 ECMA is an committee which lays down the standards for a general purpose language that can also run on the browser. You can download the [ES2018 specification](https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf) and create your own implementation of this standard. JavaScript is a language which implements the standards. There are many other languages like JScript and ActionScript which also implement the ECMAScript standard.

About JavaScript

JavaScript is a language created in 1995, even before ECMAScript. It was later modified to adopt to the ES specifications. JavaScript was originally created as a scripting language for the web browser. But over the years, it has evolved into a full fledged general purpose language, that can not only run on the browser, but also on the server side.

JavaScript is an interpreted language created by [Brendan Eich](https://brendaneich.com/). The code is interpreted and executed by what is called a JavaScript engine ( also called as JavaScript Virtual Machine ). Every browser has a JavaScript engine. For example, chrome has V8, FireFox has SpiderMonkey, Edge has Chakra and so on. The JavaScript engine can exist outside the browser also. For example Node, Deno are some of the environments which use JavaScript outside the browser environment.

**Note:**In JS, even though ; (semicolon) is optional at the end of each statement, it is ideal to use it to avoid unforeseen consequences in the code execution

sample JavaScript code

placesToVisit = ["ooty", "coorg", "munnar"];

costPerPerson = [2500, 1500, 4500]; //corresponding value as per above array

totalAmount = null;

function calculateCost(myChoice, noOfPeople) {

try {

for (i = 0; i < placesToVisit.length; i++) {

if (myChoice == placesToVisit[i]) {

totalAmount = costPerPerson[i] \* noOfPeople;

console.log(totalAmount);

}

}

} catch (err) {

console.log("Some error occured");

}

}

calculateCost("coorg", 5);

### Data types

JavaScript has following built-in data types:

* number
* string
* boolean
* null
* undefined
* object

**Comments:**

We can write single line comments using // and multi-line comments using /\*  \*/

**Variable**

We can create variables directly by giving a name and assigning a value to it. Since JS is a dynamically typed language, the type of the variable is automatically determined based on the value assigned to it. For example, a="100" will create a string type variable and a=100 will create a number type variable.

### undefined vs null

undefined and null may seem the same, but they are different. Lets take a closer look at them.



**undefined:**

undefined is a variable which is declared, but not assigned any value.

var placeToVisit;

console.log(placeToVisit);

// undefined

JavaScript

Copy

**Note**: We will discuss about 'var' later in the course

**null:**

null is an empty value which means nothing. A variable left unassigned is not null. Null must be assigned explicitly.

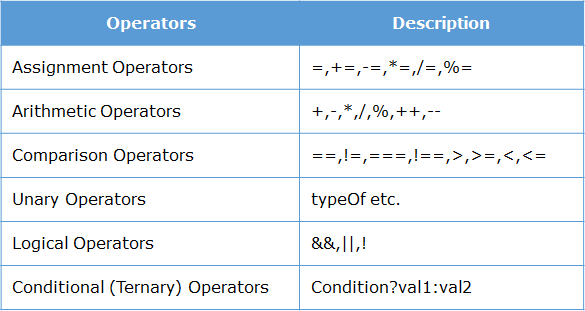
cabDetail = null;

console.log(cabDetail);

// null

Operators in JavaScript

There are several categories of operators in JavaScript:



### Double equals (==) vs Triple Equals (===)

**Double Equals (==)**

Double equals compares the values only, irrespective of the data types. For example:

console.log('100' == 100) // string & number

// true

var x = 5; // number

var y = '5' // string

console.log(x == y) // true

JavaScript

Copy

**Triple Equals (===)**

Double equals test **strict equality**which means **value**and **type** both should be same.For example:

console.log('100' === 100) // string & number

// false

var x = 5; // number

var y = '5' // string

console.log(x === y) // false

+ operator on string

Even if any one of the value is **string,**it will convert other values into string first and concatenate them into single string later. For example:

console.log('Trip' + 100) // 'Trip100'

console.log('Trip ' + undefined) // 'Trip undefined'

console.log('100'+20 //10020)

console.log(null + 'Trip') // 'nullTrip'

### typeof and exponentiation operators

**typeof**operator returns the data type of **variable / value**in string format.

console.log(typeof 100); // "number"

console.log(typeof 'Lets go to Trip'); // "string"

trip = null;

console.log(typeof trip); // "null"

JavaScript

Copy

**Exponentiation**operator (**\*\***) is used to create a value to the power of another value. This is similar to exponentiation operator in python.

console.log( 5 \*\* 2) // 25

val=5

console.log(val \*\* 2) // 25

ques.What is the output of the below code?

var package;

console.log("Package: " + package)

console.log("Amount: " + amount);

ans. Package: undefined  
An ReferenceError is thrown saying amount is not defined

**Note**: JS also supports a [do-while loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/do...while).

### Quick summary

* JavaScript is an interpreted language
* JS has 5 data types: number, string, undefined, null, object
* === is also called as strict equals
* typeof helps us identify the type of a variable or value
* JavaScript supports the regular conditional and iterational control structures

Creating functions

* function functionName(param1, param2,…){
* //statements
* }

### Functions as objects

In JavaScript, functions are actually objects. That means a function can be stored in a variable. For example:

funVariable= function myFunc(num1,num2) {

num3=num1\*num2;

return num3

}

JavaScript

Copy

Now that we have stored the function inside a variable, we can now invoke it either using the actual function name or by using the variable name. For example:

console.log(funVariable(10,20));

// 200

console.log(myFunc(10,20));

// 200

### Functions as objects

Since functions are treated as objects you can also pass them as a parameter to another function. For example, in the below code, we are passing the functions welcome() and goodbye() as parameters to the function greet()

function welcome(){console.log("Hello World");}

function goodbye(){console.log("See you later");}

function greet(choice){

choice();

}

greet(welcome);

greet(goodbye);

### Functions as objects

Since functions are treated as objects, you can also return them from a function. For example, in the below code, we are returning the function welcome() ( stored in variable hello ) from a function greet():

function greet(){

var hello=function welcome(){console.log("Hello World");}

return hello;

}

var retFunc=greet();

retFunc();

JavaScript

Copy

# **Higher Order Functions**

Functions which can either accept other functions as parameters or return other functions as parameters are called as Higher Order Functions. Many in-built functions in JS are Higher Order Functions.

# First Class Citizen:

Any object which can be assigned, passed as a parameter and returned from a function is called a First Class Citizen in a programming language. Thus,**all** functions are First Class Citizens in JS.

### Anonymous function in HOF

We have seen that we can pass one function as a parameter to another function. Usually, for such purposes we can create a function without a name. Such functions without a name are called anonymous functions. For example:

function greet(choice){

choice();

}

greet(function(){ console.log("Hello World")});

// Hello World

JavaScript

Copy

Here we can see that the function which is being passed as a parameter does not have a name.

### Arrow functions

An arrow function is a concise way of writing a function. Arrow functions are anonymous functions as they don't have a name.

Syntax:

(parameter) => function body

For example:

function greet(choice){

choice();

}

greet(function(){ console.log("Hello World") }); // Hello World

greet(()=>{ console.log("Hello World") }); // Hello World

### Arrow functions

Below are few scenario of arrow functions.

**Syntax 1**: Multi parameter, multi line code:

If code is in multiple lines, we need to have {}.

calculateCost = (ticketPrice, noOfPerson)=>{

noOfPerson= ticketPrice \* noOfPerson;

return noOfPerson;

}

console.log(calculateCost(500, 2));

// 1000

JavaScript

Copy

**Syntax 2**: No parameter, single line code:

If the code is single line, we don't need {}. The expression is evaluated and automatically returned.

trip = () => "Lets go to trip."

console.log(trip());

// Lets go to trip.

JavaScript

Copy

**Syntax 3**: One parameter, single line code:

If only one parameter, we don't need ().

trip = place => "Trip to " + place;

console.log(trip("ooty"));

// Trip to ooty

JavaScript

Copy

**Syntax 4:**One parameter, single line code:

if only one parameter, we can simply use '\_' and not use a variable name also.

trip = \_ => "Trip to " + \_;

console.log(trip("ooty"));

// Trip to ooty

Ques.Consider the JavaScript code:

myFun=function doSomething(a){

console.log(a\*a);

}

// line 4

Ans. myFun(10) not doSomething(10)

### Built in objects in JavaScript

Objects have properties and methods. JavaScript provides many standard **built-in** objects. In addition to that it also provides an option to create **user defined** objects.

Some of the commonly used built-in objects are:

* Array
* Date
* String

Array Object

Apart from constructing an array as shown above, we can also destructure an existing array. Just by assigning multiple variables to the array, we can access individual items. For example:

numArr=[100,200,300]

[a,b,c]=numArr;

// the numArr is now destructured and individual values are stored in the individual variables.

console.log(a);

console.log(b);

console.log(c);

### Array functions

Arrays comes with many built-in functions which makes working with array very easy in javascript.

**1. push()**

push() is used to insert a new element at the end of the array.

places = ["ooty", "coorg"];

places.push("munnar");

console.log(places);

// ["ooty", "coorg", "munnar"]

JavaScript

Copy

**2. pop()**

pop() is used to remove last element of array.

places = ["ooty", "coorg", "munnar"];

places.pop();

console.log(places);

// ["ooty", "coorg"]

JavaScript

Copy

**3. indexOf()**

indexOf() is used to find index of given elements.

places = ["ooty", "coorg", "munnar"];

console.log(places.indexOf("coorg"));

// 1

JavaScript

Copy

**Note**: indexOf() will return -1 if the value is not present

**4. splice(pos, n)**

splice(pos, n) will remove n elements from pos index position.

places = ["ooty", "coorg", "munnar"];

places.splice(1, 2); // This will remove 2 elements from index 1

console.log(places);

// ["ooty"]

JavaScript

Copy

**5. forEach()**

forEach() is used to interate over an array. This is a Higher Order Function. It will take a function and invoke that function on all elements of array.

places = ["ooty", "coorg", "munnar"];

places.forEach(function(place) {

console.log(place);

}

);

// ooty

// coorg

// munnar

Example

List each item in the array:

<button onclick="numbers.forEach(myFunction)">Try it</button>  
<p id="demo"></p>  
  
<script>  
demoP = document.getElementById("demo");  
var numbers = [4, 9, 16, 25];  
  
function myFunction(item, index) {  
    demoP.innerHTML = demoP.innerHTML + "index[" + index + "]: " + item + "<br>";   
}  
</script>

## **Definition and Usage**

The forEach() method calls a provided function once for each element in an array, in order.

**Note:** forEach() does not execute the function for array elements without values.

## **Syntax**

*array*.forEach(function(currentValue, index, arr), thisValue)

## **Parameter Values**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function(currentValue, index, arr) | Required. A function to be run for each element in the array. Function arguments:   |  |  | | --- | --- | | **Argument** | **Description** | | currentValue | Required. The value of the current element | | index | Optional. The array index of the current element | | arr | Optional. The array object the current element belongs to | |
| thisValue | Optional. A value to be passed to the function to be used as its "this" value. If this parameter is empty, the value "undefined" will be passed as its "this" value |

### Array functions

Let's look at some more built-in array functions.

**1. map()**

The ***map()*** array function generates a new array. It iterates over each element in the array, just like forEach. It invokes a function on each element, just like forEach.

But the difference, is forEach just invokes function on each item in the array. It does not create a new array. Hence, **map()**creates a new array based on what the function does.

placesToVisit = ["ooty","coorg","munnar"];

function display\_uppercase(place) {

return place.toUpperCase();

}

placesUpparCase = placesToVisit.map(display\_uppercase);

console.log(placesUpparCase);

JavaScript

Copy

**2. filter()**

What if we want to get all words whose length is greater than 5?

For this we can use **filter()**. filter accepts a function. It iterates over each element and creates a sub array if the function returns true.

placesToVisit = ["ooty","coorg","munnar"];

function filterPlaces(val) {

if (val.length > 5) {

return true;

}

}

filteredPlaces = placesToVisit.filter(filterPlaces);

console.log(filteredPlaces );

JavaScript

Copy

**3. find()**

Array has a method called **find()**. It returns the first element in the array which satisfies a given condition. It takes a callback. It executes the callback for each element in the array. If the callback returns true, then find returns the element for which the callback returned true and stops further iteration. If it was false for all elements, it returns undefined.

placesToVisit = ["ooty","coorg","munnar"];

function findPlaces(val) {

if (val.length > 5) {

return true;

}

}

foundPlaces = placesToVisit.find(findPlaces);

console.log(foundPlaces);

JavaScript

Copy

### Arrow functions revisited

**1. Arrow in forEach()**

Consider the ***forEach()*** function of an array. This function takes another function as parameter and invokes the function for every item in the array.

placesToVisit = ["ooty", "coorg", "munnar"];

placesToVisit.forEach(place => console.log("Trip to" + place));

// Trip to ooty

// Trip to coorg

// Trip to munnar

JavaScript

Copy

**2. Arrow in map()**

We know that an array object has a .map() function that creates a new array based on what the passed callback function does.

This map() can also be written using arrow function.

placesToVisit = ["ooty", "coorg", "munnar"];

placesUpperCase = placesToVisit.map(place => place.toUpperCase());

console.log(placesUpperCase);

// ["OOTY", "COORG", "MUNNAR"]

JavaScript

Copy

**3. Arrow in filter()**

We know that an array object has a .filter() function that returns a filtered sub array based on what the passed callback function does.

This filter() can also be written using arrow function.

placesToVisit = ["ooty", "coorg", "munnar"];

filteredPlace = placesToVisit.filter(place => place.length > 5);

console.log(filteredPlace);

// ["MUNNAR"]

JavaScript

Copy

**4. Arrow in find()**

We know that an array object has a .find() function that returns the first element in the array based on what is passed as callback function.

This find() can also be written using arrow function.

placesToVisit = ["ooty", "coorg", "munnar"];

findPlace = placesToVisit.find(place => place.length > 5);

console.log(findPlace);

// "MUNNAR"

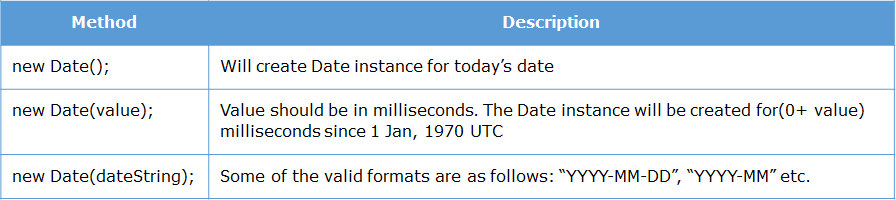
JavaScript

Copy

### Date object

**Date**is a built-in object which is used to create a **Date**instance. **Date**instance can be created using **new Date();**

Some methods of object are:



**Note**: Month in Date is 0 based.

## **JavaScript Date Output**

By default, JavaScript will use the browser's time zone and display a date as a full text string:

**Mon Sep 24 2018 16:53:59 GMT+0530 (India Standard Time)**

You will learn much more about how to display dates, later in this tutorial.

## **Creating Date Objects**

Date objects are created with the **new Date()** constructor.

There are **4 ways** to create a new date object:

new Date()  
new Date(year, month, day, hours, minutes, seconds, milliseconds)  
new Date(milliseconds)  
new Date(date string)

## **new Date()**

**new Date()** creates a new date object with the **current date and time**:

### Example

var d = new Date();

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

Date objects are static. The computer time is ticking, but date objects are not.

## **new Date(year, month, ...)**

**new Date(year, month, ...)** creates a new date object with a **specified date and time**.

7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

### Example

var d = new Date(2018, 11, 24, 10, 33, 30, 0);

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

JavaScript counts months from 0 to 11.

January is 0. December is 11.

6 numbers specify year, month, day, hour, minute, second:

### Example

var d = new Date(2018, 11, 24, 10, 33, 30);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers6)

5 numbers specify year, month, day, hour, and minute:

### Example

var d = new Date(2018, 11, 24, 10, 33);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers5)

4 numbers specify year, month, day, and hour:

### Example

var d = new Date(2018, 11, 24, 10);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers4)

3 numbers specify year, month, and day:

### Example

var d = new Date(2018, 11, 24);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers3)

2 numbers specify year and month:

### Example

var d = new Date(2018, 11);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers2)

You cannot omit month. If you supply only one parameter it will be treated as milliseconds.

### Example

var d = new Date(2018);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_new_numbers1)

## **Previous Century**

One and two digit years will be interpreted as 19xx:

### Example

var d = new Date(99, 11, 24);

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

### Example

var d = new Date(9, 11, 24);

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

## **new Date(dateString)**

**new Date(dateString)** creates a new date object from a **date string**:

### Example

var d = new Date("October 13, 2014 11:13:00");

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

Date strings are described in the next chapter.

## **JavaScript Stores Dates as Milliseconds**

JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).

Zero time is January 01, 1970 00:00:00 UTC.

Now the time is: **1537788239484** milliseconds past January 01, 1970

## **new Date(milliseconds)**

**new Date(milliseconds)** creates a new date object as**zero time plus milliseconds**:

### Example

var d = new Date(0);

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

01 January 1970 **plus** 100 000 000 000 milliseconds is approximately 03 March 1973:

### Example

var d = new Date(100000000000);

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

January 01 1970 **minus** 100 000 000 000 milliseconds is approximately October 31 1966:

### Example

var d = new Date(-100000000000);

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

### Example

var d = new Date(86400000);

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

One day (24 hours) is 86 400 000 milliseconds.

## **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.

Date methods and time zones are covered in the next chapters.

## **Displaying Dates**

JavaScript will (by default) output dates in full text string format:

Wed Mar 25 2015 05:30:00 GMT+0530 (India Standard Time)

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

When you display a date object in HTML, it is automatically converted to a string, with the **toString()** method.

### Example

d = new Date();  
document.getElementById("demo").innerHTML = d;

### Same as:

d = new Date();  
document.getElementById("demo").innerHTML = d.toString();

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

The **toUTCString()** method converts a date to a UTC string (a date display standard).

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.toUTCString();

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

The **toDateString()** method converts a date to a more readable format:

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.toDateString();

## **ISO Dates (Year and Month)**

ISO dates can be written without specifying the day (YYYY-MM):

### Example

var d = new Date("2015-03");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_iso2)

Time zones will vary the result above between February 28 and March 01.

## **ISO Dates (Only Year)**

ISO dates can be written without month and day (YYYY):

### Example

var d = new Date("2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_iso3)

Time zones will vary the result above between December 31 2014 and January 01 2015.

## **ISO Dates (Date-Time)**

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):

### Example

var d = new Date("2015-03-25T12:00:00Z");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_iso4)

Date and time is separated with a capital T.

UTC time is defined with a capital letter Z.

If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM instead:

### Example

var d = new Date("2015-03-25T12:00:00-06:30");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_iso5)

UTC (Universal Time Coordinated) is the same as GMT (Greenwich Mean Time).

Omitting T or Z in a date-time string can give different result in different browser.

## **Time Zones**

When setting a date, without specifying the time zone, JavaScript will use the browser's time zone.

When getting a date, without specifying the time zone, the result is converted to the browser's time zone.

In other words: If a date/time is created in GMT (Greenwich Mean Time), the date/time will be converted to CDT (Central US Daylight Time) if a user browses from central US.

## **JavaScript Short Dates.**

Short dates are written with an "MM/DD/YYYY" syntax like this:

### Example

var d = new Date("03/25/2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_7)

## **WARNINGS !**

In some browsers, months or days with no leading zeroes may produce an error:

var d = new Date("2015-3-25");

The behavior of "YYYY/MM/DD" is undefined.  
Some browsers will try to guess the format. Some will return NaN.

var d = new Date("2015/03/25");

The behavior of  "DD-MM-YYYY" is also undefined.  
Some browsers will try to guess the format. Some will return NaN.

var d = new Date("25-03-2015");

## **JavaScript Long Dates.**

Long dates are most often written with a "MMM DD YYYY" syntax like this:

### Example

var d = new Date("Mar 25 2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_1)

Month and day can be in any order:

### Example

var d = new Date("25 Mar 2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_2)

And, month can be written in full (January), or abbreviated (Jan):

### Example

var d = new Date("January 25 2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_4)

### Example

var d = new Date("Jan 25 2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_5)

Commas are ignored. Names are case insensitive:

### Example

var d = new Date("JANUARY, 25, 2015");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_string_6)

## **Date Input - Parsing Dates**

If you have a valid date string, you can use the **Date.parse()** method to convert it to milliseconds.

**Date.parse()** returns the number of milliseconds between the date and January 1, 1970:

### Example

var msec = Date.parse("March 21, 2012");  
document.getElementById("demo").innerHTML = msec;

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

You can then use the number of milliseconds to **convert it to a date** object:

### Example

var msec = Date.parse("March 21, 2012");  
var d = new Date(msec);  
document.getElementById("demo").innerHTML = d;

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

# **JavaScript Get Date Methods**

[❮ Previous](https://www.w3schools.com/js/js_date_formats.asp)[Next ❯](https://www.w3schools.com/js/js_date_methods_set.asp)

These methods can be used for getting information from a date object:

|  |  |
| --- | --- |
| **Method** | **Description** |
| getFullYear() | Get the **year** as a four digit number (yyyy) |
| getMonth() | Get the **month** as a number (0-11) |
| getDate() | Get the **day** as a number (1-31) |
| getHours() | Get the **hour** (0-23) |
| getMinutes() | Get the **minute** (0-59) |
| getSeconds() | Get the **second** (0-59) |
| getMilliseconds() | Get the **millisecond** (0-999) |
| getTime() | Get the time (milliseconds since January 1, 1970) |
| getDay() | Get the weekday as a number (0-6) |
| Date.now() | Get the time. ECMAScript 5. |

## **The getTime() Method**

The **getTime()** method returns the number of milliseconds since January 1, 1970:

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getTime();

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

## **The getFullYear() Method**

The **getFullYear()** method returns the year of a date as a four digit number:

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getFullYear();

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

## **The getMonth() Method**

The **getMonth()** method returns the month of a date as a number (0-11):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getMonth();

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

In JavaScript, the first month (January) is month number 0, so December returns month number 11.

You can use an array of names, and getMonth() to return the month as a name:

### Example

var d = new Date();  
var months = ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"];  
document.getElementById("demo").innerHTML = months[d.getMonth()];

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

## **The getDate() Method**

The **getDate()** method returns the day of a date as a number (1-31):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getDate();

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

## **The getHours() Method**

The **getHours()** method returns the hours of a date as a number (0-23):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getHours();

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

## **The getMinutes() Method**

The **getMinutes()** method returns the minutes of a date as a number (0-59):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getMinutes();

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

## **The getSeconds() Method**

The **getSeconds()** method returns the seconds of a date as a number (0-59):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getSeconds();

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

## **The getMilliseconds() Method**

The getMilliseconds() method returns the milliseconds of a date as a number (0-999):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getMilliseconds();

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

## **The getDay() Method**

The **getDay()** method returns the weekday of a date as a number (0-6):

### Example

var d = new Date();  
document.getElementById("demo").innerHTML = d.getDay();

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

In JavaScript, the first day of the week (0) means "Sunday", even if some countries in the world consider the first day of the week to be "Monday"

You can use an array of names, and getDay() to return the weekday as a name:

### Example

var d = new Date();  
var days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];  
document.getElementById("demo").innerHTML = days[d.getDay()];

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

## **UTC Date Methods**

UTC date methods are used for working with UTC dates (Universal Time Zone dates):

|  |  |
| --- | --- |
| **Method** | **Description** |
| getUTCDate() | Same as getDate(), but returns the UTC date |
| getUTCDay() | Same as getDay(), but returns the UTC day |
| getUTCFullYear() | Same as getFullYear(), but returns the UTC year |
| getUTCHours() | Same as getHours(), but returns the UTC hour |
| getUTCMilliseconds() | Same as getMilliseconds(), but returns the UTC milliseconds |
| getUTCMinutes() | Same as getMinutes(), but returns the UTC minutes |
| getUTCMonth() | Same as getMonth(), but returns the UTC month |
| getUTCSeconds() | Same as getSeconds(), but returns the UTC seconds |

# **JavaScript Set Date Methods**

## **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) |

### Creating object using object literal

To create an object using object literal, write your object attributes and methods.

The syntax is:

objName={

propertyName:value,

propertyName:value,

...

methodName(){

},

methodName(){

}

}

For example:

var empOne = {

name : "John",

empNumber : 1001,

emailId : "John@infy.com",

swipeIn(){console.log("Swipe In by "+this.name)}

};

None

Copy

In next page, how to access object property.

### Accessing object properties

1. To get the value: var name= empOne["name"];
2. To set the value: empOne["name"] = "John";

**Note**: **object[property]**access should be used mainly when the property names are having space, hyphen, or one that starts with a number.

### Iterating an object

Consider the below object literal:

var empOne = {

name : "John",

empNumber : 1001,

emailId : "John@infy.com"

};

JavaScript

Copy

Some of the ways of iterating over this object are:

**for..in:**

The for..in loop iterates over the object and gives the property names of the object. For example:

for(let property in empOne){

console.log(empOne[property]);

}

JavaScript

Copy

**Object.values():**

Object.values() will give all the values of an object in an array. For example:

console.log(Object.values(empOne));

JavaScript

Copy

**Object destructuring:**

We can destructure an existing object into variables. If the variable is prefixed by three dots (...) then it is called a rest variable and can store more than one property. Note that this feature is the latest ES2018 feature and is not yet supported by any browser.

{ a, ... rest } = empOne

console.log(a)

// 'John'

console.log(rest)

/\*{

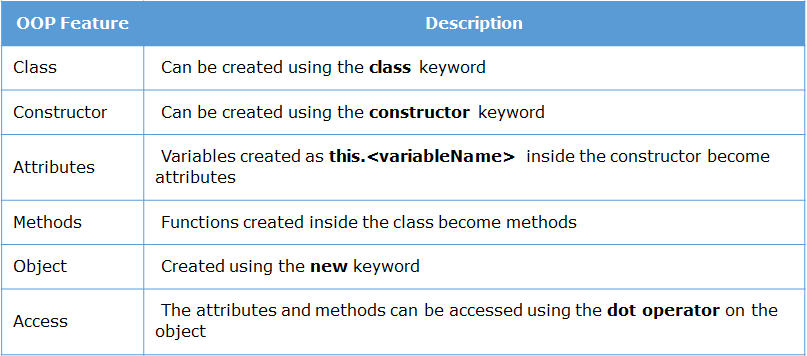
empNumber : 1001,

emailId : "John@infy.com"

}\*/

### Creating a class

JavaScript is a object oriented language. So far we have seen how to use built-in classes and objects like Date. We can also create our own classes in JavaScript.



The below code illustrates how to create a class with attributes and methods and how to create objects for the class.

**this** keyword is used to create attributes inside a class. This is equivalent to **self**in Python

class Employee{

constructor(id,name,age){

this.id=id;

this.name=name;

this.age=age;

}

swipeIn(){

console.log("Employee "+this.id+" has swiped in at "+new Date());

}

}

e1=new Employee(100,"Mark",23);

e2=new Employee(101,"Jane",24);

console.log(e1.age);

e1.swipeIn();

e2.swipeIn();

### Static methods in class

Just like in other programming languages, we can create static methods in JavaScript using the **static** keyword. Static values can be accessed only using the classname and not using **this**keyword. Else it will lead to an error.

In the below example, code is a static method and it is accessed using the classname.

class Employee{

constructor(id,name,age){

this.id=id;

this.name=name;

this.age=age;

}

swipeIn(){

console.log("Employee "+this.id+" has swiped in at "+new Date());

}

static code(){

console.log("Employee is coding");

}

}

Employee.code();

### Inhertiance

In JavaScript, one class can inherit another class using the extends keyword. The subclass inherits all the methods ( both static and non-static ) of the parent class.

In the below code, we are creating a child class called PartTimeEmployee which extends the Employee class.

class Employee{

constructor(id,name,age){

this.id=id;

this.name=name;

this.age=age;

}

swipeIn(){

console.log("Employee "+this.id+" has swiped in at "+new Date());

}

static code(){

console.log("Employee is coding");

}

}

class PartTimeEmployee extends Employee{

}

e1=new Employee(100,"Mark",23);

e2= new PartTimeEmployee();

PartTimeEmployee.code();

e2.swipeIn();

JavaScript

Copy

However, e2.swipeIn() gives undefined for the id. This is because we are not passing the values to the parent constructor. Let us look at how to achieve this.

### super keyword

In order to access the parent class constructor, the child class constructor need to invoke it using super() and pass the necessary values. Note that **super**keyword must appear before **this** keyword in constructor.

In the below example, we are passing the values to the parent class constructor using **super()**

class Employee{

constructor(id,name,age){

this.id=id;

this.name=name;

this.age=age;

}

swipeIn(){

console.log("Employee "+this.id+" has swiped in at "+new Date());

}

static code(){

console.log("Employee is coding");

}

}

class PartTimeEmployee extends Employee{

constructor(id,name,age,contractPeriod){

super(id,name,age);

this.contractPeriod=contractPeriod;

}

}

e1=new Employee(100,"Mark",23);

e2= new PartTimeEmployee(101,"Jane",34,3);

PartTimeEmployee.code();

e2.swipeIn();

console.log(e2.contractPeriod);

JavaScript

Copy

### JSON - JavaScript Object Notation

When two different programs want to pass data to each other, they need a commonly accepted format of passing that data. There many formats available like plain text, xml, json, etc.

JSON stands for JavaScript Object Notation. JSON is a way of formatting the text in such a way that it looks like a JavaScript literal object.

|  |  |
| --- | --- |
| **Object Literal** | **JavaScript Object Notation** |
| var emp\_one = {  name : "John",  empNumber : 1001,  emailId : "John@infy.com"  }  JavaScript  Copy | emp\_one='{"name":"John","empNumber":1001,"emailId":"John@infy.com"}'  JavaScript  Copy |

The keys in JSON string must be a string enclosed withing quotes. The values can be any valid JavaScript value: null, number, string, etc. It can have arrays as well as other JSON objects as values

JavaScript provides a standard built-in object called **JSON**which has methods for parsing and generating JSON data.

**parse():**This function is used to convert JSON string into an JavaScript object

var json = '{ "firstName":"Infosys", "lastName":"Limited", "pincode":570017 }';

var company = JSON.parse(json); //will convert JSON string into an JavaScript object

console.log(company.firstName +" "+ company.lastName +" "+ company.pincode );

JavaScript

Copy

**stringify():**This function is used to convert object to JSON string

var jScores = { "Java": 70, "JavaScript": 80, "CSS": 30 };

var tScores = JSON.stringify(jScores); //will convert object to JSON string

console.log(typeof(jScores)); // returns Object

console.log(typeof(tScores)); // returns String

### What are Regular Expressions

Regular Expression or regex is basically a sequence of characters indicating a pattern. With the help of this pattern, we can search or match with other strings which follow the pattern indicated.

We will now see how we can validate the name using regular expressions. Let's say that the name must not have $ symbol in it. Here is the implementation of the validateName() function.

function validateName(name){

if(name.match(/\$/)){

return false;

}

else {

return true;

}

}

JavaScript

Copy

The sequence of characters **/\$/**  is an **regular expression**. The regular expression used in the above code indicates "any character which is $"

When a regular expression is passed as a parameter to a match function, it checks if the pattern is present in the given string. If found, the **match** function returns true, else false.

### Parts of a Regular Expression

Various symbols and characters are used to form regular expressions.

# Three parts of Regular Expression :

* Meta characters
* Quantifiers
* Pre-defined classes (A pre-defined group of meta characters)

The above three concepts are very powerful tools to deal with almost any kind of string matching. Next we will understand them in detail.

### Steps to create a RegEx

Let us understand how to construct a RegEx and what are the components we need to use.

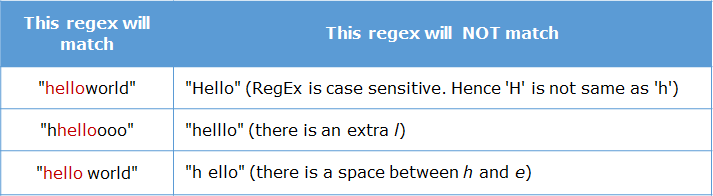
The following steps will help us create a regex to search or match any pattern:

# 

# **Step 1:** Create a regex pattern

A pattern is a series of characters that we want to search in the given test-string. Anything present in between '**/**' and '**/**' will become a pattern which can be used on a test-string.

For example, **/**hello**/** will check if the sequence of characters, in this case, h e l l o are present in the specified order, **anywhere** in the given string to be tested.

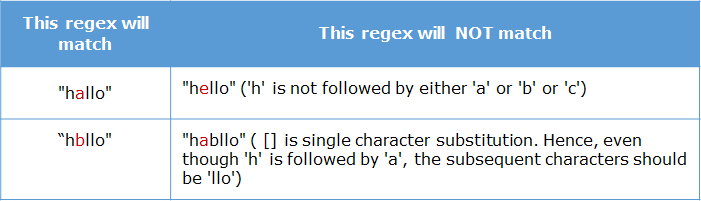


# **Step 2**: Use meta-characters like [], ^, $

A **meta-character** is a character that has a special meaning (instead of a literal meaning)

The meta character **[]**indicates a **single character.**

For example, /h**[abc]**llo**/** matches a single character present inside brackets



We also have **[^**abc**]**which will match any single character except the ones given inside brackets.

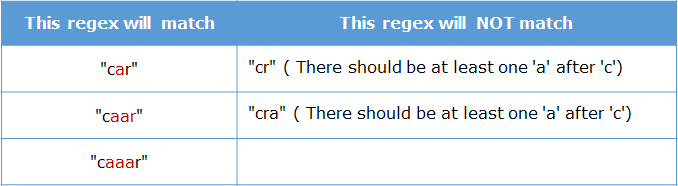
Hence, the pattern /h**[^**abc**]**llo/ will match "hello", "h!llo", "h6llo" but not "hallo"

# 

# **Step 3**: Use quantifiers like +, \*, ? and {n}

Quantifier are symbols which specify the **frequency**at which a character can appear.

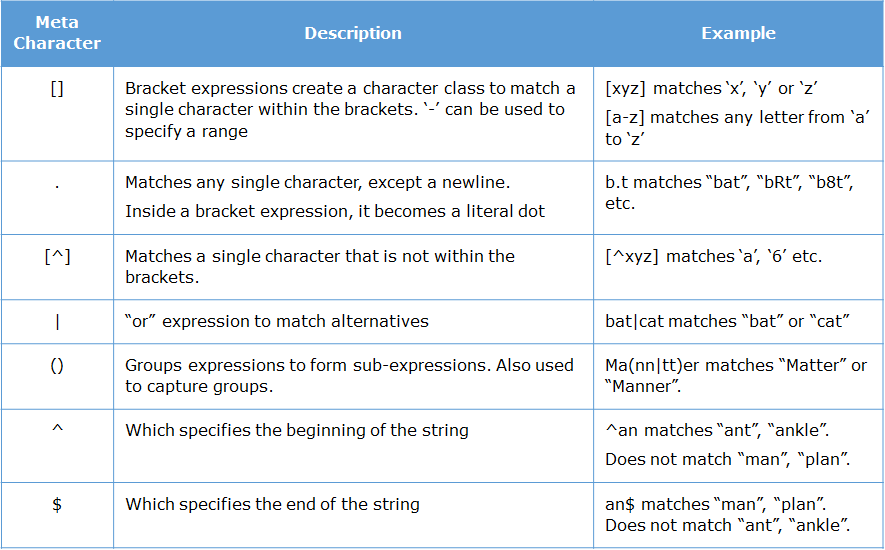
For example + quantifier matches the preceding element one or more times. For example, /c[a]**+**r/ checks if there is at least there is one 'a' after c.



### Understanding meta characters

Meta characters are the characters which have special meaning in regular expressions.

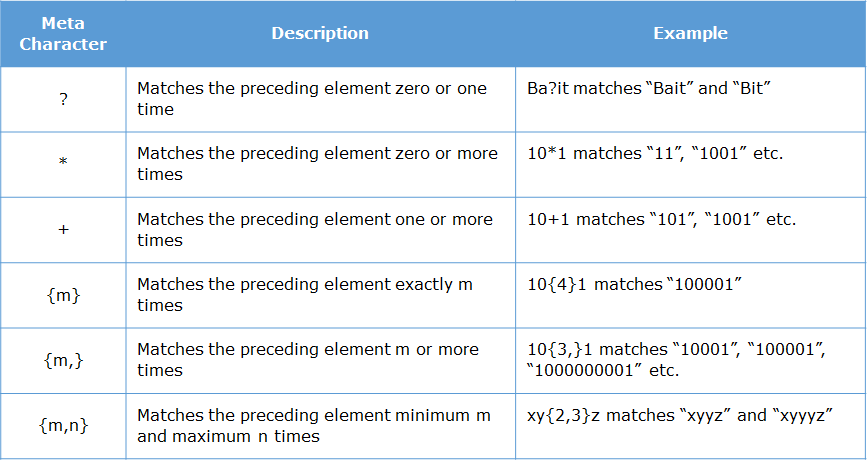
Following are few important meta characters:



### Understanding quantifiers

Quantifiers are the symbols which specify the frequency at which a character should/can appear.

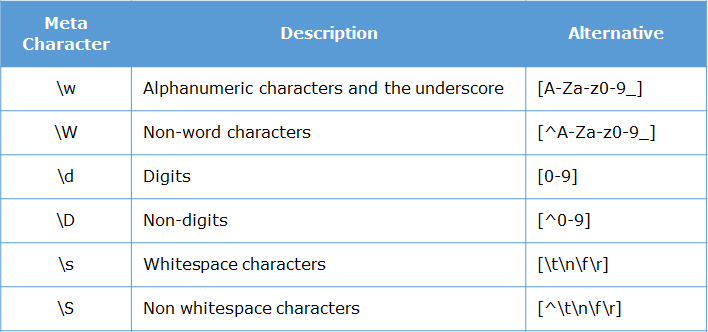
Following are a few quantifiers:



### Understanding predefined classes

Predefined classes are set of meta characters grouped together and given a special symbol.

These are the following commonly used predefined classes:



# 

# Escaping characters that have special meaning :

To escape any characters which has special meaning just prefix it by '\' . For example, \$ indicates escape the special meaning of $ and treat it as a regular character.

### What was the Error

The below code will throw an error. This is because, the validateName() method has a typing mistake. The method name is 'match' and not 'Match' with a uppercase 'M'. Because of this the code will crash.

function validateName(name){

if(name.Match(/[\$\#]/)){

console.log("Input is invalid");

return false;

}

else {

console.log("Input is valid");

return true;

}

}

function validate(){

validateName("Hello");

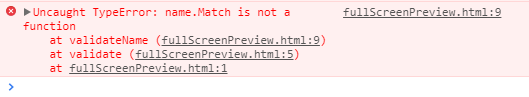
}

validate()

JavaScript

Copy

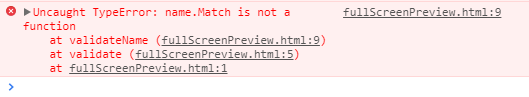
When the code crashes, the end user will not get any message in the browser screen. We have to look at the details of the error in the console. For example, the below is the output on the browser console when the error occurs.



### What is the error stack trace

When an error occurs, the **stack** **trace** of the error is displayed in the console.

It provides information regarding the name, message, error stack, and the location of the **error**, which helps in **debugging**.



# **How to trace the error?**

Press F12 on the keyboard, this will open the developer console window in Chrome. From the stack trace we know that:

* Error occurred at validateName() method
* The validateName() was called by validate() method
* validate() was invoked by click of button in the browser

The stack trace itself was shown in the browser console. That means the request:

* started at the browser,
* the control went to the validate()
* and from there to validateName()

The error started from validateName(), propagated to validate() and then propagated back to the browser.

Next let us learn more about error propagation.

### Exception propagation

In JavaScript, all errors are of type **error object**. These objects carry the information related to the error, including the stack trace.

Whenever an **exceptional event** occurs, the browser environment generates the error object and **throws** it. The moment an error object is thrown, further execution of program is stopped. If it error is not handled, then the error will be propagated to the calling environment.

The calling environment can be either a calling **method**, or the **browser**.

function validateName(name){

if(name.Match(/[\$\#]/)){

console.log("Input is invalid");

return false;

}

else {

console.log("Input is valid");

return true;

}

}

function validate(){

validateName("Hello");

}

validate()

JavaScript

Copy

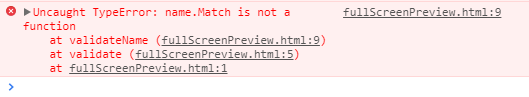
# Exception propagation steps:

* In the above scenario, an error is generated inside validateName() method
* Since validateName() method doesn't handle it, the error is propagated back to its calling environment, which is the validate()
* Since validate() method doesn't handle it, the error is propagated back to its calling environment, which is button click event
* Since the error is not even handled in button click event, it is propagated back to its calling environment i.e. browser

When the browser receives the error, it shows the stack trace in the console and terminates the program. Hence, the error started from validateName(), propagated to validate() and then propagated back to the browser.

Now let us understand how to handle these errors.

### Error object



As we saw in the console, the **Error** object contains three properties:

* **name**: defines name/type of the error. For this example the error name is 'TypeError'.
* **message**: is a short description about the error. In our case, it is 'name.Match is not a function', because Match() function with uppercase 'M' does not exist.
* **stack**: A full stack trace of the error, with error name, error message, file name, method, line information about where the error has occurred. Here, the last three lines starting with "at" is the stack trace.

There can be different values for the ***name***property, which signify different **Error**objects thrown during the execution of JS program.

# Here are some built-in error objects in JS

* **EvalError:**Is an instance of Error which represents than an error occurred regarding the **global function**. Example: **eval()**.
* **InternalError**: Is an instance of Error which represents an internal error in the JavaScript engine.  Example : "**too much recursion**".
* **RangeError**:Is an instance of Error which represents than an error occurred when a numeric variable or parameter is outside of its valid range.
* **ReferenceError**: Is an instance of Error which represents than an error occurred when de-referencing an invalid reference.
* **SyntaxError**: Is an instance of Error which represents than an error occurred while parsing some input in**eval()** or in **JSON.parse()**.
* **TypeError**: Is an instance of Error which represents than an error occurred when a variable or parameter is not of a valid type.
* **URIError**: Is an instance of Error which represents than an error occurred when **encodeURI()** or **decodeURI()** are passed with invalid parameters.

You can explore more about [errors](http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) here.

### try-catch block

Error handling is important, as unhandled errors can lead to **abrupt termination** of the program. These errors can be handled by using **try-catch**block.

function validateName(name) {

try {

if (name.Match(/[\$\#]/)) {// error occurs here

/\* All the below lines of try do not run

as error was thrown in previous line\*/

return false;

}

else {

return true;

}

}

catch (error) {

// code for Handling error

console.log(error.message);

}

}

validateName("Josh")

JavaScript

Copy

The code that can throw an error should be enclosed inside the **try** block. A try block should be immediately followed by a **catch** block.

A catch block is an **error handler** which can handle the error.

The error object thrown from try block will be passed as **parameter** to catch block.

In the above case, when **validateName()** throws an error because of wrong method name, the error object is created and thrown. This error object is caught by the catch block and performs appropriate handling of error, here we have just logged the message present in the error object.

Once the error object has been thrown, the next immediate lines in the try block will **not**be executed.

### Various conditions in a catch block

Now that we have seen different types of Errors, can we handle each of these errors in separate manner? The answer is **no**. This is because JavaScript is **dynamically typed language**, so we cannot specify the different catch blocks for each error instance.

Instead, which we can use some **conditional statements** inside the catch block.

For example:

function validateName(name) {

try {

if (name.Match(/[\$\#]/)) {// error occurs here

/\* All the below lines of try do not run

as error was thrown in previous line\*/

return false;

}

else {

return true;

}

}

catch (error) {

if (error instanceof TypeError)

console.log("Type Error Occurred");

else if (error instanceof RangeError)

console.log("Range Error Occurred");

else if (error instanceof SyntaxError)

console.log("Syntax Error Occurred");

else

console.log("Some other Error Occurred");

}

}

validateName("Josh$");

JavaScript

Copy

Here, once an error is thrown in the try block, it will be handled by the catch block. Inside the catch block, we are checking the type of the error object by using ***instanceof***operator and handling them separately.

### finally block

An error inside a try block causes the rest of the code to be skipped. This might lead to some important parts of the code not being executed.

There may be some important code which must be executed in all the conditions.

For example:

* Closing the database or file connection
* Releasing the memory allocated for objects

Hence, keeping these such code inside the try block cannot guarantee their execution.

In such situations, the **finally**block plays an important role. The **finally**block ensures that the code will be executed, irrespective of whether an error has occurred or not.

Observe how we have implemented try-catch-finally for validateName() function.

function validateName(name) {

try {

if (name.Match(/[\$\#]/)) {// error occurs here

/\* All the below lines of try do not run

as error was thrown in previous line\*/

return false;

}

else {

return true;

}

}

catch (error) {

console.log("Error Occurred");

}

finally{

console.log("Cleaning up resources");

}

}

validateName("josh$");

JavaScript

Copy

**Note**: A try block should be always followed by either a catch block or a finally block or both.

### throw keyword

Apart from the code throwing errors, we can also programmatically create our own errors and throw them to change the flow of execution.

This can be done by creating a new object of Error class and passing our own name and message to it.

var err = new Error(); //You can pass the message or not it is optional

err.name = "InvalidEmailError";

err.message ="Invalid Email";

throw err;

JavaScript

Copy

For example:

function validateName(name) {

try {

if (name.match(/\$/)) {

throw new Error("Name should not contain $");

}

else {

return true;

}

}

catch (error) {

console.log(error.message);

}

finally{

console.log("Cleaning up resources");

}

}

validateName("Hello$")

### Asynchronous in JavaScript

JavaScript is single threaded programming language which means that all code will be executed in sequence. Thus if a code takes 2 seconds to run, the browser will freeze for those 2 seconds. This causes a bad user experience. Thus, we have to execute code asynchronously whenever it takes a longer time to execute.

The DOM API gives us a facility to execute code asynchronously. This API gives us a method called setTimeout(functionName,timeInMilliseconds) which allow us to execute a function after a specific time delay.

Consider the below code:

function display(){

console.log("Lets go to Trip");

};

console.log("Before");

setTimeout(display,3000)

console.log("After");

JavaScript

Copy

**After** will be printed before "Lets go to Trip" as the function display is invoked asynchronously. The console.log("After") does not wait for the previous step to complete.

### AJAX

Our JS code can contact the server and send/fetch data. To perform this operation asynchronously we can use the XMLHttpRequest API.

The JavaScript code can asynchronously connect to a server by using a new XMLHttpRequest() object. This is also called as AJAX.

The four steps for using AJAX are:

1. Create new XMLHttpRequest()
2. Open a URL using the request object
3. Mention what should happen when a response is received
4. Actually send the request

var xhr = new XMLHttpRequest(); // 1.Create request object

xhr.open('GET', url); // 2.Open the URL

xhr.onload = function () { // 3.Mention code to run when response is received

console.log("The response from server is "+xhr.responseText);

};

xhr.send(); // 4.Send the Request

JavaScript

Copy

### Callback

A callback is a function which will get executed automatically **after** some other function gets executed completely. Thus, it is also called as **call-after**.

In fact, we have already been looking at callbacks since we started discussing about setTimeout.

Consider the below code:

function display(){

console.log("Lets go to Trip");

};

setTimeout(display,3000);

JavaScript

Copy

Here, we are not invoking display() anywhere! display is being invoked automatically by the setTimeout. We can see that display() is invoked or called after 3000 milliseconds. Thus here display is called as a callback function. Callback functions are those functions which are passed as a parameter to another function and **invoked after** the other function has completed it's execution.

### Callback

Let us use callback to address the problem of dealing with data in an asynchronous situation. Consider the below code again:

function check(value) {

console.log(value);

}

function getTrip() {

setTimeout(function() {

return "Lets go to Trip";

}, 1500);

}

value = getTrip();

check(value);

// undefined

JavaScript

Copy

This can be modified using callback as shown below:

function check(value){

console.log(value);

};

function getTrip (callback) {

setTimeout(function () {

callback("Lets go to Trip");

}, 1500);

}

getTrip(check);

// Lets go to Trip

JavaScript

Copy

Here we are passing **check** function as a callback to the getTrip() function. Now check() will be invoked inside setTimeout() instead of returning the data. Typically third party libraries expect you to pass a callback and they will invoke your callback after they have completed the task.

### Callback hell

What if a callback has a callback? While doing multiple asynchronous operations, the callback get nested to each other which leads to **callback hell.**

Callback function can not be chained together which leads to nested callback while calling it multiple time which leads to callback hell.

To overcome the problem of callback hell, we use **promise** which we will see next.

### Promise

Just like in real life, a **promise**is an object that may produce a single value some time in the future.

A promise can be:

* **resolved**- promise succeeded
* **rejected** - promise failed
* **pending** - Hasn't fulfilled or rejected yet

Promise can be created using  **new Promise().**Lets understand it with an example.

function getTrip(){

return new Promise(function(resolve){

setTimeout(function() {

resolve("Lets go to Trip");

}, 2000);

});

};

JavaScript

Copy

In the above code, we are not returning the actual data. We are returning a promise of the actual data.

Line2 says we are returning a promise. We can see that while creating a promise, we have to pass a function as a parameter. The code that will be executed in future is placed inside this function.

What is resolve in line4? Resolve is the data what we are promising. The above code says we are promising a string data in future, if everything goes well.

Even in real life, we promise about things that will happen in future. So when we create a promise in JS, we are in effect creating a data that will be available in the future, if everything goes well.

So how can we access this data in future? Well, a promise can be handled using **then**method. The general syntax is:

functionReturningAPromise().then(

function(futureData){

....

})

JavaScript

Copy

For example:

getTrip().then(

function(futureData){

console.log(futureData);

}

);

JavaScript

Copy

The above example will invoke the function inside **then** after 2000 milliseconds as getTrip() resolves a promise only after 2000 ms.

### Resolve and Reject Promise

Resolving a promise is valid only if nothing goes wrong. What if something goes wrong and we are not able to keep the promise? Then we need to reject the promise instead of resolving the promise. So instead of returning the promised data, we can return an error. In any new Promise we can define resolve and reject both.

**Syntax:**

new Promise (resolve, reject) { }

Let's understand it with example.

function getTrip(location) {

return new Promise(function(resolve, reject) {

if (location == "ooty") {

resolve("Trip to " + location);

} else {

reject(Error("Some error occured"));

}

});

}

JavaScript

Copy

Now, let's call the above function. Here, promise will be **resolved**.

getTrip("ooty").then(

function(data) {

console.log(data);

},

function(error) {

console.log(error);

}

);

// Trip to ooty

JavaScript

Copy

In this call, condition will fail and promise will be **rejected**.

getTrip("coorg").then(

function(data) {

console.log(data);

},

function(error) {

console.log(error.message);

}

);

// Error: Some error occurred

### Rejected chained promises

What will happen if any of promises fail in the chained promises?

Promises after the failed promise will not be resolved. It will throw an error and promises in chain will not be executed after it.

function bookFlight(airline) {

return new Promise(function (resolve, reject) {

if (airline == "AirIndia") {

setTimeout(resolve(5600), 2000);

} else {

reject(Error("Flight can not be booked"))

}

})

}

function bookHotel(flightPrice) {

return new Promise(function (resolve) {

setTimeout(resolve(7000 + flightPrice), 1000);

})

}

bookFlight("indigo")

.then(function (flightData) { return bookHotel(flightData) })

.then(function (cumulativeData) { console.log(" Total is " + cumulativeData) })

.catch(e => console.log(e.message))

JavaScript

Copy

The promise returned by the first **then** will throw error.

function bookFlight(){

return new Promise((resolve,reject)=>setTimeout(resolve(1000),15000));

}

function bookHotel(flightCharge){

return new Promise((resolve,reject)=>setTimeout(reject(flightCharge+2000),20000));

}

bookFlight().then((value)=>bookHotel(value)).then().catch((e)=>console.log(e));

### Async/Await

We can simplify chaining of promises using a concept called as async-await.

Consider the below code which performs some asynchronous operations.

function bookFlight() {

return new Promise(function (resolve) {

setTimeout(resolve(5600), 2000);

})

}

function bookHotel(flightPrice) {

return new Promise(function (resolve) {

setTimeout(resolve(7000 + flightPrice), 1000);

})

}

function getTotal(){

bookFlight()

.then(function (flightData) { return bookHotel(flightData) })

.then(function (cumulativeData) { console.log(" Total is " + cumulativeData) })

}

getTotal()

JavaScript

Copy

We can simplify the promise handling using async await. It involves two steps:

* Add **async**keyword in front of getTotal().
* Instead of using .then(), add **await**keyword in front of method invocation and store the returned value.

async function getTotal(){

var flightData=await bookFlight();

var cumulativeData=await bookHotel(flightData);

console.log(" Total is " + cumulativeData) })

}

JavaScript

Copy

We can see how this has simplified promise chaining. Whenever we have await, the code will wait for the promise to be resolved.

**Note**: await can be used only inside async functions.

Creating a test spec

describe('TotalTravelFare calculation Suite:',function(){

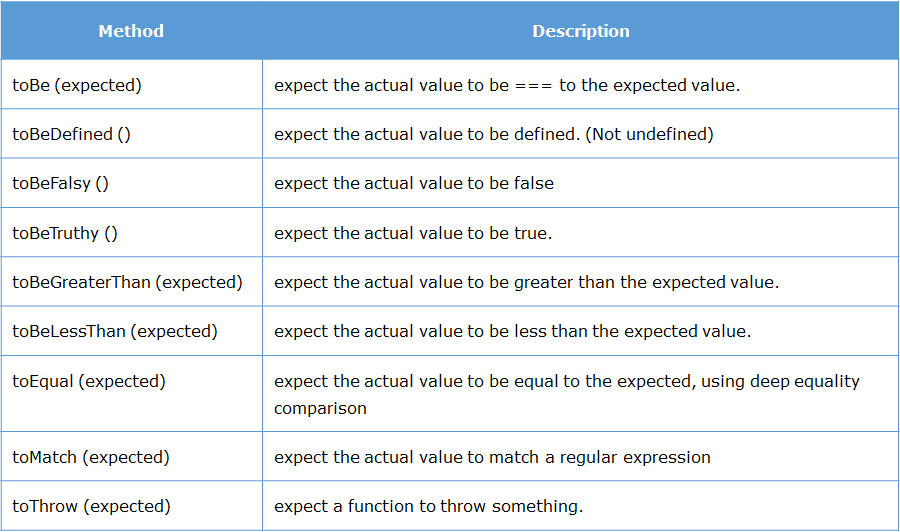
it('Test Case 1: Inputs are correct',function(){

expect(totalTravelFare(1000,20)).toEqual(1200);

});

})

### Matchers



### Importing modules

There are multiple styles of writing imports in ES6 JavaScript. The most common practice is

import { item1,item2,... } from 'filename.js'

For example:

import { TravelFare } from "./travel.js";

JavaScript

Copy

We can also handle multiple imported modules as shown below:

import { totalTravelFare, TravelFare } from "./travel.js";

console.log(totalTravelFare(1000,10));

tf= new TravelFare();

console.log(tf.totalTravelFare(1000,10);

JavaScript

Copy

**Note:**The import and export are part of ES6 and not yet supported in the Node version used. To use this in the browser, we need to add type="module" attribute in the script tag.

For example:

<script type="module" src="travel.js"></script>

### Default export

Apart from exporting , we can also do an default export. The advantage is that while importing, it can be imported with any name. For example:

# Default export

export default class TravelFare {

totalTravelFare(baseFare, taxPercentage){

var finalFare;

finalFare = baseFare\*(1+taxPercentage/100);

return finalFare;

}

}

JavaScript

Copy

# Import

import travel from "./travel.js";

JavaScript

Copy

**Note:**

* There can be only one default export per file
* While importing one should NOT use { }. Else it will cause an error

### Web Storage API

# **Featues of Web Storage:**

* Large amount of data can be stored – Atleast 5MB data can be stored in client machine.
* Reduced network overhead – Stored data is never send back to web server. Hence, there are no additional HTTP request-response cycle.
* Secure – Each domain is given a part of memory in client's machine. Webpages from same domain can share data.  One domain cannot override/access data of another domain. Hence, it is ensures data security.

# 

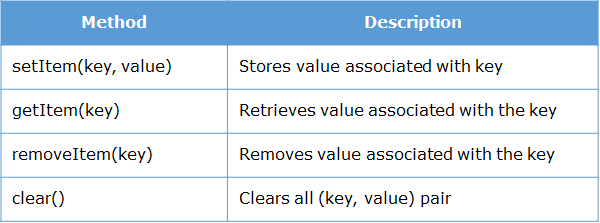
# How to use Web Storage:

We can store the data using web storage in two ways:

1. For particular session i.e. **session**storage: For session storage, sessionStorage object is used
2. Across sessions i.e. **local**storage: For local storage, localStorage object is used

Both these objects are created by JS runtime engine of browser.

Web storage API provides following four methods for managing data:



Data is stored in the form of **(key, value)** pair and both key and value are stored as **String**.

### Create a notification

In order to create notification, we need an object of Notification which can be created as shown below:

new Notification(title,options)

JavaScript

Copy

Finally we can create a notify() function for desktop notifications combining both step 1 and step 2 as shown below:

function notify(){

Notification.requestPermission();

if(Notification.permission === "default"){

alert("Please grant permission");

}

else {

var notify = new Notification("New Mail",{body:"You have 1 unread email"});

}

}

JavaScript

Copy

# Properties of Notification object:

**body**:

* Defines notification message

**permission**:

* It is a static property of Notification interface, which represent current permission to display notifications.
* Possible values for permission are denied,granted,default  ( the user choice is unknown so that browser will act as if the value is denied )

As a web developer, whenever you are developing a web-application that is supposed to notify user upon receiving updated data, you should use Notification API.

Questions:

1.(true + false) > 2 + true;

Ans. i) true+false 🡪1

ii)true+true 🡪2

iii)false+false ->0

2. function bar() {

return foo;

foo = 10;

function foo() {}

var foo = '11';

}

alert(typeof bar());

Ans. function, Functions are hoisted before variable declarations

3. "1" - - "1";

Ans. 2

### null

// This stands since the beginning of JavaScript

typeof null === 'object';

In the first implementation of JavaScript, JavaScript values were represented as a type tag and a value. The type tag for objects was 0. null was represented as the NULL pointer (0x00 in most platforms). Consequently, null had 0 as type tag, hence the "object" typeof return value. ([reference](http://www.2ality.com/2013/10/typeof-null.html))

A fix was proposed for ECMAScript (via an opt-in), but [was rejected](https://web.archive.org/web/20160331031419/http:/wiki.ecmascript.org:80/doku.php?id=harmony:typeof_null). It would have resulted in typeof null === 'null'.

What is Function's Data Type : function or object?

typeof returns the type of what ever is passed to it. A function is an object ((function () {}) instanceof Object === true), but the typeof function is defined to return "function" when an Object implements [[Call]] in ECMA-262 is supplied to it.

Functions are objects, but typeof treats them as a special case.