**JAVASCRIPT**

### What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.).

### History of JavaScript

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.

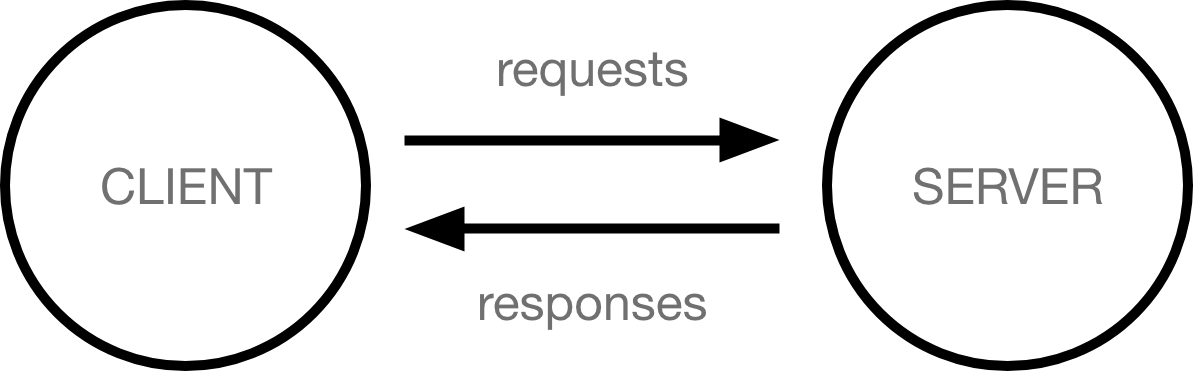
### Installing basic software’s

**Required tools :**

1. Text Editor
2. Modern Web Browsers

**How the web works**

Computers connected to internet are called Clients & Server , The Below Diagram show how they interact

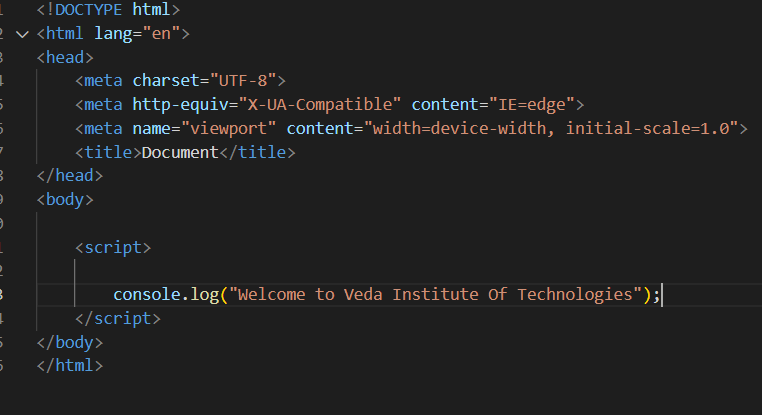


**How to write JavaScript**

We can write JavaScript in Two ways

1. Internal
2. External

**Internal JavaScript:**



We use **script** tag to write internal JavaScript.

We write the JavaScript at bottom of the body only.

**External JavaScript**

**index.js**

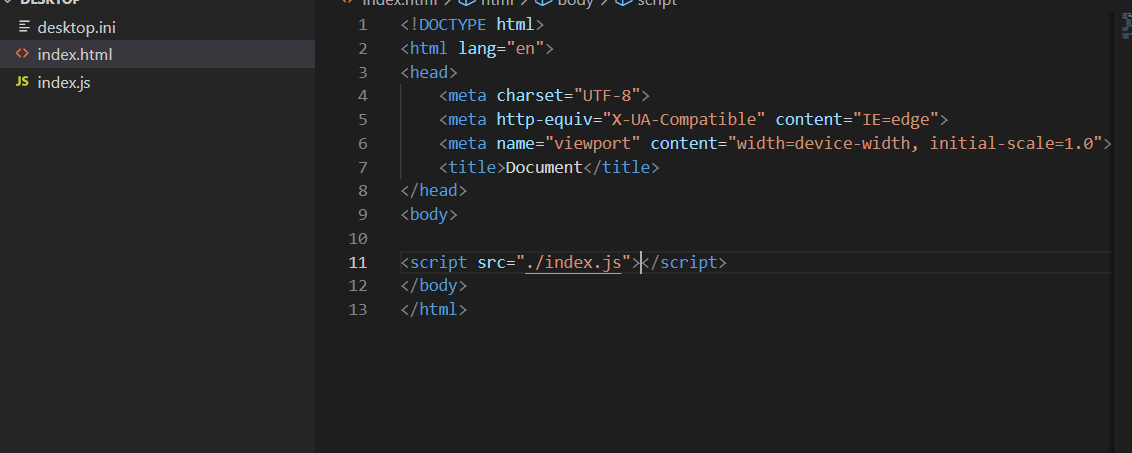
Text

Description automatically generated

1. Create a new file with **.js** extension
2. Then write JavaScript code in created file
3. Reference the JS file in Web page

**Referencing JS**

<script src=”[path of the js file]”></script>



**Note : We always reference JS file at the bottom of the body only**

**Variables**

Variables  are containers that store values. You start by declaring a variable with the let keyword, followed by the name you give to the variable:

**let myVariable;**

Semicolon Indicates the end of Statement

After declaring a variable, you can give it a value:

myVariable = 'Bob';

Also, you can do both these operations on the same line:

let myVariable = 'Bob';

You retrieve the value by calling the variable name

console.log(myVariable);

After assigning a value to a variable, you can change it later in the code

myVariable = “Heelo”;

If you want a general rule: always declare variables with const.

If you think the value of the variable can change, use let.

Note that Variable may store the values that have different data types

**why do we need variables?**

Variables are necessary to do anything interesting in programming. If values couldn't change, then you couldn't do anything dynamic

| **Variable** | **Explanation** | **Example** |
| --- | --- | --- |
| [**String**](https://developer.mozilla.org/en-US/docs/Glossary/String) | This is a sequence of text known as a string. To signify that the value is a string, enclose it in single quote marks. | let myVariable = 'Bob'; |
| [**Number**](https://developer.mozilla.org/en-US/docs/Glossary/Number) | This is a number. Numbers don't have quotes around them. | let myVariable = 10; |
| [**Boolean**](https://developer.mozilla.org/en-US/docs/Glossary/Boolean) | This is a True/False value. The words true and false are special keywords that don't need quote marks. | let myVariable = true; |
| [**Array**](https://developer.mozilla.org/en-US/docs/Glossary/array) | This is a structure that allows you to store multiple values in a single reference. | let myVariable = [1,'Bob','Steve',10]; Refer to each member of the array like this: myVariable[0], myVariable[1], etc. |
| [**Object**](https://developer.mozilla.org/en-US/docs/Glossary/Object) | This can be anything. Everything in JavaScript is an object and can be stored in a variable. Keep this in mind as you learn. | let myVariable = document.querySelector('h1'); All of the above examples too. |
|  |  |  |

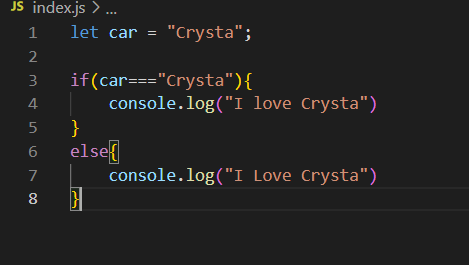
**Operators**

An Operator is mathematical symbol that produces a result based on two values or variables

| **Operator** | **Explanation** | **Symbol(s)** | **Example** |
| --- | --- | --- | --- |
| **Addition** | Add two numbers together or combine two strings. | + | 6 + 9; 'Hello ' + 'world!'; |
| **Subtraction, Multiplication, Division** | These do what you'd expect them to do in basic math. | -, \*, / | 9 - 3; 8 \* 2; // multiply in JS is an asterisk 9 / 3; |
| **Assignment** | As you've seen already: this assigns a value to a variable. | = | let myVariable = 'Bob'; |
| **Strict equality** | This performs a test to see if two values are equal. It returns a true/false (Boolean) result. | === | let myVariable = 3; myVariable === 4; |
| **Not, Does-not-equal** | This returns the logically opposite value of what it precedes. It turns a true into a false, etc.. When it is used alongside the Equality operator, the negation operator tests whether two values are *not* equal. | !, !== | For "Not", the basic expression is true, but the comparison returns false because we negate it:  let myVariable = 3; !(myVariable === 3);  "Does-not-equal" gives basically the same result with different syntax. Here we are testing "is myVariable NOT equal to 3". This returns false because myVariable IS equal to 3:  let myVariable = 3; myVariable !== 3; |

**Conditions**

Conditionals are code structures used to test if an expression returns true or not. A very common form of conditionals is the if...else statement. For example:



**The Expression inside the if() is the test** This uses the strict equality operator to compare the variable car with the string Crysta

1. If the both values same the expression returns true . then first block will be executed
2. If the expression returns false ( if both values are not same) second block will be executed (false)

**Syntax:**

if (condition1)  
 statement1  
else if (condition2)  
 statement2  
else if (condition3)  
 statement3  
else  
 statementN

**Functions**

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is a set of statements to perform specific task.

To use a function, you must define it somewhere in the scope from which you wish to call it.

Functions allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times

**Defining Functions**

A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

* The name of the function.
* A list of parameters to the function, enclosed in parentheses and separated by commas.
* The JavaScript statements that define the function, enclosed in curly brackets, { /\* … \*/ }.

**Syntax:**

function name(parameter1, parameter2, parameter3) {  
 // code to be executed  
}

**Example :**

function sum(value1,value2){

return value1+value2;

}

The above function takes two parameters value1 and value2 & it performs the sum between two values and returns the result to calling area

let result = sum(2,4);

console.log(result) // here result gets the value of 6

**Calling Functions**

Defining function does not execute it . Defining It names the function and specifies that what to do when the function called

Calling the function will perform the action with indicated parameters

**Ex : sum(5,5);**

The above function calls the function with Two arguments 5 & 5 . the function will execute the statements and returns the result value

Functions must be in scope when they are called, but the function declaration can be hoisted( function hoisting) (appear below the call in the code),

**console.log(multifly(2,2);**

**function multifly(value1,value2){**

**return value1 \* value2;**

**}**

**Function Scope :**Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

Graphical user interface, text, application

Description automatically generated

**Types of Functions**

1. Named Functions
2. Anonymous Functions
3. Auto / Immediate Invoked Functions

**Named Functions :**

Named functions in JavaScript is just a fancy way to refer to a function that uses the function keyword and then a name you can use as a reference to that function

Ex function myFun(){

console.log(“Named Function”);

}

EX: <script>

function test() {

console.log (`This is a named function!`);

};

</script>

**Anonymous Function :**

An anonymous function is a function without a name.

EX : let myFun = function(){

// stmts

}

EX: <script>

var test = function () {

console.log("This is an anonymous function!");

};

test();

</script>

**Auto / Immediate Invoked Function :**

A Immediate Invoked function is a nameless (anonymous) function that is invoked immediately after its definition.

Ex : (function () {

// stmts

})();

**Strings:**

The **String** object is used to represent and manipulate a sequence of characters.

Ex : let str = “Veda Institute of Technologies”;

**String Methods :**

1. Length (Prop)
2. indexOf
3. lastIndexOf
4. replace
5. substring
6. substr
7. charAt
8. split
9. toUpperCase
10. toLowerCase

**length**: is a read-only property it contains the length of the string.

Ex : console.log(str.length); // Expected Output : 30

**indexOf :** it returns the index of the first occurrence of specified substring

Ex: console.log(str.indexOf(“Institute”)); // Output : 5

**lastIndexOf :** it returns the index of the last occurrence of specified substring

Ex: console.log(str.indexOf(“Institute”)); // Output : 5

**replace:** The **replace()** method returns a new string with one, some, or all matches of a pattern replaced by a replacement. The pattern can be a string or a [RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp), and the replacement can be a string or a function called for each match. If pattern is a string, only the first occurrence will be replaced. The original string is left unchanged.

Ex : console.log(str.replace(“Institute”,”Tech”));

// Out put : Vedat Tech of Technologies;

**substring :**

The **substring()** method returns the part of the string between the start and end indexes, or to the end of the string.

Ex : console.log(str.substring(0,4)); // Output : Veda

**substr (deprecated) :** The **substr()** method returns a portion of the string, starting at the specified index and extending for a given number of characters afterwards.

**chatAt :** charAt() method returns the character at specified index position

Ex: str.charAt(0) // Output : V

**split :** The **split()** method takes a pattern and divides a String into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and returns the array**.**

const str = 'The quick brown fox jumps over the lazy dog.';

const words = str.split(' ');

console.log(words[3]);

// expected output: "fox"

const chars = str.split('');

console.log(chars[8]);

// expected output: "k"

const strCopy = str.split();

console.log(strCopy);

// expected output: Array ["The quick brown fox jumps over the lazy dog."]

**toUpperCase() :** The **toUpperCase()** method returns the calling string value converted to uppercase (the value will be converted to a string if it isn't one).

const sentence = 'The quick brown fox jumps over the lazy dog.';

console.log(sentence.toUpperCase());

// expected output: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."

**toLowerCase()** : The toLowerCase() method returns the calling string value converted to lower case.

const sentence = 'The quick brown fox jumps over the lazy dog.';

console.log(sentence.toLowerCase());

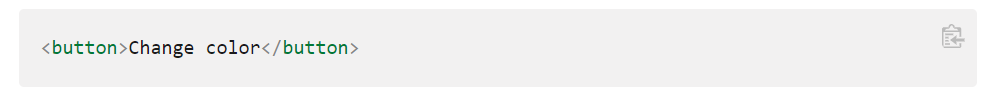
// expected output: "the quick brown fox jumps over the lazy dog."

**Events**

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so your code can react to them.

For example, if the user clicks a button on a webpage, you might want to react to that action by displaying an information box. In this article, we discuss some important concepts surrounding events, and look at how they work in browsers.

Let's look at a simple example of what we mean here. In the following example, we have a single <button>, which when pressed, makes the background change to a random color:



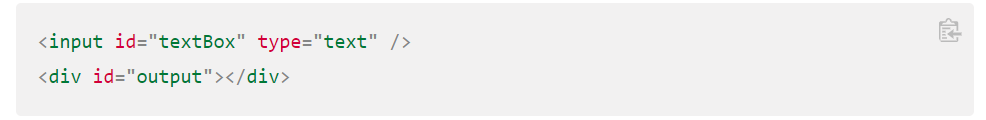
Graphical user interface, text, application

Description automatically generated

Other Events :

1. onblur
2. onmousehover
3. onmouseleave
4. onfocus
5. ondblclick
6. onkeypress
7. onkeyup

**Keyboard Event**



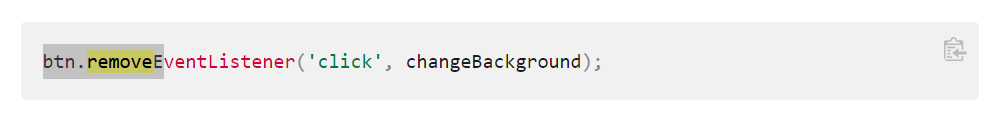
Text

Description automatically generated

**Removing Event**

If you've added an event handler using addEventListener(), you can remove it again using the removeEventListener() method. For example, this would remove the changeBackground() event handler:





**Arrays :**

a neat way of storing a list of data items under a single variable name

Arrays are generally described as "list-like objects"; they are basically single objects that contain multiple values stored in a list. Array objects can be stored in variables and dealt with in much the same way as any other type of value, the difference being that we can access each value inside the list individually, and do super useful and efficient things with the list, like loop through it and do the same thing to every value.

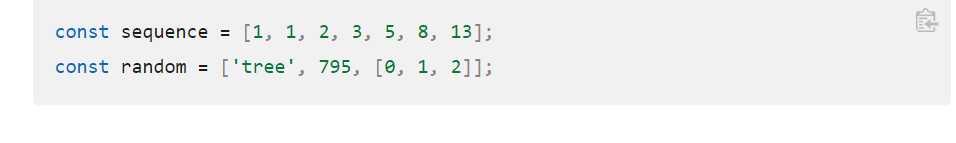
**Creating array**

Arrays consist of square brackets and items that are separated by commas.

A picture containing text

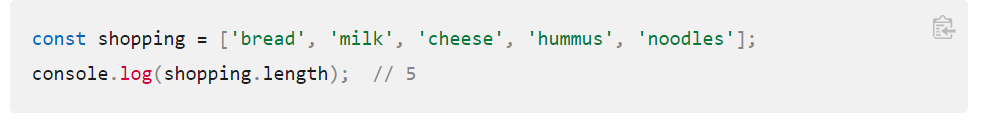
Description automatically generated

In the above example, each item is a string, but in an array we can store various data types — strings, numbers, objects, and even other arrays. We can also mix data types in a single array we do not have to limit ourselves to storing only numbers in one array, and in another only strings. For example:



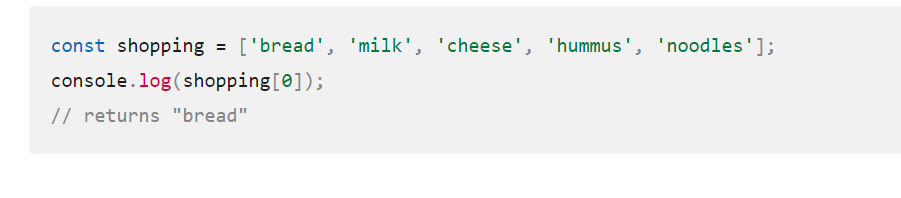
**Length**

You can find out the length of an array (how many items are in it) in exactly the same way as you find out the length (in characters) of a string — by using the length property. Try the following:



**Accessing and modifying array items:**

Items in an array are numbered, starting from zero. This number is called the item's index. So the first item has index 0, the second has index 1, and so on. You can access individual items in the array using bracket notation and supplying the item's index



We can modify an item in an array by giving a single array item a new value

Text

Description automatically generated with medium confidence

**IndexOf :**

You can find the index of a particular item using the [indexOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf) method. This takes an item as an argument and returns the index, or -1 if the item was not found in the array:

Text

Description automatically generated

**Push :**

To add one or more items to the end of an array we can use [push()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push). Note that you need to include one or more items that you want to add to the end of your array.

Graphical user interface, text

Description automatically generated

To add an item to the start of the array, use [unshift()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift):



**Pop**

To remove the last item from the array, use [pop()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop).

Graphical user interface

Description automatically generated

The pop() method returns the item that was removed. To save that item in a new variable, you could do this:

A picture containing graphical user interface

Description automatically generated

**Splice**

If you know the index of an item, you can remove it from the array using [splice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice):Text

Description automatically generated with medium confidence

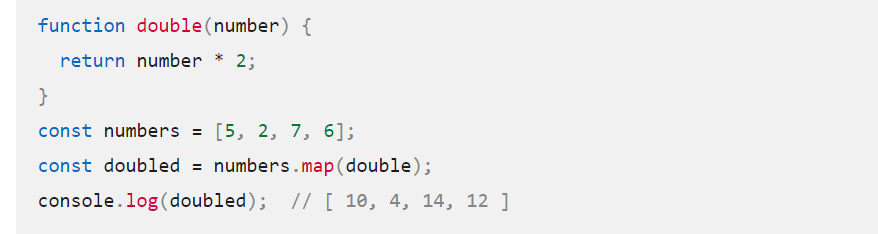
In this call to splice(), the first argument says where to start removing items, and the second argument says how many items should be removed. So you can remove more than one item:

Graphical user interface, text

Description automatically generated

**Map**

Sometimes you will want to do the same thing to each item in an array, leaving you with an array containing the changed items. You can do this using [map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map). The code below takes an array of numbers and doubles each number:



**Filter**

Sometimes you'll want to create a new array containing only the items in the original array that match some test. You can do that using [filter()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). The code below takes an array of strings and returns an array containing just the strings that are greater than 8 characters long:

Graphical user interface

Description automatically generated

**Split**

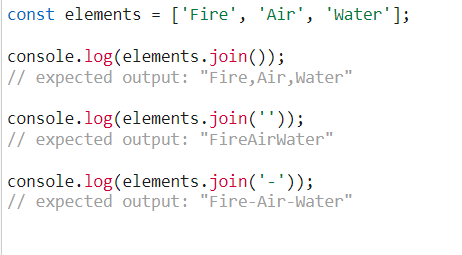
The **split()** method takes a pattern and divides a [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and returns the array.

Graphical user interface, text, application

Description automatically generated

**Join**

The **join()** method creates and returns a new string by concatenating all of the elements in an array separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

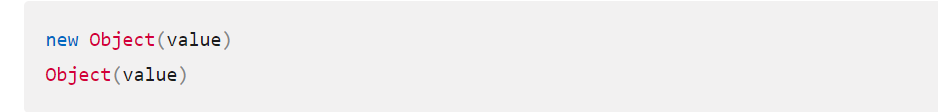


**Object**

The **Object** type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the Object() constructor or the object initializer / literal syntax.

The **Object constructor** turns the input into an object. Its behavior depends on the input's type.

* If the value is [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined), it will create and return an empty object.
* Otherwise, it will return an object of a Type that corresponds to the given value.
* If the value is an object already, it will return the value.



A picture containing shape

Description automatically generated

The following examples store an empty Object object in o

Graphical user interface, application

Description automatically generated

**Object Static methods**

**Object.assign()**

Copies the values of all enumerable own properties from one or more source objects to a target object.

**Object.create()**

Creates a new object with the specified prototype object and properties.

**Object.freeze()**

Freezes an object. Other code cannot delete or change its properties.

**Object.keys()**

Returns an array containing the names of all of the given object's **own** enumerable string properties.

For More Details [Click here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#static_methods)

# **Object initializer :**

An object initializer is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}).

Text

Description automatically generated

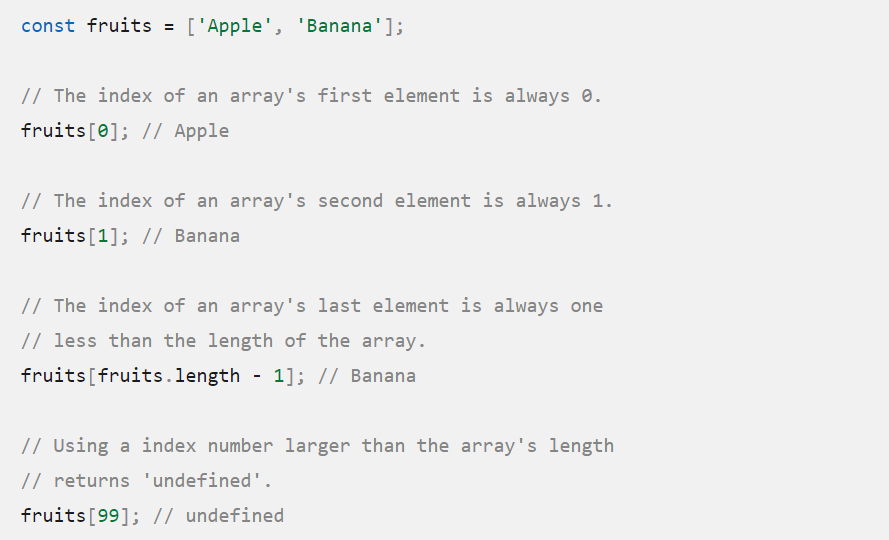
**Arrays**

Arrays are used to store collection of multiple values under single variable name.

**JavaScript arrays are resizable and can contain a mix of different data types**.

**JavaScript arrays are zero-indexed**: the first element of an array is at index 0, the second is at index 1, and so on, and the last element is at the value of the array's length property minus 1.

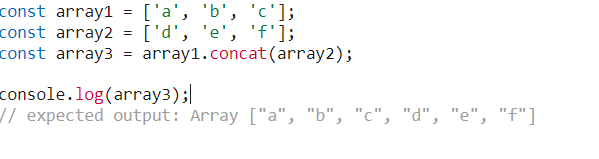
**Accessing an array item by index:**



**Methods**

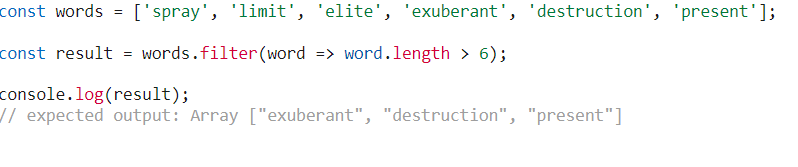
**concat():**

The **concat()** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.



**filter():**

Returns a new array containing all elements of the calling array for which the provided filtering function returns true.



[**find()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find)

Returns the value of the first element in the array that satisfies the provided testing function, or undefined if no appropriate element is found.

Example:

<script>

const ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.find(checkAge);

function checkAge(age) {

return age > 18;

}

</script>

**findIndex()**

Returns the index of the first element in the array that satisfies the provided testing function, or -1 if no appropriate element was found.

Example:

<script>

const ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.findIndex(checkAge);

function checkAge(age) {

return age > 18;

}

</script>

**findLast()**

Returns the value of the last element in the array that satisfies the provided testing function, or undefined if no appropriate element is found.

Example:

const array1 = [5, 12, 50, 130, 44];

const found = array1.findLast((element) => element > 45);

console.log(found);

// expected output: 130

**findLastIndex():**

Returns the index of the last element in the array that satisfies the provided testing function, or -1 if no appropriate element was found.

Example:

const array1 = [5, 12, 50, 130, 44];

const isLargeNumber = (element) => element > 45;

console.log(array1.findLastIndex(isLargeNumber));

// expected output: 3 (of element with value: 130)

**forEach()**

Calls a function for each element in the calling array.

Example:

<script>

let text = "";

const fruits = ["apple", "orange", "cherry"];

fruits.forEach(myFunction);

document.getElementById("demo").innerHTML = text;

function myFunction(item, index) {

text += index + ": " + item + "<br>";

}

</script>

o/p:

0: apple  
1: orange  
2: cherry

[**includes()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes)

Determines whether the calling array contains a value, returning true or false as appropriate.

Example:

<script>

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

document.getElementById("demo").innerHTML = fruits.includes("Mango");

</script>

[**indexOf()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf)

Returns the first (least) index at which a given element can be found in the calling array.

Example:

<script>

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

let index = fruits.indexOf("Apple");

document.getElementById("demo").innerHTML = index;

</script>

o/p:

2

**join()**

Joins all elements of an array into a string.

Example:

<script>

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

let text = fruits.join();

document.getElementById("demo").innerHTML = text;

</script>

o/p:

Banana,Orange,Apple,Mango

[**lastIndexOf()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf)

Returns the last (greatest) index at which a given element can be found in the calling array, or -1 if none is found.

Example:

<script>

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

let index = fruits.lastIndexOf("Apple");

document.getElementById("demo").innerHTML = index;

</script>

o/p:

2

[**map()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

Returns a new array containing the results of invoking a function on every element in the calling array.

Example:

<script>

const numbers = [4, 9, 16, 25];

document.getElementById("demo").innerHTML = numbers.map(Math.sqrt);

</script>

o/p:

2,3,4,5

[**pop()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop)

Removes the last element from an array and returns that element.

Example:

<script>

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

fruits.pop();

document.getElementById("demo").innerHTML = fruits;

</script>

o/p:

Banana,Orange,Apple

[**push()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push)

Adds one or more elements to the end of an array, and returns the new length of the array.

Example:

<script>

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

fruits.push("Kiwi");

document.getElementById("demo").innerHTML = fruits;

</script>

o/p:

Banana,Orange,Apple,Mango,Kiwi

[**reduce()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)

Executes a user-supplied "reducer" callback function on each element of the array (from left to right), to reduce it to a single value.

Example:

<script>

const numbers = [175, 50, 25];

document.getElementById("demo").innerHTML = numbers.reduce(myFunc);

function myFunc(total, num) {

return total - num;

}

</script>

o/p:

100

[**reduceRight()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight)

Executes a user-supplied "reducer" callback function on each element of the array (from right to left), to reduce it to a single value.

Example:

<script>

const numbers = [2, 45, 30, 100];

document.getElementById("demo").innerHTML = numbers.reduceRight(getSum);

function getSum(total, num) {

return total - num;

}

</script>

o/p:

23

[**reverse()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse)

Reverses the order of the elements of an array *in place*. (First becomes the last, last becomes first.)

Example:

<script>

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

document.getElementById("demo").innerHTML = fruits.reverse();

</script>

o/p:

Mango,Apple,Orange,Banana

[**shift()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift)

Removes the first element from an array and returns that element.

Example:

<script>

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

fruits.shift();

document.getElementById("demo").innerHTML = fruits;

</script>

o/p:

Orange,Apple,Mango

[**slice()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice)

Extracts a section of the calling array and returns a new array.

Example:

<script>

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

const citrus = fruits.slice(1, 3);

document.getElementById("demo").innerHTML = citrus;

</script>

o/p:

Orange,Lemon

[**some()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/some)

Returns true if at least one element in the calling array satisfies the provided testing function.

Example:

<script>

const ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.some(checkAdult);

function checkAdult(age) {

return age > 18;

}

</script>

o/p:

true

[**sort()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort)

Sorts the elements of an array in place and returns the array.

Example:

<script>

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

document.getElementById("demo").innerHTML = fruits.sort();

</script>

o/p:

Apple,Banana,Mango,Orange

[**splice()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)

Adds and/or removes elements from an array.

Example:

<script>

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

// At position 2, add 2 elements:

fruits.splice(2, 0, "Lemon", "Kiwi");

document.getElementById("demo").innerHTML = fruits;

</script>

o/p:

Banana,Orange,Lemon,Kiwi,Apple,Mango

[**unshift()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift)

Adds one or more elements to the front of an array, and returns the new length of the array.

Example:

<script>

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

fruits.unshift("Lemon", "Pineapple");

document.getElementById("demo").innerHTML = fruits;

</script>

o/p:

Lemon,Pineapple,Banana,Orange,Apple,Mango

# **Loops and iteration**

Loops offer a quick and easy way to do something repeatedly.

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times. (Note that it's possible that number could be zero!)

The various loop mechanisms offer different ways to determine the start and end points of the loop. There are various situations that are more easily served by one type of loop over the others.

**for statement:**

A [for](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for) loop repeats until a specified condition evaluates to false



Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

**do…while statement :**

The **do...while statement** creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Text, letter

Description automatically generated

**while statement**

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

Text, letter

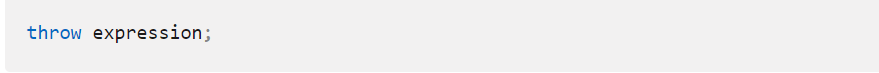
Description automatically generated

**Exception :**

An **exception** is a condition that interrupts normal code execution. In JavaScript syntax errors are a very common source of exceptions.

**throw statement**

Use the throw statement to throw an exception. A throw statement specifies the value to be thrown:



You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

Graphical user interface

Description automatically generated with medium confidence

**Handling Exceptions**

**try…catch statement:**

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

Text, application

Description automatically generated

**finally block :**

The finally block contains statements to be executed after the try and catch blocks execute. Additionally, the finally block executes before the code that follows the try…catch…finally statement.

It is also important to note that the finally block will execute whether or not an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the finally block closes the file before the script fails. Using finally here ensures that the file is never left open, even if an error occurs.

Text

Description automatically generated

If the finally block returns a value, this value becomes the return value of the entire try…catch…finally production, regardless of any return statements in the try and catch blocks:

Text

Description automatically generated with medium confidence

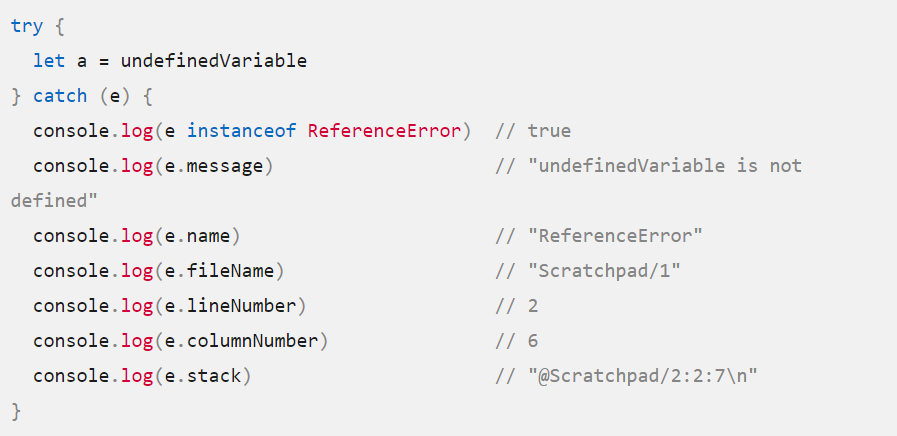
**Errors**

**Range Error :**

The **RangeError** object indicates an error when a value is not in the set or range of allowed values.

**ReferenceError :**

The **ReferenceError** object represents an error when a variable that doesn't exist (or hasn't yet been initialized) in the current scope is referenced.



**SyntaxError :**

The **SyntaxError** object represents an error when trying to interpret syntactically invalid code. It is thrown when the JavaScript engine encounters tokens or token order that does not conform to the syntax of the language when parsing code.

Graphical user interface, application

Description automatically generated

[**Click here**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types) **to see more error types**

**Promises :**

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

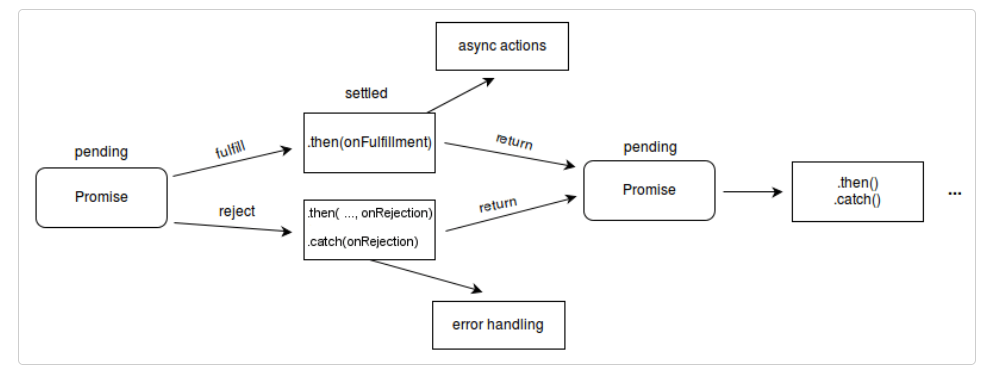
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

The *eventual state* of a pending promise can either be *fulfilled* with a value or *rejected* with a reason (error). When either of these options occur, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

A promise is said to be *settled* if it is either fulfilled or rejected, but not pending.



**.then()**

The **then()** method returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). It takes up to two arguments: callback functions for the success and failure cases of the Promise.

Text

Description automatically generated

**.catch()**

The **catch()** method returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) and deals with rejected cases only. It behaves the same as calling [Promise.prototype.then(undefined, onRejected)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)

Text, letter

Description automatically generated

**Date**

JavaScript **Date** objects represent a single moment in time in a platform-independent format. Date objects contain a Number that represents milliseconds since 1 January 1970 UTC.

# **Date() constructor**

The **Date()** constructor can create a [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) instance or return a string representing the current time

Graphical user interface, text, application

Description automatically generated

[Click here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date#static_methods) to see All date methods.

**Predefined Methods:**

**ParseInt:**

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

Ex:parseInt("40 years") is 40

**parseFloat():**

Parses a string argument and returns a floating-point number

Ex: parseFloat("10.33") is 10.33

**escape() :**

Returns the hexadecimal encoding of an argument

**unescape() :**

Returns the ASCII string for the specified value