JavaScript Modules</h1>

    <p id="demo"></p>

    <script type="module">

*import* message *from* "../EXPORTS/message.js";

        document.getElementById("demo").innerHTML = message();

    </script>

</body>

</html>

## Note

Modules only work with the HTTP(s) protocol.

A web-page opened via the file:// protocol cannot use import / export.

**IMPORTANT QUESTIONS**

**QNO1:- is JavaScript and ECMAScript same?**

Ans:-

-Yes, ECMAScript and JavaScript are often used interchangeably, but they are technically different things:

* **ECMAScript**

-The standard for the core language of JavaScript. It's a programming language that defines syntax, operators, types, and more. ECMAScript also serves as a standard for other scripting languages, like JScript and ActionScript.

* **JavaScript**

-A scripting language that conforms to the ECMAScript specifications. JavaScript was created by Brendan Eich in 1995, and it was originally named Mocha and then LiveScript.

-ECMAScript is intended to ensure that web pages are compatible across different browsers. It's also used in non-browser environments, like Node.js.

-ECMAScript versions are often abbreviated as ES1, ES2, ES3, ES5, and ES6. Since 2016, versions have been named by year, like ECMAScript 2016, 2017, 2018, 2019, and 2020.

-JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

-ECMAScript is the official name of the language.

-ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.

-Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

**Differences Between JavaScript and ECMAScript**

1. Definition

* JavaScript - High-level languages and interpreted programming languages like JavaScript are frequently employed in designing websites. JavaScript executions, such as those in web browsers, adhere to the ECMAScript requirements while including environment-specific functionality.
* ECMAScript - A standardized programming language specification is called ECMAScript. It acts as a model for languages of scripting and establishes criteria for their implementation.

2. Execution

* JavaScript - This term describes the language used in browsers and other settings. It has environment-specific functionality in addition to ECMAScript features.
* ECMAScript - ECMAScript is the abbreviation for the standardized scripting language, which is not dependent on any specific environment.

3. Browser Compatibility

* JavaScript - Browsers use ECMAScript guidelines to execute JavaScript; nevertheless, browsers may differ in availability for the most recent ECMAScript features.
* ECMAScript - By offering a uniform collection of rules and norms, ECMAScript serves as a standard that ensures compatibility across distinct implementations.

4. Usage Scope

* JavaScript - Web developers utilize JavaScript, a programming language, for client-side and web-server scripting.
* ECMAScript - This is the name of the fundamental principle specification that outlines the fundamental characteristics of this programming language.

**Top 50 JavaScript Interview Questions With Example Answers**

**JavaScript Fundamentals**

**1. What is JavaScript?**

A high-level, interpreted programming language called JavaScript makes it possible to create interactive web pages and online apps with dynamic functionality. Commonly referred to as the universal language, Javascript is primarily used by developers for front-end and back-end work.

**2. What are the different data types in JavaScript?**

JavaScript has six primitive data types:

* Number
* String
* Boolean
* Null
* Undefined
* Symbol

It also has two compound data types:

* Object
* Array

**3. What is hoisting in JavaScript?**

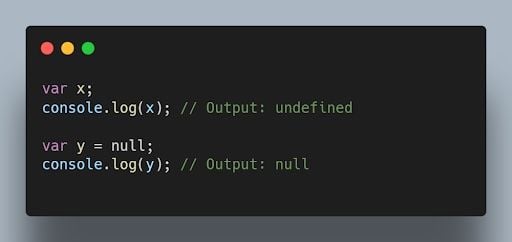
Hoisting is a JavaScript concept that refers to the process of moving declarations to the top of their scope. This means that variables and functions can be used before they are declared, as long as they are declared before they are used in a function.

For example, the following code will print "Hello, world!" even though the greeting variable is not declared until after the console.log() statement.

JavaScript code printing "Hello World" using hoisting. | Image: Akshay Kumar

**4. What is the difference between null and undefined?**

nullis an assignment value that represents [no value](https://builtin.com/software-engineering-perspectives/javascript-null-check) or an [empty value](https://builtin.com/software-engineering-perspectives/javascript-check-if-object-is-empty), while undefined is a variable that has been declared but not assigned a value.

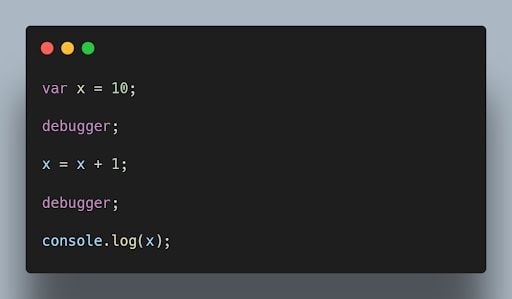
JavaScript code outputting null and undefined values. | Image: Akshay Kumar

**5. Why do we use the word “debugger” in JavaScript?**

The word “debugger” is used in JavaScript to refer to a tool that can be used to step through JavaScript code line by line. This can be helpful for debugging JavaScript code, which is the process of finding and fixing errors in JavaScript code. To use the debugger, you need to open the JavaScript console in your browser. Then, you can use debugger commands to comb through your code line by line.

It's essential to know debugging techniques as well as the more general ideas behind code optimization and speed improvement. In addition to operating smoothly, efficient code significantly enhances the user experience.

For example, the following code will print the value of the x variable at each step of the debugger.

JavaScript debugger code printing the value of x at each step. | Image: Akshay Kumar

**6. What is the purpose of the “this”keyword in JavaScript?**

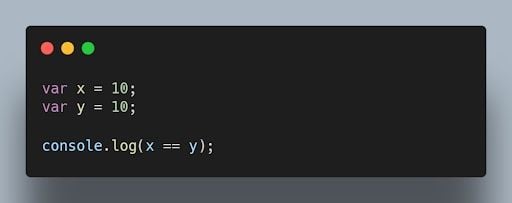
The this keyword refers to the object that is executing the current function or method. It allows access to object properties and methods within the context of that object.

JavaScript code using the this keyword to output person name. | Image: Akshay Kumar

**7. What is the difference between == and === operators in JavaScript?**

The equality == operator is a comparison operator that compares two values and returns true if they are equal. The strict equality === operator is also a comparison operator, but it compares two values and returns true only if they are equal and of the same type.

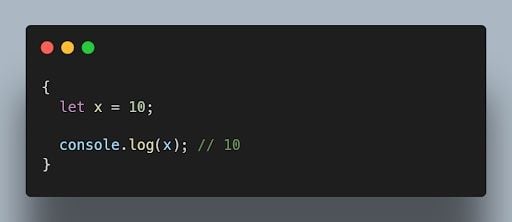
For example, the following code will return true, because the values of the x and y variables are equal.

JavaScript equality operator code comparing x and y variables. | Image: Akshay Kumar

**8. What is the difference between “var” and “let” keywords in JavaScript?**

The var and let keywords are both used to declare variables in JavaScript. However, there are some key differences between the two keywords.

The var keyword declares a global variable, which means that the variable can be accessed from anywhere in the code. The let keyword declares a local variable, which means that the variable can only be accessed within the block of code where it is declared.

JavaScript let keyword example. | Image: Akshay Kumar

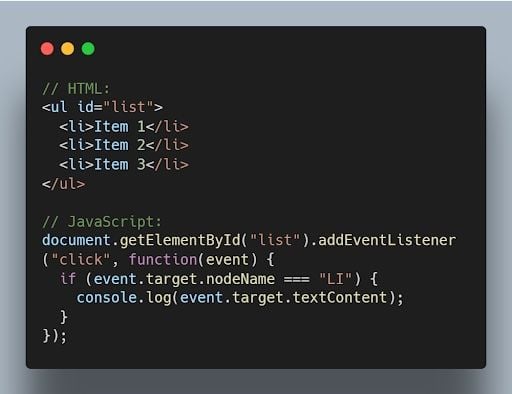
**9. What are closures in JavaScript?**

Closures (closureFn) are functions that have access to variables from an outer function even after the outer function has finished executing. They “remember” the environment in which they were created.

JavaScript closure code. | Image: Akshay Kumar

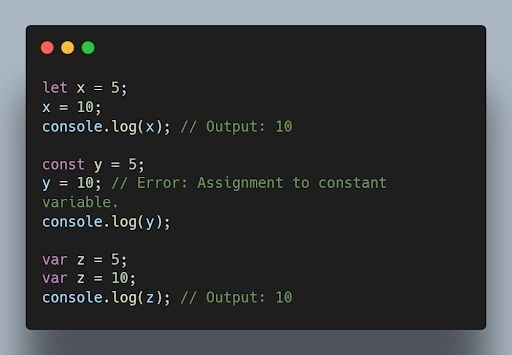
**10. What is event delegation in JavaScript?**

Event delegation is a technique where you attach a single event listener to a parent element, and that event listener handles events occurring on its child elements. It helps optimize performance and reduce memory consumption.

JavaScript event delegation code example. | Image: Akshay Kumar

**11. What is the difference between “let”, “const”, and “var”?**

let and const were introduced in ES6 and have block scope. let is reassignable, and const is non-reassignable. var is function-scoped and can be redeclared and reassigned throughout the function.

JavaScript let, const and var keywords with outputs. | Image: Akshay Kumar

**12. What is implicit type coercion in JavaScript?**

Implicit type coercion is a JavaScript concept that refers to the automatic conversion of a value from one type to another. In JavaScript, this conversion follows a priority order that typically begins with strings, then numbers, and finally booleans. If you try to add a string to a number, JavaScript will implicitly coerce the number to a string before performing the addition operation because strings have the highest priority in type coercion.

For example, when you combine the number 5 with the string '10' using the addition operator, the result is the string '510'. This occurs because JavaScript will implicitly convert the number 5 to a string following the priority of coercion, and then concatenate it to the string '10'.

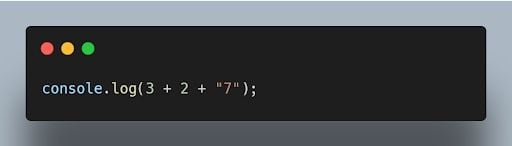
Implicit type coercion code example in JavaScript adding x and y variables. | Image: Raymond Van Hoecke

**13. Explain the concept of prototypes in JavaScript.**

Prototypes are a mechanism used by JavaScript objects for inheritance. Every JavaScript object has a prototype, which provides properties and methods that can be accessed by that object.

JavaScript prototype code example. | Image: Akshay Kumar

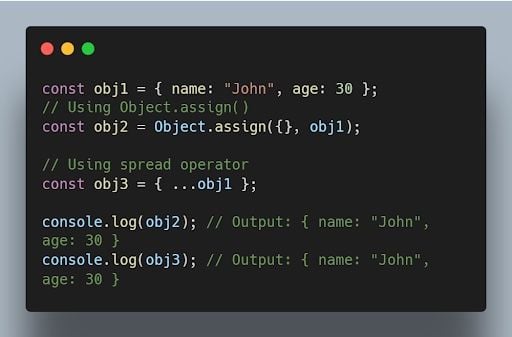
**14. What is the output of the following code?**

JavaScript console.log code. | Image: Akshay Kumar

The output will be "57". The addition operation is performed from left to right, and when a string is encountered, it performs concatenation.

**15. How can you clone an object in JavaScript?**

There are multiple ways to clone an object in JavaScript. One common method is using the Object.assign() method or the spread operator (...).

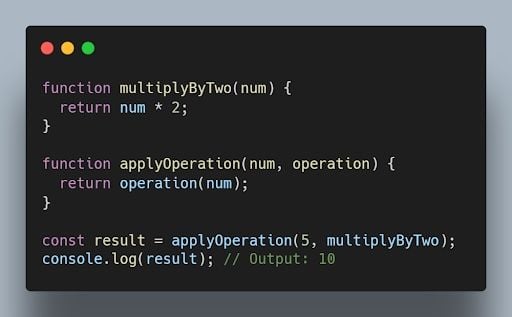
JavaScript code for cloning an object using object.assign() and ... operators. | Image: Akshay Kumar

More on JavaScript[JavaScript Question Mark (?) Operator Explained](https://builtin.com/software-engineering-perspectives/javascript-question-mark-operator)

**Intermediate Concepts**

**16. What are higher-order functions in JavaScript?**

Higher order functions are functions that can accept other functions as arguments or return functions as their results. They enable powerful functional programming patterns in JavaScript.

JavaScript higher order functions code. | Image: Akshay Kumar

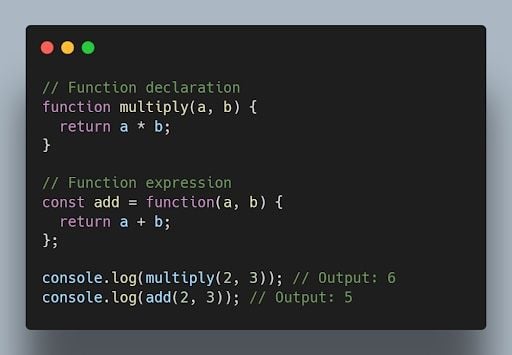
**17. What is the purpose of thebind() method in JavaScript?**

The bind() method is used to create a new function with a specified this value and an initial set of arguments. It allows you to set the context of a function permanently.

JavaScript bound() method code example. | Image: Akshay Kumar

**18. What is the difference between function declarations and function expressions?**

Function declarations are defined using the function keyword, while function expressions are defined by assigning a function to a variable. Function declarations are hoisted, while function expressions are not.

JavaScript code showing differences between declaration and expression. | Image: Akshay Kumar

**19. What are the different types of errors in JavaScript?**

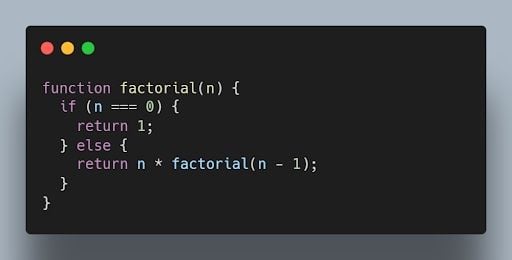
JavaScript can throw a variety of errors, including:

* **Syntax errors:** These errors occur when the JavaScript code is not syntactically correct.
* **Runtime errors:** These errors occur when the JavaScript code is executed and there is a problem.
* **Logical errors:** These errors occur when the JavaScript code does not do what it is supposed to do.

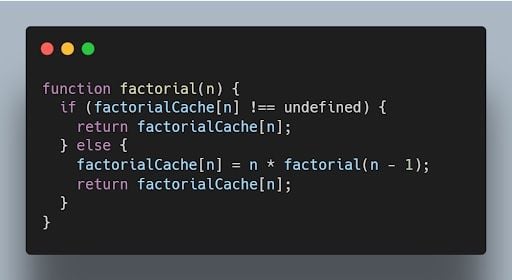
**20. What is memoization in JavaScript?**

Memoization is a technique that can be used to improve the performance of JavaScript code. Memoization works by storing the results of expensive calculations in a cache. This allows the JavaScript code to avoid re-performing the expensive calculations if the same input is provided again.

For example, the following code calculates the factorial of a number. The factorial of a number is the product of all the positive integers from one to the number.

JavaScript code to calculate the factorial of all positive integers from one to a number. | Image: Akshay Kumar

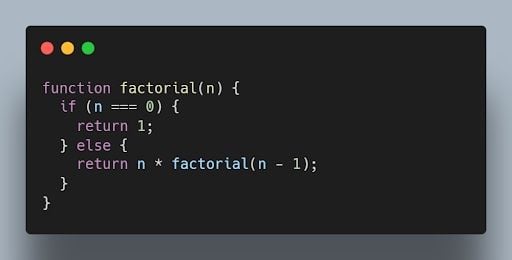
This code can be memoized as follows:

Memoized code for JavaScript factorial. | Image: Akshay Kumar

**21. What is recursion in JavaScript?**

Recursion is a programming technique that allows a function to call itself. Recursion can be used to solve a variety of problems, such as finding the factorial of a number or calculating the [Fibonacci sequence](https://builtin.com/data-science/fibonacci-sequence).

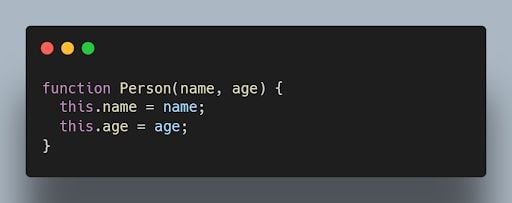
The following code shows how to use recursion to calculate the factorial of a number:

JavaScript recursion code to solve for factorial of a number. | Image: Akshay Kumar

**22. What is the use of a constructor function in JavaScript?**

A constructor function is a special type of function that is used to create objects. Constructor functions are used to define the properties and methods of an object.

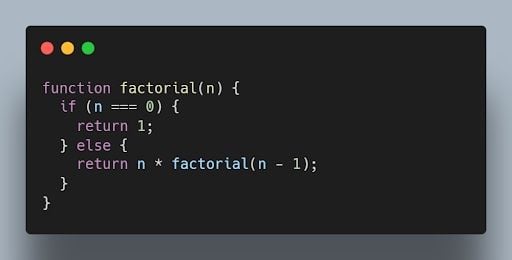
The following code shows how to create a constructor function:

Constructor function in JavaScript. | Image: Akshay Kumar

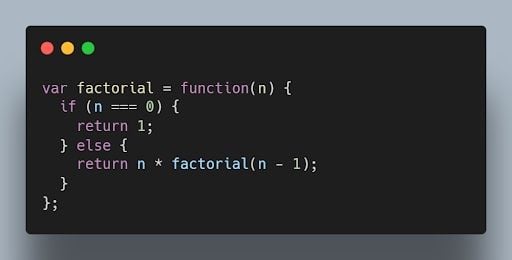
**23. What is the difference between a function declaration and a function expression in JavaScript?**

A function declaration is a statement that defines a function. A function expression is an expression that evaluates to a function.

The following code shows an example of a function declaration. This code defines a function named factorial. The factorial function calculates the factorial of a number.

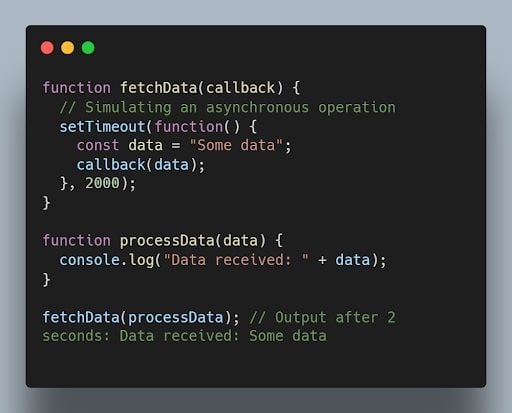
JavaScript function declaration code for a factorial function. | Image: Akshay Kumar

The following code shows an example of a function expression:

JavaScript function expression for factorial code. | Image: Akshay Kumar

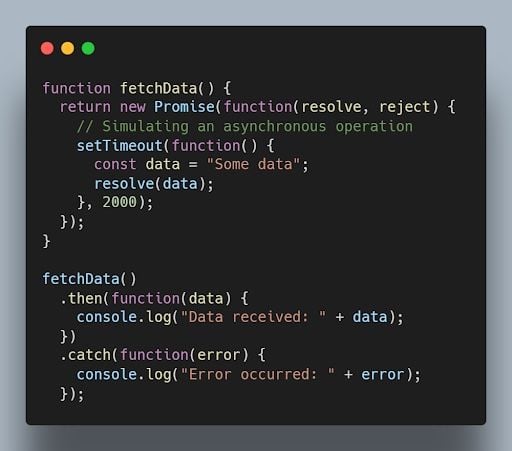
**24. What is a callback function in JavaScript?**

A callback function is a function passed as an argument to another function, which is then invoked inside the outer function. It allows asynchronous or event-driven programming.

JavaScript code for the callback function. | Image: Akshay Kumar

**25. What are promises in JavaScript?**

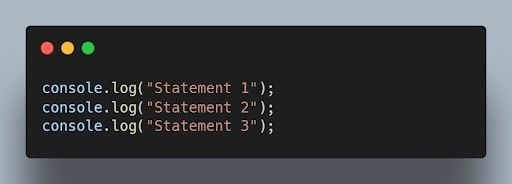
Promises are objects used for asynchronous operations. They represent the eventual completion or failure of an asynchronous operation and allow chaining and handling of success or error cases.

JavaScript promises code example. | Image: Akshay Kumar

**26. What is the difference between synchronous and asynchronous programming?**

In synchronous programming, the program execution occurs sequentially, and each statement blocks the execution until it is completed. In asynchronous programming, multiple tasks can be executed concurrently, and the program doesn’t wait for a task to finish before moving to the next one.

Synchronous coding example:

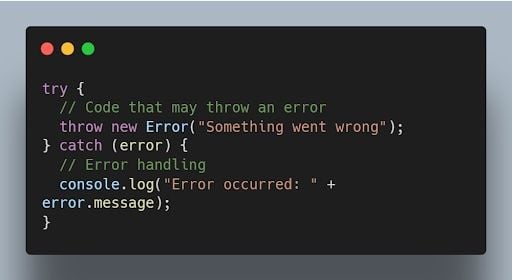
JavaScript synchronous code example. | Image: Akshay Kumar

Asynchronous code example:

Asynchronous JavaScript code example. | Image: Akshay Kumar

**27. How do you handle errors in JavaScript?**

Errors in JavaScript can be handled using try-catch blocks. The try block contains the code that may throw an error, and the catch block handles the error and provides an alternative execution path.

JavaScript try-catch blocks of code. | Image: Akshay Kumar

**28. Explain the concept of event bubbling in JavaScript.**

Event bubbling is the process where an event triggers on a nested element, and then the same event is propagated to its parent elements in the document object model (DOM) tree. It starts from the innermost element and goes up to the document root.

Example:

JavaScript code using event bubbling. | Image: Akshay Kumar

When you click on the child element, both the child and parent event handlers will be triggered, and the output will be:

JavaScript code output after clicking on the child element with event bubbling. | Image: Akshay Kumar

**29. What are arrow functions in JavaScript?**

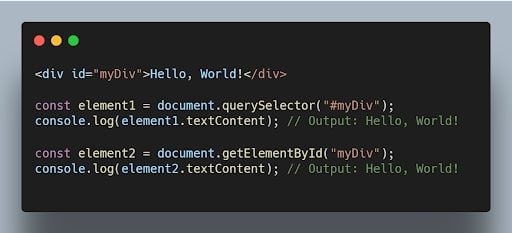
Arrow functions are a concise syntax for writing JavaScript functions. They have a more compact syntax compared to traditional function expressions and inherit the this value from their surrounding scope.

For example:

JavaScript arrow functions code example. | Image: Akshay Kumar

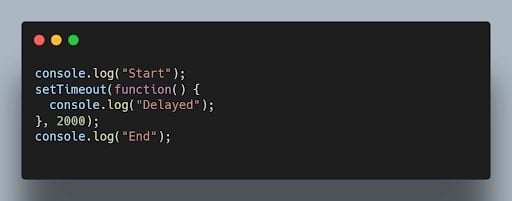
**30. What is the difference between querySelectorand getElementById?**

querySelector is a more versatile method that allows you to select elements using [CSS](https://builtin.com/software-engineering-perspectives/css)-like selectors, while getElementById specifically selects an element with the specified ID.

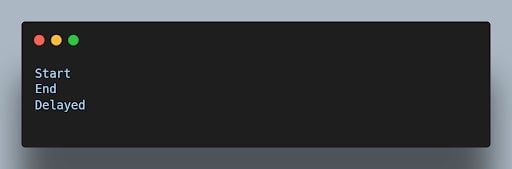
JavaScript code comparing querySelector and getElementByID methods. | Image: Akshay Kumar

**31. What is the purpose of the setTimeout() function in JavaScript?**

The setTimeout() function is used to delay the execution of a function or the evaluation of an expression after a specified amount of time in milliseconds.

JavaScript setTimeout() function code. | Image: Akshay Kumar

Output after two seconds:

JavaScript setTimeout code output after two seconds. | Image: Akshay Kumar

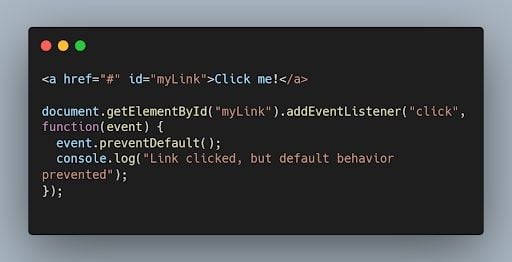
**32. What is event delegation and why is it useful?**

Event delegation is a technique where you attach a single event listener to a parent element to handle events occurring on its child elements. It’s useful for dynamically created elements or when you have a large number of elements.

JavaScript event delegation code example. | Image: Akshay Kumar

**33. How can you prevent the default behavior of an event in JavaScript?**

You can use the preventDefault() method on the event object within an event handler to prevent the default behavior associated with that event.

JavaScript preventDefault() method code example. | Image: Akshay Kumar

**34. What is the difference between localStorageand sessionStoragein JavaScript?**

Both localStorage and sessionStorage are web storage objects in JavaScript, but they have different scopes and lifetimes.

* localStorage persists data even after the browser window is closed and is accessible across different browser tabs/windows of the same origin.
* sessionStorage stores data for a single browser session and is accessible only within the same tab or window.

JavaScript localStorage and sessionStorage code comparisons. | Image: Akshay Kumar

**35. How can you convert a string to lowercase in JavaScript?**

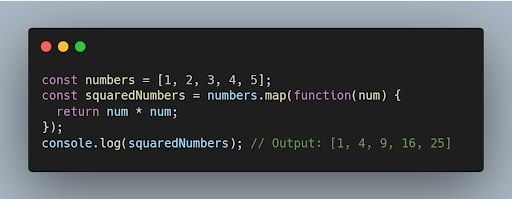
You can use the toLowerCase() method to convert a string to lowercase in JavaScript.

JavaScript toLowerCase() code example. | Image: Akshay Kumar

**Advanced Concepts**

**36. What is the purpose of the map() function in JavaScript?**

The map() function is used to [iterate over an array](https://builtin.com/software-engineering-perspectives/javascript-array-contains) and apply a transformation or computation on each element. It returns a new array with the results of the transformation.

JavaScript map() function code example. | Image: Akshay Kumar

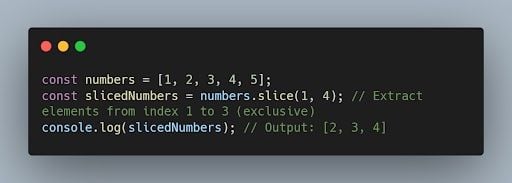
**37. What is the difference between splice() and slice()?**

* splice() is used to modify an array by adding, removing, or replacing elements at a specific position.
* slice() is used to create a new array that contains a portion of an existing array, specified by the starting and ending indices.

Example of splice():

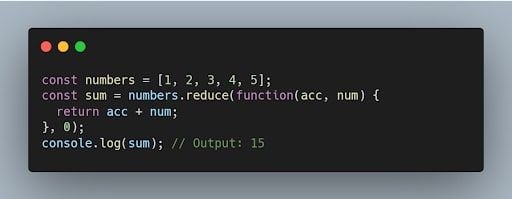
JavaScript splice() function code example. | Image: Akshay Kumar

Example of slice():

JavaScript slice() code example. | Image: Akshay Kumar

**38. What is the purpose of the reduce() function in JavaScript?**

The [reduce() function](https://builtin.com/software-engineering-perspectives/javascript-reduce) is used to reduce an array to a single value by applying a function to each element and accumulating the result.

JavaScript reduce() function example. | Image: Akshay Kumar

**39. How can you check if an array includes a certain value in JavaScript?**

You can use the includes() method to check if an array includes a specific value. It returns true if the value is found, and false otherwise.

JavaScript includes() method code on an array. | Image: Akshay Kumar

**40. What is the difference between prototype and instance properties in JavaScript?**

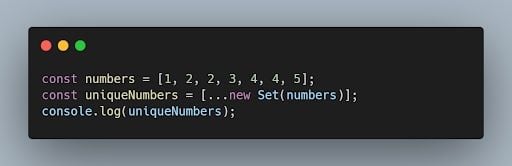
A prototype property is a property that is defined on the prototype object of a constructor function. Instance properties are properties that are defined on individual objects that are created by a constructor function.

Prototype properties are shared by all objects that are created by a constructor function. Instance properties are not shared by other objects.

**41. What is the difference between an array and an object in JavaScript?**

An array is a [data structure](https://builtin.com/software-engineering-perspectives/javascript-algorithms-and-data-structures) that can store a collection of values. An object is a data structure that can store a collection of properties.

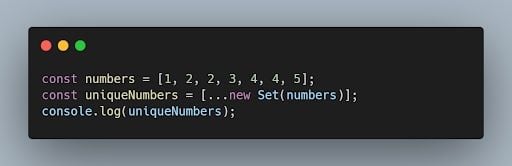
Arrays are indexed by numbers. Objects are indexed by strings. Arrays can only store primitive data types and objects. Objects can store primitive data types, objects and arrays.

JavaScript differences between array and object code example. | Image: Akshay Kumar

**42. How can you remove duplicates from an array in JavaScript?**

One way to remove duplicates from an array is by using the Set object or by using the [*filter() method*](https://builtin.com/software-engineering-perspectives/javascript-filter) with the indexOf() method.

Example:

JavaScript code example removing duplicates using the filter() method. | Image: Akshay Kumar

**43. What is the purpose of the fetch() function in JavaScript?**

The fetch() function is used to make asynchronous HTTP requests in JavaScript. It returns a Promise that resolves to the response from the server.

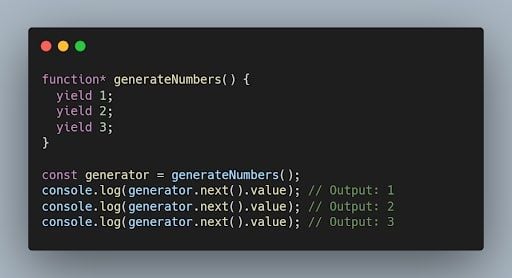
Example:

JavaScript fetch() code function example. | Image: Akshay Kumar

**44. What is a generator function in JavaScript?**

A generator function is a special type of function that can be paused and resumed during its execution. It allows generating a sequence of values over time, using the yield keyword.

Example:

JavaScript generator function code example. | Image: Akshay Kumar

**45. What are the different events in JavaScript?**

There are many different events in JavaScript, but some of the most common events include:

* **Click**: The click event occurs when a user clicks on an HTML element.
* **Mouseover**: The mouseover event occurs when a user's mouse pointer moves over an HTML element.
* **Keydown**: The keydown event occurs when a user presses a key on the keyboard.
* **Keyup**: The keyup event occurs when a user releases a key on the keyboard.
* **Change**: The change event occurs when a user changes the value of an HTML input element.

**46. What are the different ways to access an HTML element in JavaScript?**

There are three main ways to access an [HTML](https://builtin.com/software-engineering-perspectives/html) element in JavaScript:

1. **Using the getElementById() method:** The getElementById() method takes a string as an argument and returns the HTML element with the specified ID.
2. **Using the getElementsByTagName() method:** The getElementsByTagName() method takes a string as an argument and returns an array of all the HTML elements with the specified tag name.
3. **Using the querySelector() method**: The querySelector() method takes a CSS selector as an argument and returns the first HTML element that matches the selector.

**47. What is the scope of a variable in JavaScript?**

The scope of a variable in JavaScript is the part of the code where the variable can be accessed. Variables declared with the var keyword have a local scope, which means that they can only be accessed within the block of code where they are declared. Variables declared with the let keyword have a block scope, which means that they can only be accessed within the block of code where they are declared and any nested blocks. Variables declared with the const keyword have a global scope, which means that they can be accessed from anywhere in the code.

**48. What are the different ways to create objects in JavaScript?**

There are multiple ways to create objects in JavaScript, including object literals, constructor functions, the Object.create() method and the class syntax introduced in ECMAScript 2015 (ES6).

Example using object literals:

JavaScript object literals code example. | Image: Akshay Kumar

**49. What is the purpose of the windowobject in JavaScript?**

The window object represents the browser window. The window object can be used to access the browser’s features, such as the location bar, the status bar and the bookmarks bar.

**50. What is the purpose of the asyncand awaitkeywords in JavaScript?**

The async and await keywords are used for handling asynchronous operations in a more synchronous-like manner. The async keyword is used to define an asynchronous function, and the await keyword is used to pause the execution of an async function until a promise is fulfilled or rejected.

Example:



**51. What are the different data types present in javascript?**

To know the type of a JavaScript variable, we can use the **typeof**operator.

**1. Primitive types**

**String**- It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

Example :

**var** str = "Vivek Singh Bisht"; //using double quotes

**var** str2 = 'John Doe'; //using single quotes

* **Number**- It represents a number and can be written with or without decimals.

Example :

**var** x = 3; //without decimal

**var** y = 3.6; //with decimal

* **BigInt**- This data type is used to store numbers which are above the limitation of the Number data type. It can store large integers and is represented by adding “n” to an integer literal.

Example :

**var** bigInteger = 234567890123456789012345678901234567890;

* **Boolean**- It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

Example :

**var** a = 2;

**var** b = 3;

**var** c = 2;

(a == b) // returns false

(a == c) //returns true

* **Undefined**- When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

Example :

**var** x; // value of x is undefined

**var** y = undefined; // we can also set the value of a variable as undefined

* **Null**- It represents a non-existent or a invalid value.

Example :

**var** z = null;

* **Symbol**- It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

Example :

**var** symbol1 = Symbol('symbol');

* typeof **of primitive types**:

**typeof** "John Doe" // Returns "string"

**typeof** 3.14 // Returns "number"

**typeof** true // Returns "boolean"

**typeof** 234567890123456789012345678901234567890n // Returns bigint

**typeof** undefined // Returns "undefined"

**typeof** null // Returns "object" (kind of a bug in JavaScript)

**typeof** Symbol('symbol') // Returns Symbol

**2. Non-primitive types**

* Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.
* Object - Used to store collection of data.
* Example:

// Collection of data in key-value pairs

**var** obj1 = {

x: 43,

y: "Hello world!",

z: **function**(){

**return** this.x;

}

}

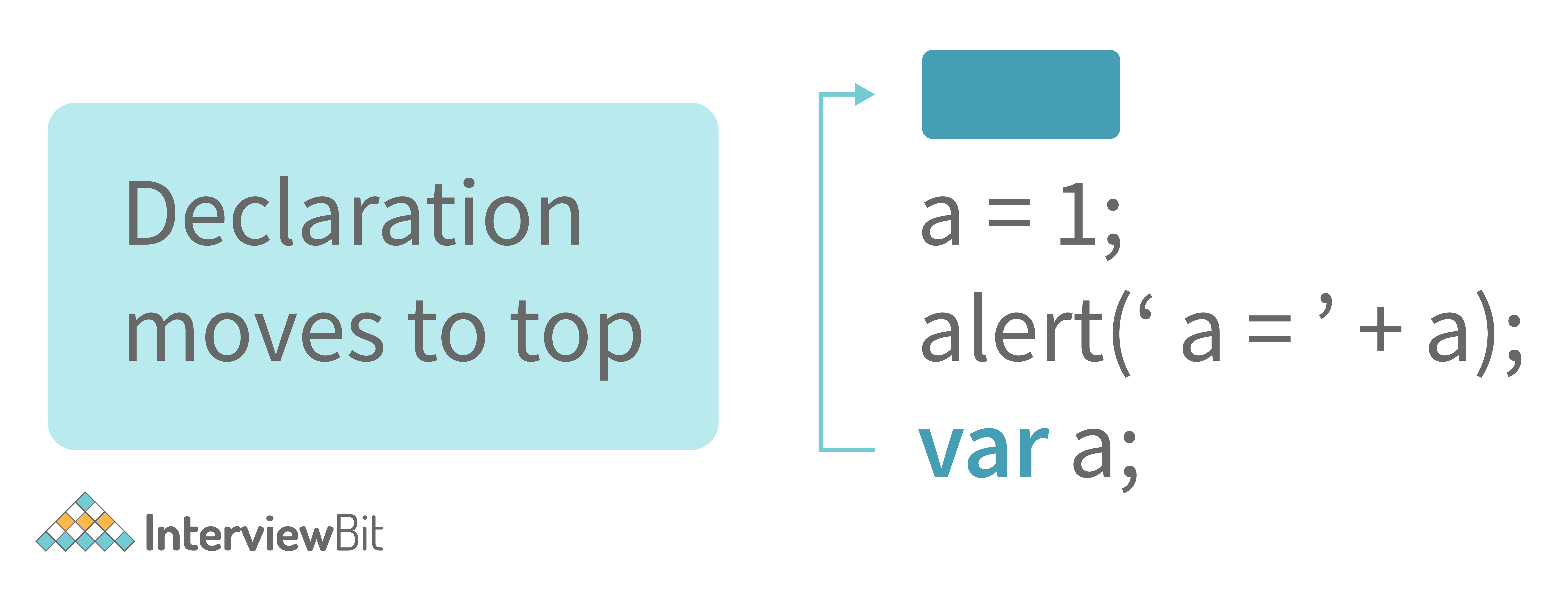
// Collection of data as an ordered list

**var** array1 = [5, "Hello", true, 4.1];

**Note- It is important to remember that any data type that is not a primitive data type, is of Object type in javascript.**

**52. Explain Hoisting in javascript.**

Hoisting is the default behaviour of javascript where all the variable and function declarations are moved on top.



This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.  
  
**Example 1:**

hoistedVariable = 3;

console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized

**var** hoistedVariable;

**Example 2:**

hoistedFunction(); // Outputs " Hello world! " even when the function is declared after calling

**function** **hoistedFunction**(){

console.log(" Hello world! ");

}

**Example 3:**

// Hoisting takes place in the local scope as well

**function** **doSomething**(){

x = 33;

console.log(x);

**var** x;

}

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

**Note - Variable initializations are not hoisted, only variable declarations are hoisted:**

**var** x;

console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted

x = 23;

**Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code:**

"use strict";

x = 23; // Gives an error since 'x' is not declared

**var** x;

**53. Why do we use the word “debugger” in javascript?**

The debugger for the browser must be activated in order to debug the code. Built-in debuggers may be switched on and off, requiring the user to report faults. The remaining section of the code should stop execution before moving on to the next line while debugging.

**54. Difference between “ == “ and “ === “ operators.**

Both are comparison operators. The difference between both the operators is that “==” is used to compare values whereas, “ === “ is used to compare both values and types.

**Example:**

**var** x = 2;

**var** y = "2";

(x == y) // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"

**55. Difference between var and let keyword in javascript.**

Some differences are

1. From the very beginning, the 'var' keyword was used in JavaScript programming **whereas the keyword**'let' was just added in 2015.
2. The keyword 'Var' has a function scope. Anywhere in the function, the variable specified using var is accessible but in ‘let’ the scope of a variable declared with the 'let' keyword is limited to the block in which it is declared. Let's start with a Block Scope.
3. In ECMAScript 2015, let and const are hoisted but not initialized. Referencing the variable in the block before the variable declaration results in a ReferenceError because the variable is in a "temporal dead zone" from the start of the block until the declaration is processed.

**Explore InterviewBit’s Exclusive Live Events**

By

**56. Explain Implicit Type Coercion in javascript.**

Implicit type coercion in javascript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

* **String coercion**

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

**var** x = 3;

**var** y = "3";

x + y // Returns "33"

Example 2:

**var** x = 24;

**var** y = "Hello";

x + y // Returns "24Hello";

Note - ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string:

**var** name = "Vivek";

**var** surname = " Bisht";

name + surname // Returns "Vivek Bisht"

Let’s understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types ( one being a number type and the other being a string type ), it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

Note - Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place.

**var** x = 3;

Var y = "3";

x - y //Returns 0 since the variable y (string type) is converted to a number type

* **Boolean Coercion**

Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.  
  
Truthy values are those which will be converted (coerced) to **true**. Falsy values are those which will be converted to **false**.  
  
All values except **false, 0, 0n, -0, “”, null, undefined, and NaN**are truthy values.

**If statements:**

Example:

**var** x = 0;

**var** y = 23;

**if**(x) { console.log(x) } // The code inside this block will not run since the value of x is 0(Falsy)

**if**(y) { console.log(y) } // The code inside this block will run since the value of y is 23 (Truthy)

* **Logical operators:**

Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.**  
  
**OR ( | | ) operator**- If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.  
  
**AND ( && ) operator**- If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.  
  
Example:

**var** x = 220;

**var** y = "Hello";

**var** z = undefined;

x | | y // Returns 220 since the first value is truthy

x | | z // Returns 220 since the first value is truthy

x && y // Returns "Hello" since both the values are truthy

y && z // Returns undefined since the second value is falsy

**if**( x && y ){

console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)

}

**if**( x || z ){

console.log("Code runs"); // This block runs because x || y returns 220(Truthy)

}

* **Equality Coercion**

Equality coercion takes place when using ‘ == ‘ operator. As we have stated before  
  
**The ‘ == ‘ operator compares values and not types.**  
  
While the above statement is a simple way to explain == operator, it’s not completely true  
  
The reality is that while using the ‘==’ operator, coercion takes place.  
  
The ‘==’ operator, converts both the operands to the same type and then compares them.  
  
Example:

**var** a = 12;

**var** b = "12";

a == b // Returns true because both 'a' and 'b' are converted to the same type and then compared. Hence the operands are equal.

Coercion does not take place when using the ‘===’ operator. Both operands are not converted to the same type in the case of ‘===’ operator.

Example:

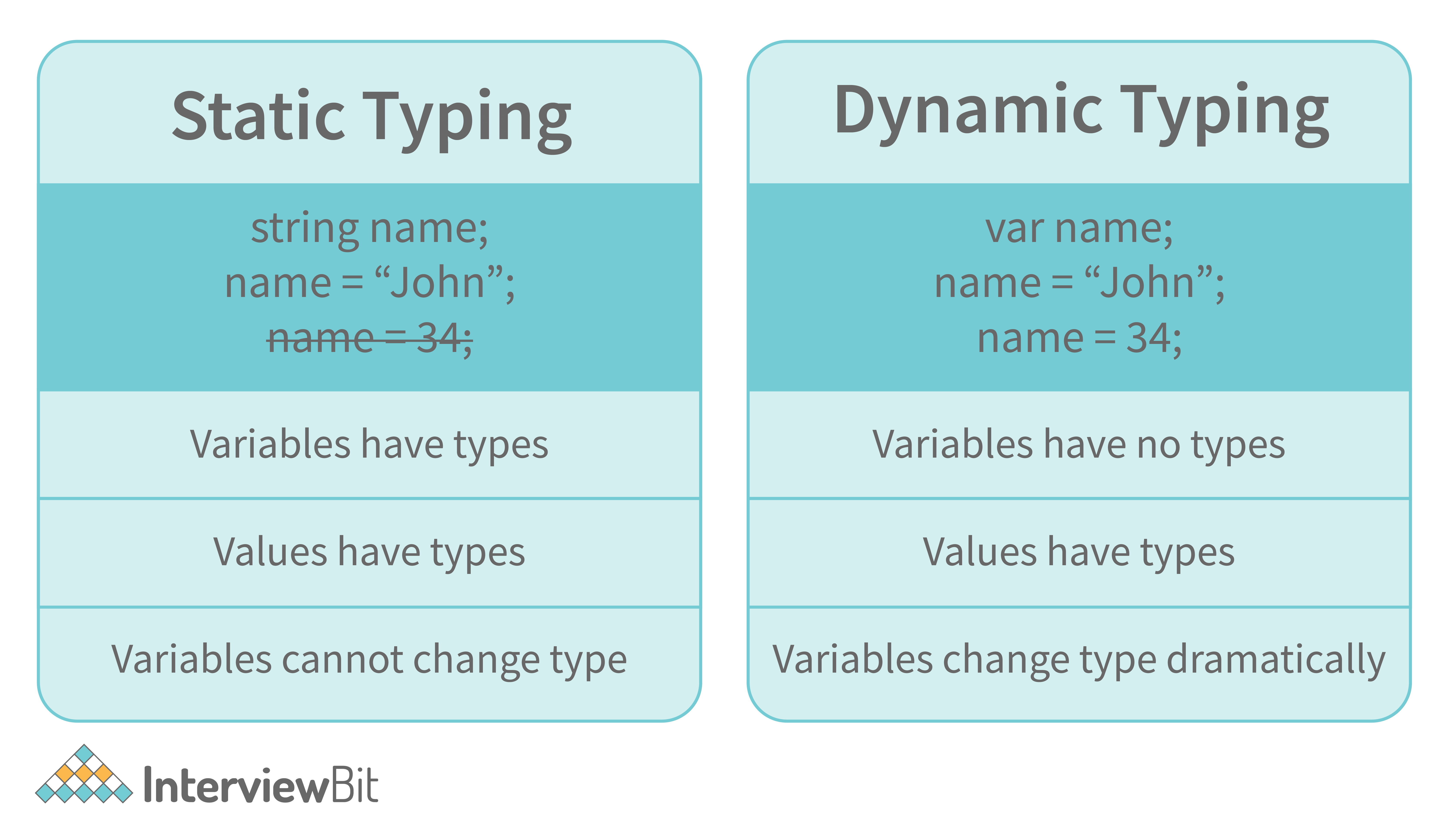
**var** a = 226;

**var** b = "226";

a === b // Returns false because coercion does not take place and the operands are of different types. Hence they are not equal.

**7. Is javascript a statically typed or a dynamically typed language?**

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time**in contrast to a statically typed language, where the type of a variable is checked during **compile-time.**



Since javascript is a loosely(dynamically) typed language, variables in JS are not associated with any type. A variable can hold the value of any data type.

For example, a variable that is assigned a number type can be converted to a string type:

**var** a = 23;

**var** a = "Hello World!";

**58. What is NaN property in JavaScript?**

NaN property represents the **“Not-a-Number”**value. It indicates a value that is not a legal number.

**typeof**of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()**function,

Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

isNaN("Hello") // Returns true

isNaN(345) // Returns false

isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a number)

isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)

isNaN(false) // Returns false

isNaN(undefined) // Returns true

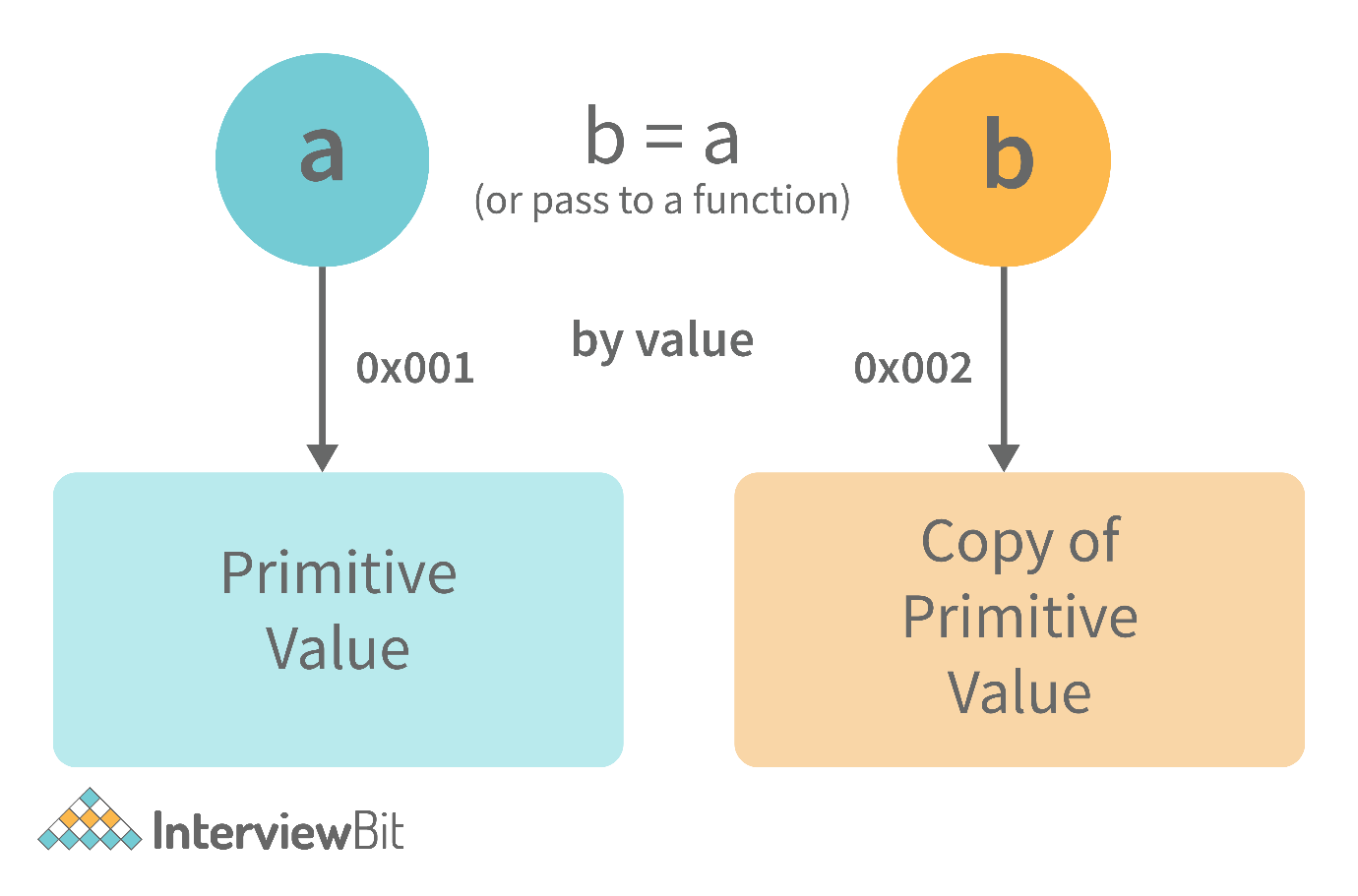
**59. Explain passed by value and passed by reference.**

**In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.**  
  
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

**var** x = 2;

In the above example, we created a variable x and assigned it a value of “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non-primitive data types,  
  
**Assign operator dealing with primitive types:**



**var** y = 234;

**var** z = y;

In the above example, the assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead, it is pointing to a new location in the memory.

**var** y = #8454; // y pointing to address of the value 234

**var** z = y;

**var** z = #5411; // z pointing to a completely new address of the value 234

// Changing the value of y

y = 23;

console.log(z); // Returns 234, since z points to a new address in the memory so changes in y will not effect z

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.  
  
**Assign operator dealing with non-primitive types:**



**var** obj = { name: "Vivek", surname: "Bisht" };

**var** obj2 = obj;

In the above example, the assign operator directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

**var** obj = #8711; // obj pointing to address of { name: "Vivek", surname: "Bisht" }

**var** obj2 = obj;

**var** obj2 = #8711; // obj2 pointing to the same address

// changing the value of obj1

obj.name = "Akki";

console.log(obj2);

// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same address.

From the above example, we can see that while passing non-primitive data types, the assigned operator directly passes the address (reference).  
  
Therefore, non-primitive data types are always **passed by reference.**

**60. What is an Immediately Invoked Function in JavaScript?**

**An Immediately Invoked Function ( known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.**

Syntax of IIFE :

(**function**(){

// Do something;

})();

To understand IIFE, we need to understand the two sets of parentheses that are added while creating an IIFE :  
  
The first set of parenthesis:

(**function** (){

//Do something;

})

While executing javascript code, whenever the compiler sees the word “function”, it assumes that we are declaring a function in the code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

**function**() {

//Do something;

}

// Compiler gives an error since the syntax of declaring a function is wrong in the code above.

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it’s a function expression.  
  
The second set of parenthesis:

(**function** (){

//Do something;

})();

From the definition of an IIFE, we know that our code should run as soon as it is defined. A function runs only when it is invoked. If we do not invoke the function, the function declaration is returned:

(**function** (){

// Do something;

})

// Returns the function declaration

**Therefore to invoke the function, we use the second set of parenthesis.**

**61. What do you mean by strict mode in javascript and characteristics of javascript strict-mode?**

In ECMAScript 5, a new feature called JavaScript Strict Mode allows you to write a code or a function in a "strict" operational environment. In most cases, this language is 'not particularly severe' when it comes to throwing errors. In 'Strict mode,' however, all forms of errors, including silent errors, will be thrown. As a result, debugging becomes a lot simpler.  Thus programmer's chances of making an error are lowered.

Characteristics of strict mode in javascript

1. Duplicate arguments are not allowed by developers.
2. In strict mode, you won't be able to use the JavaScript keyword as a parameter or function name.
3. The 'use strict' keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers.
4. Engineers will not be allowed to create global variables in 'Strict Mode.

**62. Explain Higher Order Functions in javascript.**

**Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.**  
  
Higher-order functions are a result of functions being **first-class citizens**in javascript.

Examples of higher-order functions:

**function** **higherOrder**(fn) {

fn();

}

higherOrder(**function**() { console.log("Hello world") });

**function** **higherOrder2**() {

**return** **function**() {

**return** "Do something";

}

}

**var** x = higherOrder2();

x() // Returns "Do something"

**63. Explain “this” keyword.**

**The “this” keyword refers to the object that the function is a property of.**  
  
**The value of the “this” keyword will always depend on the object that is invoking the function.\**

Confused? Let’s understand the above statements by examples:

**function** **doSomething**() {

console.log(this);

}

doSomething();

What do you think the output of the above code will be?

Note - Observe the line where we are invoking the function.

Check the definition again:

**The “this” keyword refers to the object that the function is a property of.**

In the above code, the function is a property of which object?

Since the function is invoked in the global context, **the function is a property of the global object.**

Therefore, the output of the above code will be **the global object.**Since we ran the above code inside the browser, the global object is **the window object.**

Example 2:

**var** obj = {

name: "vivek",

getName: **function**(){

console.log(this.name);

}

}

obj.getName();

In the above code, at the time of invocation, the getName function is a property of the object **obj**, therefore, **this**keyword will refer to the object **obj**, and hence the output will be “vivek”.

Example 3:

**var** obj = {

name: "vivek",

getName: **function**(){

console.log(this.name);

}

}

**var** getName = obj.getName;

**var** obj2 = {name:"akshay", getName };

obj2.getName();

Can you guess the output here?

The output will be “akshay”.

Although the getName function is declared inside the object **obj**, at the time of invocation, getName() is a property of **obj2**, therefore the “this” keyword will refer to **obj2**.

The silly way to understand the “**this”**keyword is, whenever the function is invoked, check the object before the **dot**. The value of **this**. keyword will always be the object before the **dot**.

If there is no object before the dot-like in example1, the value of this keyword will be the global object.

Example 4:

**var** obj1 = {

address : "Mumbai,India",

getAddress: **function**(){

console.log(this.address);

}

}

**var** getAddress = obj1.getAddress;

**var** obj2 = {name:"akshay"};

obj2.getAddress();

Can you guess the output?

**The output will be an error.**

Although in the code above, this keyword refers to the object **obj2**, obj2 does not have the property “address”‘, hence the getAddress function throws an error.

**64. What do you mean by Self Invoking Functions?**

Without being requested, a self-invoking expression is automatically invoked (initiated). If a function expression is followed by (), it will execute automatically. A function declaration cannot be invoked by itself.

Normally, we declare a function and call it, however, anonymous functions may be used to run a function automatically when it is described and will not be called again. And there is no name for these kinds of functions.

**65. Explain call(), apply() and, bind() methods.**

**1. call():**

* It’s a predefined method in javascript.
* This method invokes a method (function) by specifying the owner object.
* Example 1:

**function** **sayHello**(){

**return** "Hello " + this.name;

}

**var** obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"

* call() method allows an object to use the method (function) of another object.
* Example 2:

**var** person = {

age: 23,

getAge: **function**(){

**return** this.age;

}

}

**var** person2 = {age: 54};

person.getAge.call(person2);

// Returns 54

* call() accepts arguments:

**function** **saySomething**(message){

**return** this.name + " is " + message;

}

**var** person4 = {name: "John"};

saySomething.call(person4, "awesome");

// Returns "John is awesome"

**apply()**  
  
The apply method is similar to the call() method. The only difference is that,  
  
**call() method takes arguments separately whereas, apply() method takes arguments as an array.**

**function** **saySomething**(message){

**return** this.name + " is " + message;

}

**var** person4 = {name: "John"};

saySomething.apply(person4, ["awesome"]);

**2. bind():**

* This method returns a new function, where the value of **“this”**keyword will be bound to the owner object, which is provided as a parameter.
* Example with arguments:

**var** bikeDetails = {

displayDetails: **function**(registrationNumber,brandName){

**return** this.name+ " , "+ "bike details: "+ registrationNumber + " , " + brandName;

}

}

**var** person1 = {name: "Vivek"};

**var** detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object

detailsOfPerson1();

//Returns Vivek, bike details: TS0122, Bullet

**66. What is the difference between exec () and test () methods in javascript?**

* **test ()** and **exec ()** are RegExp expression methods used in javascript.
* We'll use **exec ()** to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
* We will use a**test ()** to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'.

**67. What is currying in JavaScript?**

**Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.**

Example of a curried function:

**function** **add** (a) {

**return** **function**(b){

**return** a + b;

}

}

add(3)(4)

For Example, if we have a function **f(a,b)**, then the function after currying, will be transformed to **f(a)(b).**  
  
By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.  
  
Let’s see currying in action:

**function** **multiply**(a,b){

**return** a\*b;

}

**function** **currying**(fn){

**return** **function**(a){

**return** **function**(b){

**return** fn(a,b);

}

}

}

**var** curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12

As one can see in the code above, we have transformed the function **multiply(a,b)**to a function **curriedMultiply**, which takes in one parameter at a time.

**68. What are some advantages of using External JavaScript?**

External JavaScript is the JavaScript Code (script) written in a separate file with the extension.js, and then we link that file inside the <head> or <body> element of the HTML file where the code is to be placed.

Some advantages of external javascript are

1. It allows web designers and developers to collaborate on HTML and javascript files.
2. We can reuse the code.
3. Code readability is simple in external javascript.

**69. Explain Scope and Scope Chain in javascript.**

Scope in JS determines the accessibility of variables and functions at various parts of one’s code.  
  
In general terms, the scope will let us know at a given part of code, what are variables and functions we can or cannot access.  
  
There are three types of scopes in JS:

* Global Scope
* Local or Function Scope
* Block Scope

**Global Scope:**Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

**var** globalVariable = "Hello world";

**function** **sendMessage**(){

**return** globalVariable; // can access globalVariable since it's written in global space

}

**function** **sendMessage2**(){

**return** sendMessage(); // Can access sendMessage function since it's written in global space

}

sendMessage2(); // Returns “Hello world”

**Function Scope:**Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

**function** **awesomeFunction**(){

**var** a = 2;

**var** multiplyBy2 = **function**(){

console.log(a\*2); // Can access variable "a" since a and multiplyBy2 both are written inside the same function

}

}

console.log(a); // Throws reference error since a is written in local scope and cannot be accessed outside

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope

**Block Scope:**Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope. Block scope tells us that any variable declared inside a block { }, can be accessed only inside that block and cannot be accessed outside of it.

{

**let** x = 45;

}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

**for**(**let** i=0; i<2; i++){

// do something

}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop block

**Scope Chain:**JavaScript engine also uses Scope to find variables. Let’s understand that using an example:

**var** y = 24;

**function** **favFunction**(){

**var** x = 667;

**var** anotherFavFunction = **function**(){

console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, outputs 667

}

**var** yetAnotherFavFunction = **function**(){

console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction and does not find it, so looks for variable in global scope, finds it and outputs 24

}

anotherFavFunction();

yetAnotherFavFunction();

}

favFunction();

**As you can see in the code above, if the javascript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.**

If the variable is not found in the global space as well, a reference error is thrown.

**70. Explain Closures in JavaScript.**

Closures are an ability of a function to remember the variables and functions that are declared in its outer scope.

**var** Person = **function**(pName){

**var** name = pName;

this.getName = **function**(){

**return** name;

}

}

**var** person = **new** Person("Neelesh");

console.log(person.getName());

Let’s understand closures by example:

**function** **randomFunc**(){

**var** obj1 = {name:"Vivian", age:45};

**return** **function**(){

console.log(obj1.name + " is "+ "awesome"); // Has access to obj1 even when the randomFunc function is executed

}

}

**var** initialiseClosure = randomFunc(); // Returns a function

initialiseClosure();

Let’s understand the code above,  
  
The function randomFunc() gets executed and returns a function when we assign it to a variable:

**var** initialiseClosure = randomFunc();

-The returned function is then executed when we invoke initialiseClosure:

initialiseClosure();

The line of code above outputs “Vivian is awesome” and this is possible because of closure.

console.log(obj1.name + " is "+ "awesome");

-When the function randomFunc() runs, it seems that the returning function is using the variable obj1 inside it:

-Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed.  
  
**This ability of a function to store a variable for further reference even after it is executed is called Closure.**

**71. Mention some advantages of javascript.**

There are many advantages of javascript. Some of them are

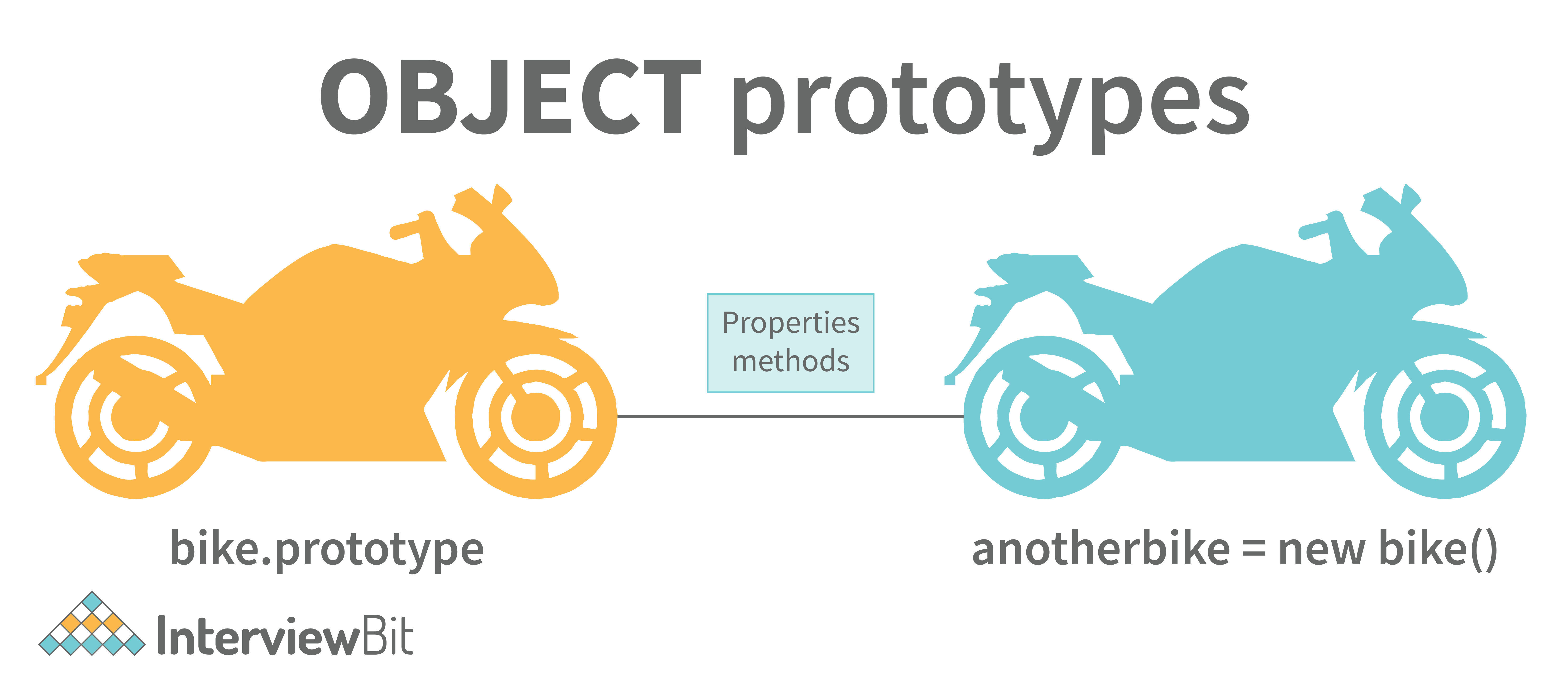
1. Javascript is executed on the client-side as well as server-side also. There are a variety of Frontend Frameworks that you may study and utilize. However, if you want to use JavaScript on the backend, you'll need to learn NodeJS. It is currently the only JavaScript framework that may be used on the backend.
2. Javascript is a simple language to learn.
3. Web pages now have more functionality because of Javascript.
4. To the end-user, Javascript is quite quick.

**72. What are object prototypes?**

All javascript objects inherit properties from a prototype. For example,

* Date objects inherit properties from the Date prototype
* Math objects inherit properties from the Math prototype
* Array objects inherit properties from the Array prototype.
* On top of the chain is **Object.prototype.**Every prototype inherits properties and methods from the Object.prototype.
* **A prototype is a blueprint of an object. The prototype** allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.

Let’s see prototypes help us use methods and properties:



**var** arr = [];

arr.push(2);

console.log(arr); // Outputs [2]

In the code above, as one can see, we have not defined any property or method called push on the array “arr” but the javascript engine does not throw an error.

The reason is the use of prototypes. As we discussed before, Array objects inherit properties from the Array prototype.

The javascript engine sees that the method push does not exist on the current array object and therefore, looks for the method push inside the Array prototype and it finds the method.

Whenever the property or method is not found on the current object, the javascript engine will always try to look in its prototype and if it still does not exist, it looks inside the prototype's prototype and so on.

**73. What are callbacks?**

A callback is a function that will be executed after another function gets executed. In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function, and can be used as a property of an object.

**Functions that are used as an argument to another function are called callback functions.**Example:

**function** **divideByHalf**(sum){

console.log(Math.floor(sum / 2));

}

**function** **multiplyBy2**(sum){

console.log(sum \* 2);

}

**function** **operationOnSum**(num1,num2,operation){

**var** sum = num1 + num2;

operation(sum);

}

operationOnSum(3, 3, divideByHalf); // Outputs 3

operationOnSum(5, 5, multiplyBy2); // Outputs 20

* In the code above, we are performing mathematical operations on the sum of two numbers. The operationOnSum function takes 3 arguments, the first number, the second number, and the operation that is to be performed on their sum (callback).
* Both divideByHalf and multiplyBy2 functions are used as callback functions in the code above.
* These callback functions will be executed only after the function operationOnSum is executed.
* Therefore, a callback is a function that will be executed after another function gets executed.

**74. What are the types of errors in javascript?**

There are two types of errors in javascript.

1. **Syntax error**: Syntax errors are mistakes or spelling problems in the code that cause the program to not execute at all or to stop running halfway through. Error messages are usually supplied as well.
2. **Logical error**: Reasoning mistakes occur when the syntax is proper but the logic or program is incorrect. The application executes without problems in this case. However, the output findings are inaccurate. These are sometimes more difficult to correct than syntax issues since these applications do not display error signals for logic faults.

**75. What is memoization?**

Memoization is a form of caching where the return value of a function is cached based on its parameters. If the parameter of that function is not changed, the cached version of the function is returned.  
Let’s understand memoization, by converting a simple function to a memoized function:

Note- Memoization is used for expensive function calls but in the following example, we are considering a simple function for understanding the concept of memoization better.

Consider the following function:

**function** **addTo256**(num){

**return** num + 256;

}

addTo256(20); // Returns 276

addTo256(40); // Returns 296

addTo256(20); // Returns 276

In the code above, we have written a function that adds the parameter to 256 and returns it.  
  
When we are calling the function addTo256 again with the same parameter (“20” in the case above), we are computing the result again for the same parameter.  
  
Computing the result with the same parameter, again and again, is not a big deal in the above case, but imagine if the function does some heavy-duty work, then, computing the result again and again with the same parameter will lead to wastage of time.

This is where memoization comes in, by using memoization we can store(cache) the computed results based on the parameters. If the same parameter is used again while invoking the function, instead of computing the result, we directly return the stored (cached) value.

Let’s convert the above function addTo256, to a memoized function:

**function** **memoizedAddTo256**(){

**var** cache = {};

**return** **function**(num){

**if**(num **in** cache){

console.log("cached value");

**return** cache[num]

}

**else**{

cache[num] = num + 256;

**return** cache[num];

}

}

}

**var** memoizedFunc = memoizedAddTo256();

memoizedFunc(20); // Normal return

memoizedFunc(20); // Cached return

In the code above, if we run the memoizedFunc function with the same parameter, instead of computing the result again, it returns the cached result.

Note- Although using memoization saves time, it results in larger consumption of memory since we are storing all the computed results.

**76. What is recursion in a programming language?**

Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it arrives at a result.

function add(number) {

if (number <= 0) {

return 0;

} else {

return number + add(number - 1);

}

}

add(3) => 3 + add(2)

3 + 2 + add(1)

3 + 2 + 1 + add(0)

3 + 2 + 1 + 0 = 6

Example of a recursive function:  
  
The following function calculates the sum of all the elements in an array by using recursion:

function computeSum(arr){

if(arr.length === 1){

return arr[0];

}

else{

return arr.pop() + computeSum(arr);

}

}

computeSum([7, 8, 9, 99]); // Returns 123

**77. What is the use of a constructor function in javascript?**

Constructor functions are used to create objects in javascript.

When do we use constructor functions?

If we want to create multiple objects having similar properties and methods, constructor functions are used.

**Note- The name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.**

Example:

**function** **Person**(name,age,gender){

this.name = name;

this.age = age;

this.gender = gender;

}

**var** person1 = **new** Person("Vivek", 76, "male");

console.log(person1);

**var** person2 = **new** Person("Courtney", 34, "female");

console.log(person2);

In the code above, we have created a constructor function named Person. Whenever we want to create a new object of the type Person, We need to create it using the new keyword:

**var** person3 = **new** Person("Lilly", 17, "female");

The above line of code will create a new object of the type Person. Constructor functions allow us to group similar objects.

**78. What is DOM?**

* DOM stands for Document Object Model.  DOM is a programming interface for HTML and XML documents.
* When the browser tries to render an HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document.
* Example of how HTML code gets converted to DOM:



**79. Which method is used to retrieve a character from a certain index?**

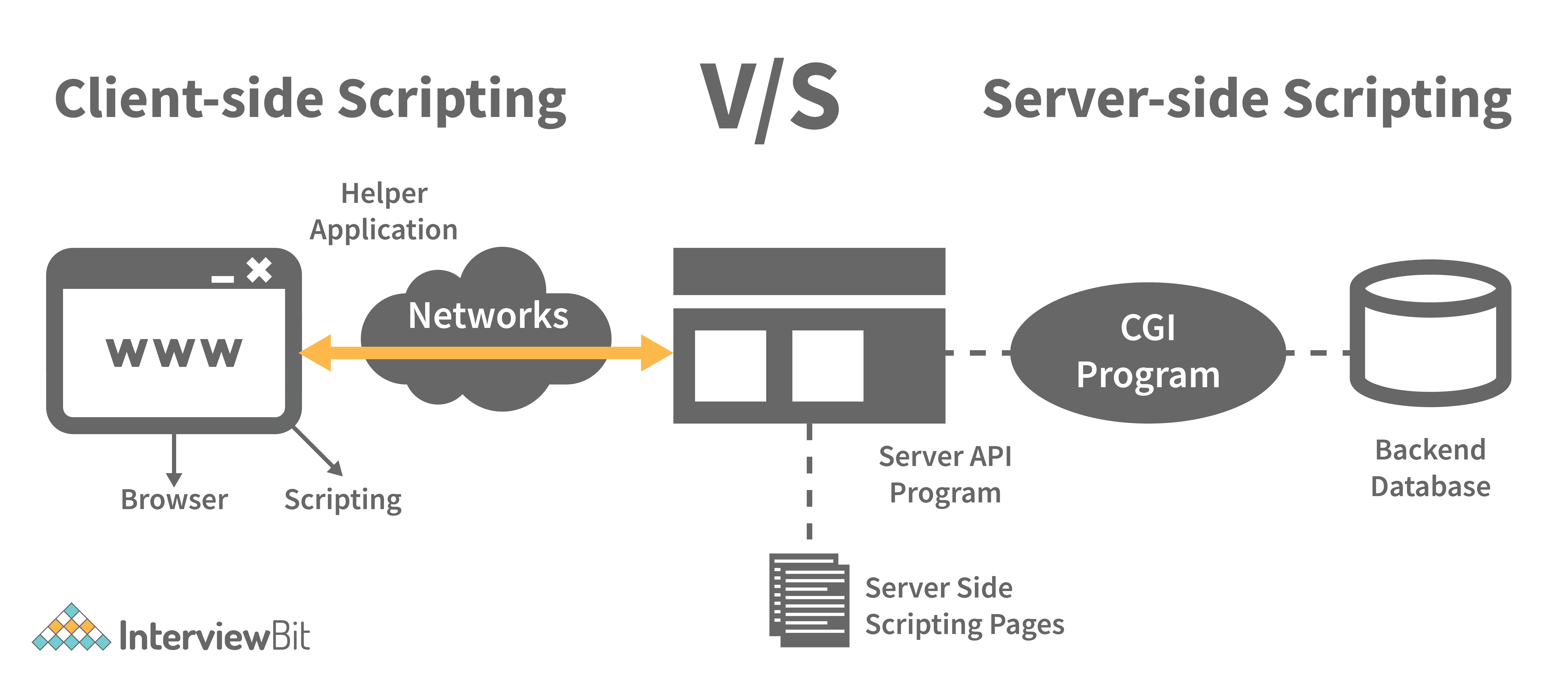
The charAt() function of the JavaScript string finds a char element at the supplied index. The index number begins at 0 and continues up to n-1, Here n is the string length. The index value must be positive, higher than, or the same as the string length.

**80. What do you mean by BOM?**

Browser Object Model is known as BOM. It allows users to interact with the browser. A browser's initial object is a window. As a result, you may call all of the window's functions directly or by referencing the window. The document, history, screen, navigator, location, and other attributes are available in the window object.

**81. What is the distinction between client-side and server-side JavaScript?**

Client-side JavaScript is made up of two parts, a fundamental language and predefined objects for performing JavaScript in a browser. JavaScript for the client is automatically included in the HTML pages. At runtime, the browser understands this script.



Server-side JavaScript, involves the execution of JavaScript code on a server in response to client requests. It handles these requests and delivers the relevant response to the client, which may include client-side JavaScript for subsequent execution within the browser

**82. What are arrow functions?**

Arrow functions were introduced in the ES6 version of javascript. They provide us with a new and shorter syntax for declaring functions. Arrow functions can only be used as a function expression.  
  
Let’s compare the normal function declaration and the arrow function declaration in detail:

// Traditional Function Expression

**var** add = **function**(a,b){

**return** a + b;

}

// Arrow Function Expression

**var** arrowAdd = (a,b) => a + b;

Arrow functions are declared without the function keyword. If there is only one returning expression then we don’t need to use the return keyword as well in an arrow function as shown in the example above. Also, for functions having just one line of code, curly braces { } can be omitted.

// Traditional function expression

**var** multiplyBy2 = **function**(num){

**return** num \* 2;

}

// Arrow function expression

**var** arrowMultiplyBy2 = num => num \* 2;

If the function takes in only one argument, then the parenthesis () around the parameter can be omitted as shown in the code above.

**var** obj1 = {

valueOfThis: **function**(){

**return** this;

}

}

**var** obj2 = {

valueOfThis: ()=>{

**return** this;

}

}

obj1.valueOfThis(); // Will return the object obj1

obj2.valueOfThis(); // Will return window/global object

The biggest difference between the traditional function expression and the arrow function is the handling of **this**keyword. By general definition, **this**keyword always refers to the object that is calling the function. As you can see in the code above, **obj1.valueOfThis()**returns obj1 since **this**keyword refers to the object calling the function.

In the arrow functions, there is no binding of **this**keyword. Thiskeyword inside an arrow function does not refer to the object calling it. It rather inherits its value from the parent scope which is the window object in this case. Therefore, in the code above, **obj2.valueOfThis()**returns the window object.

**83. What do mean by prototype design pattern?**

The Prototype Pattern produces different objects, but instead of returning uninitialized objects, it produces objects that have values replicated from a template – or sample – object. Also known as the Properties pattern, the Prototype pattern is used to create prototypes.

The introduction of business objects with parameters that match the database's default settings is a good example of where the Prototype pattern comes in handy. The default settings for a newly generated business object are stored in the prototype object.

The Prototype pattern is hardly used in traditional languages, however, it is used in the development of new objects and templates in JavaScript, which is a prototypal language.

**84. Differences between declaring variables using var, let and const.**

Before the ES6 version of javascript, only the keyword var was used to declare variables. With the ES6 Version, keywords let and const were introduced to declare variables.

|  |  |  |  |
| --- | --- | --- | --- |
| keyword | const | let | var |
| global scope | no | no | yes |
| function scope | yes | yes | yes |
| block scope | yes | yes | no |
| can be reassigned | no | yes | yes |

**Let’s understand the differences with examples:**

**var** variable1 = 23;

**let** variable2 = 89;

**function** **catchValues**(){

console.log(variable1);

console.log(variable2);

// Both the variables can be accessed anywhere since they are declared in the global scope

}

window.variable1; // Returns the value 23

window.variable2; // Returns undefined

* The variables declared with the let keyword in the global scope behave just like the variable declared with the var keyword in the global scope.
* Variables declared in the global scope with var and let keywords can be accessed from anywhere in the code.
* But, there is one difference! Variables that are declared with the var keyword in the global scope are added to the window/global object. Therefore, they can be accessed using window.variableName.  
  Whereas, the variables declared with the let keyword are not added to the global object, therefore, trying to access such variables using window.variableName results in an error.

**var vs let in functional scope**

**function** **varVsLetFunction**(){

**let** awesomeCar1 = "Audi";

**var** awesomeCar2 = "Mercedes";

}

console.log(awesomeCar1); // Throws an error

console.log(awesomeCar2); // Throws an error

Variables are declared in a functional/local scope using **var**and **let**keywords behave exactly the same, meaning, they cannot be accessed from outside of the scope.

{

**var** variable3 = [1, 2, 3, 4];

}

console.log(variable3); // Outputs [1,2,3,4]

{

**let** variable4 = [6, 55, -1, 2];

}

console.log(variable4); // Throws error

**for**(**let** i = 0; i < 2; i++){

//Do something

}

console.log(i); // Throws error

**for**(**var** j = 0; j < 2; i++){

// Do something

}

console.log(j) // Outputs 2

* In javascript, a block means the code written inside the curly braces **{}**.
* Variables declared with **var**keyword do not have block scope. It means a variable declared in block scope **{}**with the **var**keyword is the same as declaring the variable in the global scope.
* Variables declared with **let**keyword inside the block scope cannot be accessed from outside of the block.

**Const keyword**

* Variables with the **const**keyword behave exactly like a variable declared with the let keyword with only one difference, **any variable declared with the const keyword cannot be reassigned.**
* Example:

**const** x = {name:"Vivek"};

x = {address: "India"}; // Throws an error

x.name = "Nikhil"; // No error is thrown

**const** y = 23;

y = 44; // Throws an error

In the code above, although we can change the value of a property inside the variable declared with **const**keyword, we cannot completely reassign the variable itself.

**85. What is the rest parameter and spread operator?**

Both rest parameter and spread operator were introduced in the ES6 version of javascript.  
  
**Rest parameter ( … ):**

* It provides an improved way of handling the parameters of a function.
* Using the rest parameter syntax, we can create functions that can take a variable number of arguments.
* Any number of arguments will be converted into an array using the rest parameter.
* It also helps in extracting all or some parts of the arguments.
* Rest parameters can be used by applying three dots (...) before the parameters.

**function** **extractingArgs**(...args){

**return** args[1];

}

// extractingArgs(8,9,1); // Returns 9

**function** **addAllArgs**(...args){

**let** sumOfArgs = 0;

**let** i = 0;

**while**(i < args.length){

sumOfArgs += args[i];

i++;

}

**return** sumOfArgs;

}

addAllArgs(6, 5, 7, 99); // Returns 117

addAllArgs(1, 3, 4); // Returns 8

**\*\*Note- Rest parameter should always be used at the last parameter of a function:**

// Incorrect way to use rest parameter

**function** **randomFunc**(a,...args,c){

//Do something

}

// Correct way to use rest parameter

**function** **randomFunc2**(a,b,...args){

//Do something

}

* **Spread operator (…):**Although the syntax of the spread operator is exactly the same as the rest parameter, the spread operator is used to spreading an array, and object literals. We also use spread operators where one or more arguments are expected in a function call.

**function** **addFourNumbers**(num1,num2,num3,num4){

**return** num1 + num2 + num3 + num4;

}

**let** fourNumbers = [5, 6, 7, 8];

addFourNumbers(...fourNumbers);

// Spreads [5,6,7,8] as 5,6,7,8

**let** array1 = [3, 4, 5, 6];

**let** clonedArray1 = [...array1];

// Spreads the array into 3,4,5,6

console.log(clonedArray1); // Outputs [3,4,5,6]

**let** obj1 = {x:'Hello', y:'Bye'};

**let** clonedObj1 = {...obj1}; // Spreads and clones obj1

console.log(obj1);

**let** obj2 = {z:'Yes', a:'No'};

**let** mergedObj = {...obj1, ...obj2}; // Spreads both the objects and merges it

console.log(mergedObj);

// Outputs {x:'Hello', y:'Bye',z:'Yes',a:'No'};

\*\*\*Note- Key differences between rest parameter and spread operator:

* Rest parameter is used to take a variable number of arguments and turns them into an array while the spread operator takes an array or an object and spreads it
* Rest parameter is used in function declaration whereas the spread operator is used in function calls.

**86. In JavaScript, how many different methods can you make an object?**

In JavaScript, there are several ways to declare or construct an object.

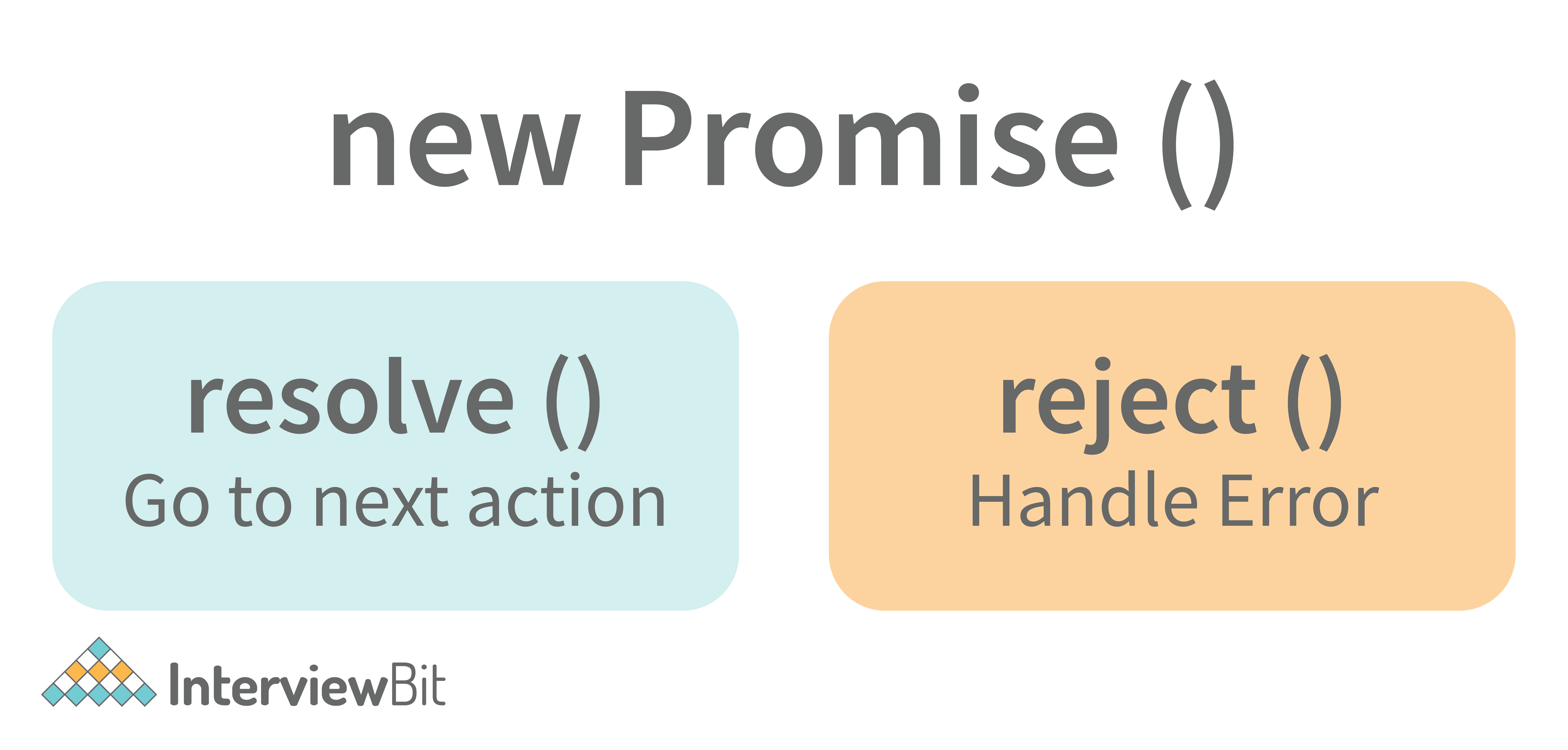
1. Object.
2. using Class.
3. create Method.
4. Object Literals.
5. using Function.
6. Object Constructor.

**87. What is the use of promises in javascript?**

**Promises are used to handle asynchronous operations in javascript.**  
  
Before promises, callbacks were used to handle asynchronous operations. But due to the limited functionality of callbacks, using multiple callbacks to handle asynchronous code can lead to unmanageable code.  
  
Promise object has four states -

* Pending - Initial state of promise. This state represents that the promise has neither been fulfilled nor been rejected, it is in the pending state.
* Fulfilled - This state represents that the promise has been fulfilled, meaning the async operation is completed.
* Rejected - This state represents that the promise has been rejected for some reason, meaning the async operation has failed.
* Settled - This state represents that the promise has been either rejected or fulfilled.

A promise is created using the **Promise**constructor which takes in a callback function with two parameters, **resolve**and **reject**respectively.



**resolve**is a function that will be called when the async operation has been successfully completed.  
  
**reject**is a function that will be called, when the async operation fails or if some error occurs.  
  
Example of a promise:  
  
**Promises are used to handle asynchronous operations like server requests, for ease of understanding, we are using an operation to calculate the sum of three elements.**  
  
In the function below, we are returning a promise inside a function:

**function** **sumOfThreeElements**(...elements){

**return** **new** Promise((resolve,reject)=>{

**if**(elements.length > 3 ){

reject("Only three elements or less are allowed");

}

**else**{

**let** sum = 0;

**let** i = 0;

**while**(i < elements.length){

sum += elements[i];

i++;

}

resolve("Sum has been calculated: "+sum);

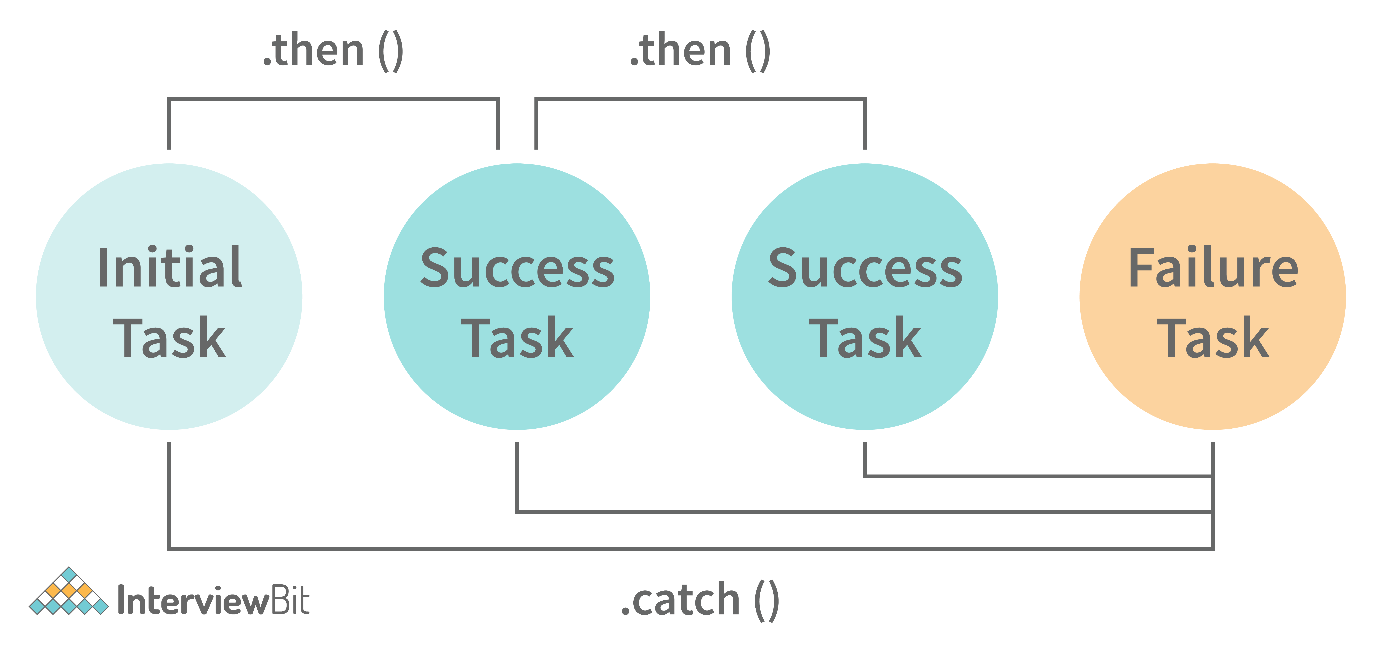
}

})

}

In the code above, we are calculating the sum of three elements, if the length of the elements array is more than 3, a promise is rejected, or else the promise is resolved and the sum is returned.

We can consume any promise by attaching then() and catch() methods to the consumer.



**then()**method is used to access the result when the promise is fulfilled.

**catch()**method is used to access the result/error when the promise is rejected. In the code below, we are consuming the promise:

sumOfThreeElements(4, 5, 6)

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

.catch(error=> console.log(error));

// In the code above, the promise is fulfilled so the then() method gets executed

sumOfThreeElements(7, 0, 33, 41)

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

.catch(error=> console.log(error));

// In the code above, the promise is rejected hence the catch() method gets executed

**88. What are classes in javascript?**

Introduced in the ES6 version, classes are nothing but syntactic sugars for constructor functions. They provide a new way of declaring constructor functions in javascript.  Below are the examples of how classes are declared and used:

// Before ES6 version, using constructor functions

**function** **Student**(name,rollNumber,grade,section){

this.name = name;

this.rollNumber = rollNumber;

this.grade = grade;

this.section = section;

}

// Way to add methods to a constructor function

Student.prototype.getDetails = **function**(){

**return** 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade: ${this.grade}, Section:${this.section}';

}

**let** student1 = **new** Student("Vivek", 354, "6th", "A");

student1.getDetails();

// Returns Name: Vivek, Roll no:354, Grade: 6th, Section:A

// ES6 version classes

**class** **Student**{

**constructor**(name,rollNumber,grade,section){

this.name = name;

this.rollNumber = rollNumber;

this.grade = grade;

this.section = section;

}

// Methods can be directly added inside the class

**getDetails**(){

**return** 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade:${this.grade}, Section:${this.section}';

}

}

**let** student2 = **new** Student("Garry", 673, "7th", "C");

student2.getDetails();

// Returns Name: Garry, Roll no:673, Grade: 7th, Section:C

Key points to remember about classes:

* Unlike functions, classes are not hoisted. A class cannot be used before it is declared.
* A class can inherit properties and methods from other classes by using the extend keyword.
* All the syntaxes inside the class must follow the strict mode(‘use strict’) of javascript. An error will be thrown if the strict mode rules are not followed.

**89. What are generator functions?**

Introduced in the ES6 version, generator functions are a special class of functions.  
  
**They can be stopped midway and then continue from where they had stopped.**  
  
Generator functions are declared with the **function\***keyword instead of the normal **function**keyword:

**function**\* **genFunc**(){

// Perform operation

}

In normal functions, we use the **return**keyword to return a value and as soon as the return statement gets executed, the function execution stops:

**function** **normalFunc**(){

**return** 22;

console.log(2); // This line of code does not get executed

}

In the case of generator functions, when called, they do not execute the code, instead, they return a **generator object**. This generator object handles the execution.

**function**\* **genFunc**(){

**yield** 3;

**yield** 4;

}

genFunc(); // Returns Object [Generator] {}

The generator object consists of a method called **next()**, this method when called, executes the code until the nearest **yield**statement, and returns the yield value.  
  
For example, if we run the next() method on the above code:

genFunc().next(); // Returns {value: 3, done:false}

As one can see the next method returns an object consisting of a **value**and **done**properties.  Value property represents the yielded value. Done property tells us whether the function code is finished or not. (Returns true if finished).

Generator functions are used to return iterators. Let’s see an example where an iterator is returned:

**function**\* **iteratorFunc**() {

**let** count = 0;

**for** (**let** i = 0; i < 2; i++) {

count++;

**yield** i;

}

**return** count;

}

**let** iterator = iteratorFunc();

console.log(iterator.next()); // {value:0,done:false}

console.log(iterator.next()); // {value:1,done:false}

console.log(iterator.next()); // {value:2,done:true}

As you can see in the code above, the last line returns **done:true**, since the code reaches the return statement.

**90. Explain WeakSet in javascript.**

In javascript, a Set is a collection of unique and ordered elements. Just like Set, WeakSet is also a collection of unique and ordered elements with some key differences:

* Weakset contains only objects and no other type.
* An object inside the weakset is referenced weakly. This means, that if the object inside the weakset does not have a reference, it will be garbage collected.
* Unlike Set, WeakSet only has three methods, **add()**, **delete()**and **has()**.

**const** newSet = **new** Set([4, 5, 6, 7]);

console.log(newSet);// Outputs Set {4,5,6,7}

**const** newSet2 = **new** WeakSet([3, 4, 5]); //Throws an error

**let** obj1 = {message:"Hello world"};

**const** newSet3 = **new** WeakSet([obj1]);

console.log(newSet3.has(obj1)); // true

**91. Why do we use callbacks?**

A callback function is a method that is sent as an input to another function (now let us name this other function "thisFunction"), and it is performed inside the thisFunction after the function has completed execution.

JavaScript is a scripting language that is based on events. Instead of waiting for a reply before continuing, JavaScript will continue to run while monitoring for additional events. Callbacks are a technique of ensuring that a particular code does not run until another code has completed its execution.

**92. Explain WeakMap in javascript.**

In javascript, Map is used to store key-value pairs. The key-value pairs can be of both primitive and non-primitive types. WeakMap is similar to Map with key differences:

* The keys and values in weakmap should always be an object.
* If there are no references to the object, the object will be garbage collected.

**const** map1 = **new** Map();

map1.set('Value', 1);

**const** map2 = **new** WeakMap();

map2.set('Value', 2.3); // Throws an error

**let** obj = {name:"Vivek"};

**const** map3 = **new** WeakMap();

map3.set(obj, {age:23});

**93. What is Object Destructuring?**

Object destructuring is a new way to extract elements from an object or an array.

* **Object destructuring:**Before ES6 version:

**const** classDetails = {

strength: 78,

benches: 39,

blackBoard:1

}

**const** classStrength = classDetails.strength;

**const** classBenches = classDetails.benches;

**const** classBlackBoard = classDetails.blackBoard;

The same example using object destructuring:

**const** classDetails = {

strength: 78,

benches: 39,

blackBoard:1

}

**const** {strength:classStrength, benches:classBenches,blackBoard:classBlackBoard} = classDetails;

console.log(classStrength); // Outputs 78

console.log(classBenches); // Outputs 39

console.log(classBlackBoard); // Outputs 1

As one can see, using object destructuring we have extracted all the elements inside an object in one line of code. If we want our new variable to have the same name as the property of an object we can remove the colon:

**const** {strength:strength} = classDetails;

// The above line of code can be written as:

**const** {strength} = classDetails;

* **Array destructuring:**Before ES6 version:

**const** arr = [1, 2, 3, 4];

**const** first = arr[0];

**const** second = arr[1];

**const** third = arr[2];

**const** fourth = arr[3];

The same example using object destructuring:

**const** arr = [1, 2, 3, 4];

**const** [first,second,third,fourth] = arr;

console.log(first); // Outputs 1

console.log(second); // Outputs 2

console.log(third); // Outputs 3

console.log(fourth); // Outputs 4

**94. Difference between prototypal and classical inheritance**

Programers build objects, which are representations of real-time entities, in traditional OO programming. Classes and objects are the two sorts of abstractions. A class is a generalization of an object, whereas an object is an abstraction of an actual thing. A Vehicle, for example, is a specialization of a Car. As a result, automobiles (class) are descended from vehicles (object).

Classical inheritance differs from prototypal inheritance in that classical inheritance is confined to classes that inherit from those remaining classes, but prototypal inheritance allows any object to be cloned via an object linking method. Despite going into too many specifics, a prototype essentially serves as a template for those other objects, whether they extend the parent object or not.

**95. What is a Temporal Dead Zone?**

Temporal Dead Zone is a behaviour that occurs with variables declared using **let**and **const**keywords. It is a behaviour where we try to access a variable before it is initialized. Examples of temporal dead zone:

x = 23; // Gives reference error

**let** x;

**function** **anotherRandomFunc**(){

message = "Hello"; // Throws a reference error

**let** message;

}

anotherRandomFunc();

In the code above, both in the global scope and functional scope, we are trying to access variables that have not been declared yet. This is called the **Temporal Dead Zone**.

**96. What do you mean by JavaScript Design Patterns?**

JavaScript design patterns are repeatable approaches for errors that arise sometimes when building JavaScript browser applications. They truly assist us in making our code more stable.

They are divided mainly into 3 categories

1. Creational Design Pattern
2. Structural Design Pattern
3. Behavioral Design Pattern.

* **Creational Design Pattern:**The object generation mechanism is addressed by the JavaScript Creational Design Pattern. They aim to make items that are appropriate for a certain scenario.
* **Structural Design Pattern:**The JavaScript Structural Design Pattern explains how the classes and objects we've generated so far can be combined to construct bigger frameworks. This pattern makes it easier to create relationships between items by defining a straightforward way to do so.
* **Behavioral Design Pattern:**This design pattern highlights typical patterns of communication between objects in JavaScript. As a result, the communication may be carried out with greater freedom.

**97. Is JavaScript a pass-by-reference or pass-by-value language?**

The variable's data is always a reference for objects, hence it's always pass by value. As a result, if you supply an object and alter its members inside the method, the changes continue outside of it. It appears to be pass by reference in this case. However, if you modify the values of the object variable, the change will not last, demonstrating that it is indeed passed by value.

**98. Difference between Async/Await and Generators usage to achieve the same functionality.**

* Generator functions are run by their generator yield by yield which means one output at a time, whereas Async-await functions are executed sequentially one after another.
* Async/await provides a certain use case for Generators easier to execute.
* The output result of the Generator function is always value: X, done: Boolean, but the return value of the Async function is always an assurance or throws an error.

**99. What are the primitive data types in JavaScript?**

A primitive is a data type that isn't composed of other data types. It's only capable of displaying one value at a time. By definition, every primitive is a built-in data type (the compiler must be knowledgeable of them) nevertheless, not all built-in datasets are primitives. In JavaScript, there are 5 different forms of basic data. The following values are available:

1. Boolean
2. Undefined
3. Null
4. Number
5. String

**100. What is the role of deferred scripts in JavaScript?**

The processing of HTML code while the page loads are disabled by nature till the script hasn't halted. Your page will be affected if your network is a bit slow, or if the script is very hefty. When you use Deferred, the script waits for the HTML parser to finish before executing it. This reduces the time it takes for web pages to load, allowing them to appear more quickly.

**101. What has to be done in order to put Lexical Scoping into practice?**

To support lexical scoping, a JavaScript function object's internal state must include not just the function's code but also a reference to the current scope chain.

**102. What is the purpose of the following JavaScript code?**

**var** scope = "global scope";

**function** **check**()

{

**var** scope = "local scope";

**function** **f**()

{

**return** scope;

}

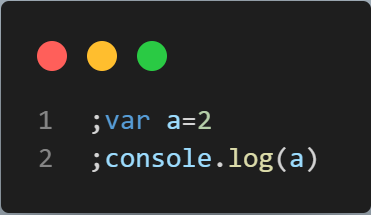
**return** f;

}

Every executing function, code block, and script as a whole in JavaScript has a related object known as the Lexical Environment. The preceding code line returns the value in scope.

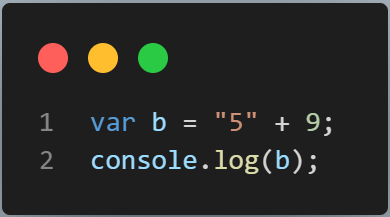
**Interesting Facts About Javascript!**

* A semicolon is commonly used to conclude a statement in computer languages. Although JavaScript already accomplishes this, statements can also start with a semicolon.



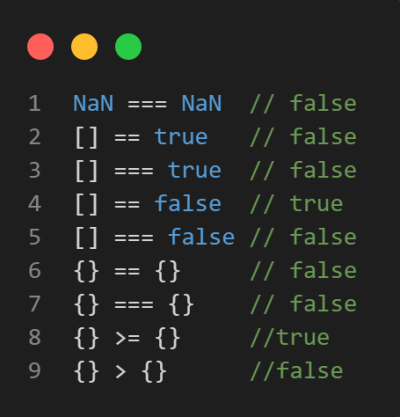
The above bit of code will display 2 in the console without raising an error!

* You can combine a number with a string in JavaScript. A string without any errors will be the outcome.

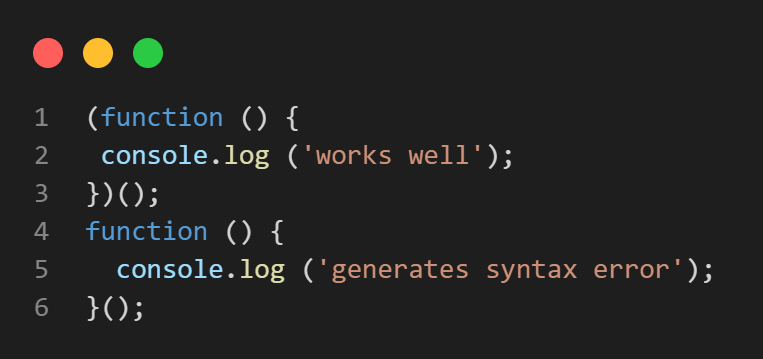


In the console, the code snippet will show "59"!

* When used in JavaScript, comparison operators frequently behave strangely. Observe a few instances.



* A useful feature of JavaScript is called Immediately Invoked Function Expression, which enables a function to be called automatically after it has been defined.

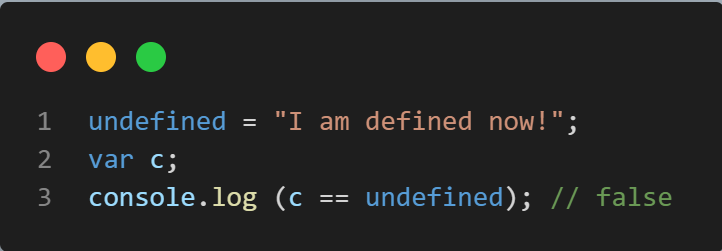


Here, the first function, which is an IIFE, operates without issue, while the second function produces a syntax error.

* Two functions in JavaScript can be distinguished from one another by the placement of the parenthesis.



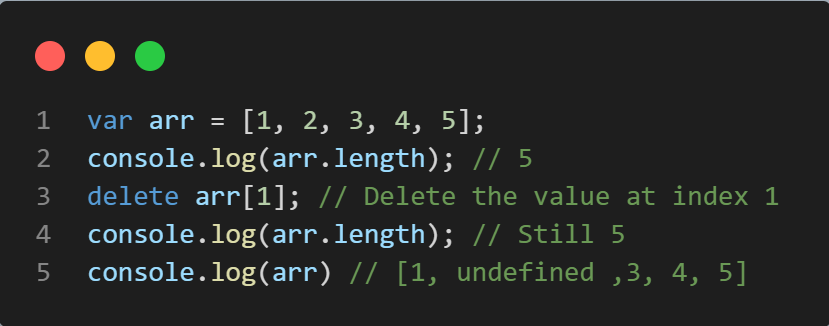
* Undefined is not a reserved word in JavaScript despite having a specific meaning. Although the following code snippet appears strange, this is the only way to tell if a variable is undefined.



* You cannot use src and the codes in a single <script> tags like below:​​​​​​​

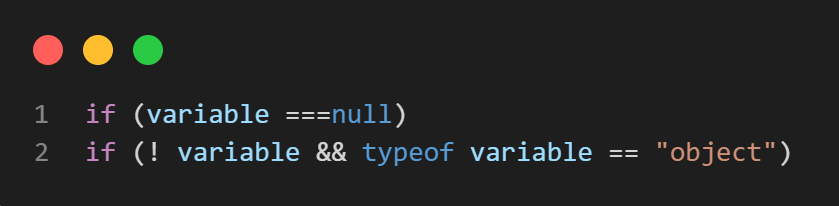


* Delete A Value from Array



Never fall victim to the delete keyword trap. The value of the array will be deleted, but the value will still be there. The value will be deleted if the keyword delete is used, but the array will not actually be re-indexed. It will merely leave the slot vacant. The slot at index 1 in the aforementioned example will be empty, which is why the log entry for that slot reads "undefined." Therefore, even deleting all of the values from an array with the delete keyword will not alter the array's length. Use array.splice(1,1) instead to remove the value from the first to the first number of values.​​​​​​​

* Why type of null Is Object?​​​​​​​



Nothing here should surprise anyone. It is a JavaScript BUG that is 20 years old. Yes, that is an issue that was impossible to remedy because a significant amount of online code relies on the typeof null === "object" phrase. If not, we anticipate the browser to return "null" as the string value when typeof null is used. Use the following methods to see if a variable has a null value type. The only value that is false yet still an object is Null, which is false