**JAVASCRIPT**

**JavaScripts**

* HTML to define the content of web pages
* CSS to specify the layout of web pages
* JavaScript to program the behaviour of web pages

**JavaScript Variables:**

* Variable means anything that can vary. In JavaScript, a variable stores the data value that can be changed later.
* Use the reserved keyword var to declare a variable in JavaScript.

**Syntax:**

var <*variable-name*>;

var <*variable-name*> = <*value*>;

**Example: Variable Initialization:**

*var msg;*

*msg = "Hello JavaScript!";*

*alert(msg);*

*var num = 100;*

*var hundred = num;*

**Example: Multiple Variables in a Single Line**

*var one = 1, two = 'two', three;*

**Loosely typed Variables:**

* JavaScript is a loosely typed language. It means it does not require a data type to be declared. You can assign any literal values to a variable, e.g., string, integer, float, Boolean, etc.

**Example: Loosely Typed Variables**

*var myvariable = 1; // numeric value*

*myvariable = 'one'; // string value*

*myvariable = 1.1; // decimal value*

*myvariable = true; // Boolean value*

*myvariable = null; // null value*

**Hoisting of var:**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

*console.log (greeter).*

*var greeter = "say hello”.*

it is interpreted as this:

*var greeter.*

*console.log(greeter); // greeter is undefined*

*greeter = "say hello"*

So var variables are hoisted to the top of their scope and initialized with a value of undefined.

**Problem with var:**

There's a weakness that comes with var. I'll use the example below to explain:

*var greeter = "hey hi”;*

*var times = 4;*

*if (times > 3) {*

*var greeter = "say Hello instead”;*

*}*

*console.log(greeter) // "say Hello instead"*

**Let**

let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that we just covered. Let's consider why this is so.

**let is block scoped**

A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.

So, a variable declared in a block with let is only available for use within that block.

**let can be updated but not re-declared.**

Just like var, a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. So while this will work:

*let greeting = "say Hi”;*

*greeting = "say Hello instead”;*

this will return an error:

*let greeting = "say Hi”;*

*let greeting = "say Hello instead”;*

**Hoisting of let:**

use a let variable before declaration, you'll get a Reference Error.

**Const**

Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.

Ex:

*Const name = ‘suriya’;*

**JS datatypes:**

1. Primitive data type
2. Non-primitive (reference) data type

**Primitive Types:**

|  |  |
| --- | --- |
| **Data Type** | **Description** |
| String | represents sequence of characters e.g. "hello" |
| Number | represents numeric values e.g. 100 |
| Boolean | represents boolean value either false or true |
| Undefined | represents undefined value |
| Null | represents null i.e. no value at all |

**Non-Primitive Types:**

|  |  |
| --- | --- |
| **Data Type** | **Description** |
| Object | represents instance through which we can access members |
| Array | represents group of similar values |
| RegExp | represents regular expression |

**Objects:**

JavaScript variables can also contain many values.

Objects are variables too. But objects can contain many values.

Object values are written as **name : value** pairs (name and value separated by a colon).

**Example**

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

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

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

**Object Properties:**

The named values, in JavaScript objects, are called **properties**.

**JavaScript Arrays:**

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

*const cars = ["Saab", "Volvo", "BMW"];*

**Why Use Arrays?**

*let car1 = "Saab";  
let car2 = "Volvo";  
let car3 = "BMW”.*

**Creating an Array:**

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

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

**Example**

*const cars = ["Saab", "Volvo", "BMW"];*

You can also create an array, and then provide the elements:

**Example**

*const cars = [];  
cars[0]= "Saab";  
cars[1]= "Volvo";  
cars[2]= "BMW";*

**Accessing Array Elements**

You access an array element by referring to the **index number**:

*const cars = ["Saab", "Volvo", "BMW"];  
let car = cars[0];*

**Changing an Array Element**

**Example**

*const cars = ["Saab", "Volvo", "BMW"];  
cars[0] = "Opel";*

# JavaScript Array Methods:

# toString()

The JavaScript method toString() converts an array to a string of (comma separated) array values.

Ex:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
console.log(fruits.toString());

# join()

The join() method also joins all array elements into a string.

Ex:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
console.log(fruits.join(" \* "));

# pop()

The pop() method removes the last element from an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();

The pop() method returns the value that was "popped out":

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits.pop();

# push()

The push() method adds a new element to an array (at the end):

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");

The push() method returns the new array length:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let length = fruits.push("Kiwi");

# shift()

The shift() method removes the first array element and "shifts" all other elements to a lower index.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();

The shift() method returns the value that was "shifted out":

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits.shift();

# unshift()

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

The unshift() method returns the new array length.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

# length

The length property provides an easy way to append a new element to an array:

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

# delete()

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0];

# Note:

Array elements can be deleted using the JavaScript operator delete.

Use pop() or shift() instead.

# Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays.

const myGirls = ["Cecilie", "Lone"];  
const myBoys = ["Emil", "Tobias", "Linus"];  
  
const myChildren = myGirls.concat(myBoys);

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments:

const arr1 = ["Cecilie", "Lone"];  
const arr2 = ["Emil", "Tobias", "Linus"];  
const arr3 = ["Robin", "Morgan"];  
const myChildren = arr1.concat(arr2, arr3);

The concat() method can also take strings as arguments:

const arr1 = ["Emil", "Tobias", "Linus"];  
const myChildren = arr1.concat("Peter");

# Splicing and Slicing Arrays

The splice() method adds new items to an array.

The slice() method slices out a piece of an array.

# splice()

The splice() method can be used to add new items to an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");

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**.

The splice() method returns an array with the deleted items:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 2);

# splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

# slice()

The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1);

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.

const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1, 3);

# Note

The slice() method creates a new array.

The slice() method does not remove any elements from the source array.

# Sorting an Array

The sort() method sorts an array alphabetically:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();

# Reversing an Array

The reverse() method reverses the elements in an array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();  
fruits.reverse();

# Numeric Sort

By default, the sort() function sorts values as **strings**.

This works well for strings ("Apple" comes before "Banana").

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the sort() method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b});

Use the same trick to sort an array descending:

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a});

# The Compare Function

The purpose of the compare function is to define an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments:

function(a, b){return a - b}

When the sort() function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative, a is sorted before b.

If the result is positive, b is sorted before a.

If the result is 0, no changes are done with the sort order of the two values.

# The Difference between Arrays and Objects

* In JavaScript, **arrays** use **numbered indexes**.
* In JavaScript, **objects** use **named indexes**.

# JavaScript Conditional Statements

The **JavaScript if-else statement** is used to execute the code whether condition is true or false. There are three forms of if statement in JavaScript.

1. If Statement
2. If else statement
3. if else if statement

### If statement

It evaluates the content only if expression is true. The signature of JavaScript if statement is given below.

Syntax:

if(expression){

//content to be evaluated

}

### Flowchart of JavaScript If statement



### JavaScript If...else Statement

It evaluates the content whether condition is true of false. The syntax of JavaScript if-else statement is given below.

Ex:

if(expression){

//content to be evaluated if condition is true

}

else{

//content to be evaluated if condition is false

}

### Flowchart of JavaScript If...else statement



### JavaScript If...else if statement

It evaluates the content only if expression is true from several expressions. The signature of JavaScript if else if statement is given below.

if(expression1){

//content to be evaluated if expression1 is true

}

else if(expression2){

//content to be evaluated if expression2 is true

}

else if(expression3){

//content to be evaluated if expression3 is true

}

else{

//content to be evaluated if no expression is true

}

**Ex:**

var a=10;

if(a==10){

document.write("a is equal to 10");

}

else if(a==15){

document.write("a is equal to 15");

}

else if(a==20){

document.write("a is equal to 20");

}

else{

document.write("a is not equal to 10, 15 or 20");

}

# JavaScript Switch

The **JavaScript switch statement** is used to execute one code from multiple expressions. It is just like else if statement that we have learned in previous page. But it is convenient than if..else..if because it can be used with numbers, characters etc.

Syntax:

switch(expression){

case value1:

 code to be executed;

 break;

case value2:

 code to be executed;

 break;

......

default:

 code to be executed if above values are not matched;

}

***Ex:***

var grade='B';

var result;

switch(grade){

case 'A':

result="A Grade";

break;

case 'B':

result="B Grade";

break;

case 'C':

result="C Grade";

break;

default:

result="No Grade";

}

document.write(result);

# Conditional (ternary) operator

The **conditional (ternary) operator** is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) followed by a colon (:), and finally the expression to execute if the condition is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy). This operator is frequently used as an alternative to an [if...else](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) statement.

Syntax:

condition ? exprIfTrue : exprIfFalse

***Ex:***

const age = 26;

const beverage = age >= 21 ? "food": "Juice";

console.log(beverage); // " food "

### JavaScript functions

To avoid repeating the same code all over places, you can use a function to wrap that code and reuse it.

### Declare a function

To declare a function, you use the function keyword, followed by the function name, a list of parameters, and the function body as follows

function functionName(parameters) {

// function body

// ...

}

A function can accept zero, one, or multiple parameters. In the case of multiple parameters, you need to use a comma to separate two parameters.

The following declares a function **say**() that accepts no parameter:

function say() {

}

The following declares a function named square() that accepts one parameter:

function square(a) {

//

}

And the following declares a function named add() that accepts two parameters:

function add(a, b) {

//

}

### Calling a function

Calling a function is also known as invoking a function. To call a function, you use its name followed by arguments enclosing in parentheses like this

functionName(arguments);

*Ex:*

say('Hello');

### Parameters vs. Arguments

The terms parameters and arguments are often used interchangeably. However, they are essentially different.

When declaring a function, you specify the parameters. However, when calling a function, you pass the arguments that are corresponding to the parameters.

For example, in the **say**() function, the **message** is the parameter and the **'Hello'** string is an argument that corresponds to the **message** parameter.

### Returning a value

Every function in JavaScript implicitly returns undefined unless you explicitly specify a return value. For example:

function say(message) {

console.log(message);

}

let result = say('Hello');

console.log('Result:', result);

### Function hoisting

In JavaScript, you can use a function before declaring it. For example:

showMe(); // a hoisting example

function showMe(){

console.log('an hoisting example');

}

# Advantage of JavaScript function

* Code reusability: We can call a function several times so it save coding.
* Less coding: It makes our program compact. We don’t need to write many lines of code each time to perform a common task.

# Arrow Function:

* Arrow functions are introduced in ES6, which provides you a more accurate way to write the functions in JavaScript.
* They allow us to write smaller function syntax. Arrow functions make your code more readable and structured.
* Arrow functions are **anonymous functions** (the functions without a name and not bound with an identifier).
* They are also called as **Lambda Functions** in different language.

# Syntax:

const functionName = (param1, param2, ?..) => {

    //body of the function

}

There are three parts to an Arrow Function or Lambda Function:

* **Parameters:** Any function may optionally have the parameters.
* **Fat arrow notation/lambda notation:** It is the notation for the **arrow (=>).**
* **Statements:** It represents the instruction set of the function.

### Ex:

// function expression

var myfun1 = function show() {

 console.log("It is a Function Expression");

}

// Anonymous function

var myfun2 = function () {

    console.log("It is an Anonymous Function");

   }

//Arrow function

var myfun3 = () => {

    console.log("It is an Arrow Function");

   };

myfun1();

myfun2();

myfun3();

**Optional parentheses for the single parameter**

var num = x => {

    console.log(x);

}

num(140);

**Optional braces for single statement and the empty braces if there is not any parameter required.**

var show = () => console.log("Hello World");

show();

# Arrow Function with Parameters

If you require to pass more than one parameter by using an arrow function, then you have to pass them within the parentheses.

var show = (a,b,c) => {

    console.log(a + " " + b + " " + c );

}

show(100,200,300);

# Arrow function with default parameters

In ES6, the function allows the initialization of parameters with default values, if there is no value passed to it, or it is undefined.

var show = (a, b=200) => {

console.log(a + " " + b);

}

show(100);

In the above function, the value of **b** is set to **200** by default. The function will always consider **200** as the value of **b** if no value of **b** is explicitly passed.

var show = (a, b=200) => {

console.log(a + " " + b);

}

show(100,500);

# Advantages of Arrow Functions

* It reduces the code size.
* Return statement and Functional braces are optional for single line functions.
* Lexically bind the context lexically bind the context.

# Increment and Decrement:

++: Increment Operator

--: Decrement Operator

Post Increment:

Let num = 10;

Let result = num++; // result = (num+1)

Pre Increment:

Let result = ++num; // result = (num+1)

Post Decrement:

Let num = 10;

Let result = num--; // result = (num-1)

Pre Decrement:

Let result = --num; // result = (num-1)

# JavaScript Loops

The **JavaScript loops** are used *to iterate the piece of code* using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

There are four types of loops in JavaScript.

1. for loop
2. while loop
3. do-while loop
4. for-in loop
5. for-of loop

# For loop

The **JavaScript for loop** iterates the elements for the fixed number of times. It should be used if number of iteration is known.

Syntax:

for (initialization; condition; increment)

{

    code to be executed

}

Ex:

for (i=1; i<=5; i++)

{

console.log(i );

}

# while loop

The **JavaScript while loop** iterates the elements for the infinite number of times. It should be used if number of iteration is not known.

*Syntax:*

while (condition)

{

    code to be executed

}

*Ex:*

var i=11;

while (i<=15)

{

document.write(i + "<br/>");

i++;

}

# do while loop

The **JavaScript do while loop** iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false.

*Syntax:*

do{

    code to be executed

}while (condition);

*Ex:*

var i=21;

do{

document.write(i + "<br/>");

i++;

}while (i<=25);

# For In Loop

The JavaScript for in statement loops through the properties of an Object:

for (key in object) {  
  // *code block to be executed*  
}

*Ex:*

const person = {fname:"John", lname:"Doe", age:25};  
  
let text = "";  
for (let x in person) {  
  text += person[x];  
}

# For In Over Arrays

The JavaScript for in statement can also loop over the properties of an Array:

for (variable in array) {  
  code  
}

Ex:

const numbers = [45, 4, 9, 16, 25];  
  
let txt = "";  
for (let x in numbers) {  
  txt += numbers[x];  
}

# note:

Do not use **for in** over an Array if the index **order** is important.

The index order is implementation-dependent, and array values may not be accessed in the order you expect.

It is better to use a **for** loop, a **for of** loop, or **Array.forEach()** when the order is important.

# For Of Loop

The JavaScript for of statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more.

### Syntax:

for (variable of iterable) {  
  // *code block to be executed*  
}

# Looping over an Array

const cars = ["BMW", "Volvo", "Mini"];  
  
let text = "";  
for (let x of cars) {  
  text += x;  
}

# Looping over a String

let language = "JavaScript";  
  
let text = "";  
for (let x of language) {  
text += x;  
}

# ForEach:

The JavaScript array forEach() method is used to invoke the specified function once for each array element.

# Syntax

array.forEach(function(currentvalue,index,arr),thisArg)

# Parameter

|  |  |
| --- | --- |
| *function()* | Required.  A function to run for each array element. |
| *currentValue* | Required. The value of the current element. |
| *Index* | Optional. The index of the current element. |
| *Arr* | Optional. The array of the current element. |
| *thisValue* | Optional. Default undefined. A value passed to the function as its this value. |

### Return Value

Return value is undefined

Ex:

let sum = 0;  
const numbers = [65, 44, 12, 4];  
numbers.forEach(myFunction);  
  
function myFunction(item) {  
  sum += item;  
}

# filter()

* The filter() method creates a new array filled with elements that pass a test provided by a function.
* The filter() method does not execute the function for empty elements.
* The filter() method does not change the original array.

### Syntax

array.filter(function(currentValue, index, arr), thisValue)

### Parameters

|  |  |
| --- | --- |
| *function()* | Required. A function to run for each array element. |
| *currentValue* | Required. The value of the current element. |
| *Index* | Optional. The index of the current element. |
| *Arr* | Optional. The array of the current element. |
| *thisValue* | Optional. Default undefined A value passed to the function as its this value. |

### Return Value(array)

* Containing the elements that pass the test.
* If no elements pass the test it returns an empty array.

### Ex:

const ages = [32, 33, 16, 40];  
const result = ages.filter(checkAdult);  
  
function checkAdult(age) {  
  return age >= 18;  
}

# map()

* map() creates a new array from calling a function for every array element.
* map() calls a function once for each element in an array.
* map() does not execute the function for empty elements.
* map() does not change the original array.

# Syntax

array.map(function(currentValue, index, arr), thisValue)

# Parameter:

|  |  |
| --- | --- |
| *function()* | Required. A function to be run for each array element. |
| *currentValue* | Required. The value of the current element. |
| *Index* | Optional. The index of the current element. |
| *Arr* | Optional. The array of the current element. |
| *thisValue* | Optional. Default value undefined. A value passed to the function to be used as its this value. |

# Return Value(array)

The results of a function for each array element.

# every()

* The every() method executes a function for each array element.
* The every() method returns true if the function returns true for all elements.
* The every() method returns false if the function returns false for one element.
* The every() method does not execute the function for empty elements.
* The every() method does not change the original array

# Syntax

array.every(function(currentValue, index, arr), thisValue)

# Parameters

|  |  |
| --- | --- |
| Parameter | Description |
| function() | Required. A function to be run for each element in the array. |
| currentValue | Required. The value of the current element. |
| Index | Optional. The index of the current element. |
| Arr | Optional. The array of the current element. |
| thisValue | Optional. Default undefined. A value passed to the function as its this value. |

### Return Value(boolean)

**true** if all elements pass the test, otherwise **false**.

Ex:

const ages = [32, 33, 16, 40];  
  
ages.every(checkAge)  
  
function checkAge(age) {  
  return age > 18;  
}

# some()

* The some() method checks if any array elements pass a test (provided as a callback function).
* The some() method executes the callback function once for each array element.
* The some() method returns true (and stops) if the function returns true for one of the array elements.
* The some() method returns false if the function returns false for all of the array elements.
* The some() method does not execute the function for empty array elements.
* The some() method does not change the original array.

# Syntax

array.some(function(value, index, arr), this)

|  |  |
| --- | --- |
| Parameter | Description |
| *Function* | Required. A function to run for each array element. |
| Function parameters:   |  |  | | --- | --- | | *value* | Required. The value of the current element. | | *index* | Optional. The index of the current element. | | *arr* | Optional. The array the current element belongs to. | | |
| *This* | Optional. Default undefined. A value passed to the function to be used as its "this" value. |

# Return Value

true if any of the aray elements pass the test, otherwise false.

# Ex:

const ages = [3, 10, 18, 20];  
  
ages.some(checkAdult);   
function checkAdult(age) {  
  return age > 18;  
}

# JavaScript String Methods

# String charAt()

The JavaScript string charAt() method is used to find out a char value present at the specified index in a string.

# Ex:

var str="Javatpoint";

document.writeln(str.charAt(4));

# String charCodeAt()

The JavaScript string **charCodeAt()** method is used to find out the Unicode value of a character at the specific index in a string.

# Ex:

var x="Javatpoint";

document.writeln(x.charCodeAt(3));

# String concat()

The JavaScript string concat() method combines two or more strings and returns a new string. This method doesn't make any change in the original string.

# Ex:

var x="Javatpoint";

var y=".com";

document.writeln(x.concat(y));

let fistName = 'james';

let lastName = 'Vasanth';

let fullName = fistName +' '+ lastName;

console.log(fullName);

# indexOf()

var web="Learn JavaScript on Javatpoint";

document.write(web.indexOf('a'));

# lastIndexOf()

var web="Learn JavaScript on Javatpoint";

document.write(web.lastIndexOf('a'));

# String search()

# Ex 1:

var str="JavaScript is a scripting language. Scripting languages are often interpreted";

document.writeln(str.search("scripting"));

# Ex 2:

var str="JavaScript is a scripting language. Scripting languages are often interpreted";

document.writeln(str.search(/Scripting/));

# Ex 3:

var str="JavaScript is a scripting language. Scripting languages are often interpreted";

document.writeln(str.search(/Scripting/i));

# String match()

It searches a specified regular expression in a given string and returns that regular expression if a match occurs.

# Ex:

var str="Javatpoint";

document.writeln(str.match("Java"));

# String replace()

It replaces a given string with the specified replacement.

# Ex:

var str="Javatpoint";

document.writeln(str.replace("tpoint","Script"));

replaceAll:

str="Javatpoint Javatpoint Javatpoint Javatpoint ";

document.writeln(str.replace(/point/g,"Point"));

# String substr()

It is used to fetch the part of the given string on the basis of the specified starting position and length.

# Ex:

var str="Javatpoint";

document.writeln(str.substr(0,4));

# String substring()

It is used to fetch the part of the given string on the basis of the specified index.

# Ex:

var str="Javatpoint";

document.writeln(str.substring(0,4));

# String slice()

It is used to fetch the part of the given string. It allows us to assign positive as well negative index.

# Ex:

var str = "Javatpoint";

document.writeln(str.slice(2,5));

# toLowerCase()

It converts the given string into lowercase letter.

# Ex:

var str = "JAVATPOINT";

document.writeln(str.toLowerCase());

# String toString()

The JavaScript toString() method returns a string representing the calling object. In other words, this method provides a string representation of the object and treats same as the valueof() method.

# Ex:

var str="Javatpoint";

document.writeln(str.toString());

# String valueOf()

The JavaScript string valueOf() method is used to find out the primitive value of String object. This method is invoked by JavaScript automatically.

# Ex:

var str=new String("Javatpoint");

document.writeln(str.valueOf());

# String split()

It splits a string into substring array, then returns that newly created array.

Ex:

let str = "test";

console.log(str.split());

# String trim()

It trims the white space from the left and right side of the string.

Ex:

let str = " test";

console.log(str.trim());

# Date:

The **JavaScript date** object can be used to get year, month and day. You can display a timer on the webpage by the help of JavaScript date object.

Create date object before use below methods.

Let date = new Date();// Object creation

|  |  |
| --- | --- |
| Methods | Description |
| getDate() | It returns the integer value between 1 and 31 that represents the day for the specified date on the basis of local time. |
| getDay() | It returns the integer value between 0 and 6 that represents the day of the week on the basis of local time. |
| getFullYear() | It returns the integer value that represents the year on the basis of local time. |
| getHours() | It returns the integer value between 0 and 23 that represents the hours on the basis of local time. |
| getMilliseconds() | It returns the integer value between 0 and 999 that represents the milliseconds on the basis of local time. |
| getMinutes() | It returns the integer value between 0 and 59 that represents the minutes on the basis of local time. |
| getMonth() | It returns the integer value between 0 and 11 that represents the month on the basis of local time. |
| getSeconds() | It returns the integer value between 0 and 60 that represents the seconds on the basis of local time. |

# array.reduce()

Javascript **reduce()** is an inbuilt array method that executes a [callback function](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function) known as a **reducer** on each element of the array and results in a single output value.

The reduce() method executes the function for each value of the array (non-empty array) from left to right.

# Syntax:

let arr = [];

arr.reduce(function(acc, curVal, index, src), initVal);

# Parameters:

**Accumulator** | **required:** This is the accumulated value previously returned from the function or the initial value if it was supplied.

**Current Value** | **required:** This is the value of the current element in the array.

**Current Index | optional:** This is the index of the current element in the array.

**Source Array** | **optional:** This is the array object reduce() was called upon.

**Initial Value |** **optional:** This is the initial value to be passed to the function. If no initial value is supplied, the first element in the array will be used as the initial accumulator value and the second element becomes the current value

Ex:

const numbers = [1, 2, 3, 4, 5, 6];

function sum\_reducer(accumulator, currentValue) {

return accumulator + currentValue;

}

let sum = numbers.reduce(sum\_reducer);

console.log(sum); // 21

// using arrow function

let summation = numbers.reduce(

(accumulator, currentValue) => accumulator + currentValue

);

console.log(summation); // 21

# JavaScript DOM

* The document object represents the whole html document.
* When html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

# DOM Tree Structure:



### Selecting elements:

1. [getElementById()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementbyid/)
2. [getElementsByName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbyname/)
3. [getElementsByTagName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbytagname/)
4. [getElementsByClassName()](https://www.javascripttutorial.net/javascript-dom/javascript-getelementsbyclassname/)
5. [querySelector()](https://www.javascripttutorial.net/javascript-dom/javascript-queryselector/)

### JavaScript getElementById() method

* The document.getElementById() method returns an Element object that represents an HTML element with an id that matches a specified string.
* If the document has no element with the specified id, the document.getElementById() returns null.
* Because the id of an element is unique within an HTML document, the document.getElementById() is a quick way to access an element.

### Syntax:

const element = document.getElementById(id);

* In this syntax, the id is a string that represents the id of the element to select. The id is case-sensitive. For example, the 'root' and 'Root' are totally different.
* The id is unique within an HTML document. However, HTML is a forgiving language. If the HTML document has multiple elements with the same id, the document.getElementById() method returns the first element it encounters.

### Example:

<html>

<head>

<title>JavaScript getElementById() Method</title>

</head>

<body>

<p id="message">A paragraph</p>

</body>

</html>

* The document contains a <p> element that has the id attribute with the value message:

const p = document.getElementById('message');

console.log(p);

### getElementsByName() method

* Every element on an HTML document may have a name attribute:
* Unlike the id attribute, multiple HTML elements can share the same value of the name.

### Syntax:

let elements = document.getElementsByName(name);

* The getElementsByName() accepts a name which is the value of the name attribute of elements and returns a live NodeList of elements.
* The return collection of elements is live. It means that the return elements are automatically updated when elements with the same name are [inserted](https://www.javascripttutorial.net/dom/manipulating/insert-an-element-before-an-existing-element/) and/or [removed](https://www.javascripttutorial.net/javascript-dom/javascript-removechild/) from the document.

### Example:

<html>

<body>

<p>Please rate the service:</p>

<p>

<label for="very-poor">

<input type="radio" name="rate" value="Very poor" id="very-poor"> Very poor

</label>

<label for="poor">

<input type="radio" name="rate" value="Poor" id="poor"> Poor

</label>

<label for="ok">

<input type="radio" name="rate" value="OK" id="ok"> OK

</label>

<label for="good">

<input type="radio" name="rate" value="Good"> Good

</label>

<label for="very-good">

<input type="radio" name="rate" value="Very Good" id="very-good"> Very Good

</label>

</p>

<p>

<button id="btnRate">Submit</button>

</p>

<p id="output"></p>

</body>

</html>

JS code:

let btn = document.getElementById('btnRate');

let output = document.getElementById('output');

btn.addEventListener('click', () => {

let rates = document.getElementsByName('rate');

rates.forEach((rate) => {

if (rate.checked) {

output.innerText = `You selected: ${rate.value}`;

}

});

});

### getElementsByTagName() method

* The getElementsByTagName() is a method of the document object or a specific DOM element.
* The getElementsByTagName() method accepts a tag name and returns a live HTMLCollection of elements with the matching tag name in the order which they appear in the document.

Syntax:

let elements = document.getElementsByTagName(tagName);

* The return collection of the getElementsByTagName() is live, meaning that it is automatically updated when elements with the matching tag name are added and/or removed from the document.

Example:

<html>

<body>

<h1>JavaScript getElementsByTagName() Demo</h1>

<h2>First heading</h2>

<p>This is the first paragraph.</p>

<h2>Second heading</h2>

<p>This is the second paragraph.</p>

<h2>Third heading</h2>

<p>This is the third paragraph.</p>

<button id="btnCount">Count H2</button>

</body>

</html>

JS code:

let btn = document.getElementById('btnCount');

btn.addEventListener('click', () => {

let headings = document.getElementsByTagName('h2');

alert(`The number of H2 tags: ${headings.length}`);

});

### getElementsByClassName() method

* The getElementsByClassName() method returns an array-like of objects of the child elements with a specified class name.
* The getElementsByClassName() method is available on the document element or any other elements.
* When calling the method on the document element, it searches the entire document and returns the child elements of the document:

let elements = document.getElementsByClassName(names);

* Code language: JavaScript (javascript)
* However, when calling the method on a specific element, it returns the descendants of that specific element with the given class name:

let elements = rootElement.getElementsByClassName(names);

* Code language: JavaScript (javascript)
* The method returns the elements which is a live HTMLCollection of the matches elements.
* The *names* parameter is a string that represents one or more class names to match; To use multiple class names, you separate them by space.

### Example:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>JavaScript getElementsByClassName</title>

</head>

<body>

<header>

<nav>

<ul id="menu">

<li class="item">HTML</li>

<li class="item">CSS</li>

<li class="item highlight">JavaScript</li>

<li class="item">TypeScript</li>

</ul>

</nav>

<h1>getElementsByClassName Demo</h1>

</header>

<section>

<article>

<h2 class="secondary">Example 1</h2>

</article>

<article>

<h2 class="secondary">Example 2</h2>

</article>

</section>

</body>

</html>

### Example:

let menu = document.getElementById('menu');

let items = menu.getElementsByClassName('item');

for (let i = 0; i < collection.length; i++) {  
  items [i].style.backgroundColor = "red";  
}

### querySelector() and querySelectorAll() methods:

The querySelector() is a method of the Element interface. The querySelector() method allows you to select the first element that matches one or more CSS selectors.

let element = parentNode.querySelector(selector);

In this syntax, the selector is a CSS selector or a group of CSS selectors to match the descendant elements of the parentNode.

If the selector is not valid CSS syntax, the method will raise a SyntaxError exception.

If no element matches the CSS selectors, the querySelector() returns null.

The querySelector() method is available on the document object or any Element object.

Besides the querySelector(), you can use the querySelectorAll() method to select all elements that match a CSS selector or a group of CSS selectors:

let elementList = parentNode.querySelectorAll(selector);

The querySelectorAll() method returns a static NodeList of elements that match the CSS selector. If no element matches, it returns an empty NodeList.

**Note** that the NodeList is an array-like object, not an array object. However, in modern web browsers, you can use the [forEach()](https://www.javascripttutorial.net/javascript-array-foreach/) method or the [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop.

To convert the NodeList to an array, you use the Array.from() method like this:

let nodeList = document.querySelectorAll(selector);

let elements = Array.from(nodeList);

### 1) Universal selector

let element = document.querySelector('\*');

let elements = document.querySelectorAll('\*');

### 2) Type selector

let firstHeading = document.querySelector('h1');

let heading2 = document.querySelectorAll('h2');

### 3) Class selector

let note = document.querySelector('.menu-item');

let notes = document.querySelectorAll('.menu-item');

### 4) ID Selector

let logo = document.querySelector('#logo');

### 5) Attribute selector

let autoplay = document.querySelector('[autoplay]');

let autoplays = document.querySelectorAll('[autoplay]');

### Grouping selectors

let elements = document.querySelectorAll('div, p');

### Traversing elements:

1. [Get the parent element](https://www.javascripttutorial.net/javascript-dom/javascript-get-parent-element-parentnode/) – get the parent node of an element.
2. [Get child elements](https://www.javascripttutorial.net/javascript-dom/javascript-get-child-element/) – get children of an element.
3. [Get siblings of an element](https://www.javascripttutorial.net/javascript-dom/javascript-siblings/) – get siblings of an element.

### Manipulating elements:

# JavaScript CreateElement:

The document.createElement() accepts an HTML tag name and returns a new Node with the Element type.

let element = document.createElement(htmlTag);

# Ex:

let div = document.createElement('div');

div.id = 'content';

div.className = 'note';

// create a new text node and add it to the div

let text = document.createTextNode('CreateElement example');

div.appendChild(text);

let h2 = document.createElement('h2');

h2.textContent = 'Add h2 element to the div';

div.appendChild(h2);

document.body.appendChild(div);

# JavaScript appendChild

The appendChild() is a method of the Node interface. The appendChild() method allows you to add a node to the end of the list of child nodes of a specified parent node.

parentNode.appendChild(childNode);

# JavaScript textContent

To get the text content of a node and its descendants, you use the textContent property:

let text = node.textContent;

# Example:

<div id="note">

JavaScript textContent Demo!

<span style="display:none">Hidden Text!</span>

<!-- my comment -->

</div>

JS Code:

let note = document.getElementById('note');

console.log(note.textContent);

let note = document.getElementById('note');

console.log(note.innerText);

### textContent vs. innerText

Use the textContent property to return the concatenation of the textContent of every child node. You can use it to set a text for a node.

The innerText returns the human-readable text that takes CSS into account.

# JavaScript innerHTML

The innerHTML is a property of the Element that allows you to get or set the HTML markup contained within the element.

element.innerHTML = 'new content';

# Reading the innerHTML

let content = element.innerHTML;

# JavaScript removeChild

To remove a child element of a node, you use the removeChild() method:

let childNode = parentNode.removeChild(childNode);

The childNode is the child node of the parentNode that you want to remove. If the childNode is not the child node of the parentNode, the method throws an exception.

The removeChild() returns the removed child node from the DOM tree but keeps it in the memory, which can be used later.

Example:

<ul id="menu">

<li>Home</li>

<li>Products</li>

<li>About Us</li>

</ul>

Ex:

let menu = document.getElementById('menu');

menu.removeChild(menu.lastElementChild);