'setInterval' vs 'setTimeout'

setTimeout(expression, timeout); runs the code/function once after the timeout.

setInterval(expression, timeout); runs the code/function in intervals, with the length of the timeout between them.

Example:

var intervalID = setInterval(alert, 1000); // Will alert every second.

// clearInterval(intervalID); // Will clear the timer.

setTimeout(alert, 1000); // Will alert once, after a second.

Array.splice() vs Array.slice()

1. The **splice()**method returns the removed item(s) in an array and **slice()**method returns the selected element(s) in an array, as a new array object.

2. The **splice()** method changes the original array and **slice()** method doesn’t change the original array.

3. The **splice()** method can take n number of arguments:

**Argument 1:**Index, Required. An integer that specifies at what position to add /remove items, Use negative values to specify the position from the end of the array.

**Argument 2:**Optional. The number of items to be removed. If set to 0(zero), no items will be removed. And if not passed, all item(s) from provided index will be removed.

**Argument 3…n:**Optional. The new item(s) to be added to the array.

|  |  |
| --- | --- |
|  | var array=[1,2,3,4,5];  console.log(array.splice(2));  // shows [3, 4, 5], returned removed item(s) as a new array object.    console.log(array);  // shows [1, 2], original array altered.    var array2=[6,7,8,9,0];  console.log(array2.splice(2,1));  // shows [8]    console.log(array2.splice(2,0));  //shows [] , as no item(s) removed.    console.log(array2);  // shows [6,7,9,0]    var array3=[11,12,13,14,15];  console.log(array3.splice(2,1,"Hello","World"));  // shows [13]    console.log(array3);  // shows [11, 12, "Hello", "World", 14, 15]               -5 -4 -3 -2 -1              |  |  |  |  |  var array4=[16,17,18,19,20];               |  |  |  |  |               0  1  2  3  4    console.log(array4.splice(-2,1,"me"));  // shows  [19]    console.log(array4);  // shows [16, 17, 18, "me", 20] |

**If Argument(1) is NaN, it is treated as if it were 0.**

|  |  |
| --- | --- |
|  | var array5=[21,22,23,24,25];  console.log(array5.splice(NaN,4,"NaN is Treated as 0"));  // shows [21,22,23,24]    console.log(array5);  // shows ["NaN is Treated as 0",25] |

**If Argument(2) is less than 0 or equal to NaN, it is treated as if it were 0.**

|  |  |
| --- | --- |
|  | var array6=[26,27,28,29,30];  console.log(array6.splice(2,-5,"Hello"));  // shows []    console.log(array6);  // shows [26,27,"Hello",28,29,30]    console.log(array6.splice(3,NaN,"World"));  // shows []    console.log(array6);  // shows [26,27,"Hello","World",28,29,30] |

**If Argument(1) or Argument(2) is greater than Array’s length, either argument will use the Array’s length.k**

|  |  |
| --- | --- |
|  | var array7=[31,32,33,34,35];  console.log(array7.splice(23,3,"Add Me"));  // shows []    console.log(array7);  // shows [31,32,33,34,35,"Add Me"]    console.log(array7.splice(2,34,"Add Me Too"));  // shows [33,34,35,"Add Me"]    console.log(array7);  // shows [31,32,"Add Me Too"] |

4. The **slice()** method can take 2 arguments:

**Argument 1:**Required. An integer that specifies where to start the selection (The first element has an index of 0). Use negative numbers to select from the end of an array.

**Argument 2:**Optional. An integer that specifies where to end the selection. If omitted, all elements from the start position and to the end of the array will be selected. Use negative numbers to select from the end of an array.

|  |  |
| --- | --- |
|  | var array=[1,2,3,4,5]  console.log(array.slice(2));  // shows [3, 4, 5], returned selected element(s).    console.log(array.slice(-2));  // shows [4, 5], returned selected element(s).  console.log(array);  // shows [1, 2, 3, 4, 5], original array remains intact.    var array2=[6,7,8,9,0];  console.log(array2.slice(2,4));  // shows [8, 9]    console.log(array2.slice(-2,4));  // shows [9]    console.log(array2.slice(-3,-1));  // shows [8, 9]    console.log(array2);  // shows [6, 7, 8, 9, 0] |

**If either argument is NaN, it is treated as if it were 0.**

|  |  |
| --- | --- |
|  | var array3=[11,12,13,14,15];  console.log(array3.slice(NaN,NaN));  // shows []    console.log(array3.slice(NaN,4));  // shows [11,12,13,14]    console.log(array3);  // shows [11,12,13,14,15] |

**If either argument is greater than the Array’s length, either argument will use the Array’s length**

|  |  |
| --- | --- |
|  | var array4=[16,17,18,19,20];  console.log(array4.slice(23,24));  // shows []    console.log(array4.slice(23,2));  // shows []    console.log(array4.slice(2,23));  // shows [18,19,20]    console.log(array4);  // shows [16,17,18,19,20] |

# escape()

The escape() function computes a new string in which certain characters have been replaced by a hexadecimal escape sequence

The escape() function encodes a string.

This function makes a string portable, so it can be transmitted across any network to any computer that supports ASCII characters.

This function encodes special characters, with the exception of: \* @ - \_ + . /

## Syntax

escape(str)

### **Parameters**

**str**

A string to be encoded.

### **Return value**

A new string in which certain characters have been escaped.

Why innerHtml is not used?

Every time innerHTML is set, the HTML has to be parsed, a DOM constructed, and inserted into the document. This takes time.

For example, if elm.innerHTML has thousands of divs, tables, lists, images, etc, then calling .innerHTML += ... is going to cause the parser to re-parse *all that stuff* over again. This could also break references to already constructed DOM elements and cause other chaos. In reality, all you want to do is append a single new element to the end.

It's better to just call appendChild:

var newElement = document.createElement('div');

newElement.innerHTML = '<div>Hello World!</div>';

elm.appendChild(newElement);​​​​​​​​​​​​​​​​

What shall be the JavaScript code to identify the browser on which the page is opened?

You can either use this JavaScript library - [https://raw.githubusercontent.co...](https://raw.githubusercontent.com/darcyclarke/Detect.js/master/detect.js) and run the following:

**var** pageOpen = detect.parse(navigator.userAgent);

console.log("Visitor browser - " + pageOpen.browser.family + "browser version - " + pageOpen.browser.version);

OR

just run the following:

**function** getBrowserName(){

**var** navAgent = navigator.userAgent;

**var** browserName;

**if** ((versionOffset=navAgent.indexOf("Opera"))!=-1) {

browserName = "Opera";

}

**else** **if** ((versionOffset=navAgent.indexOf("MSIE"))!=-1) {

browserName = "Microsoft Internet Explorer";

}

**else** **if** ((versionOffset=navAgent.indexOf("Chrome"))!=-1) {

browserName = "Chrome";

}

**else** **if** ((versionOffset=navAgent.indexOf("Safari"))!=-1) {

browserName = "Safari";

}

**else** **if** ((versionOffset=navAgent.indexOf("Firefox"))!=-1) {

browserName = "Firefox";

}

**else** **if** ( (nameOffset=navAgent.lastIndexOf(' ')+1) < (versionOffset=navAgent.lastIndexOf('/')) ) {

browserName = navAgent.substring(nameOffset,versionOffset);

**if** (browserName.toLowerCase()==browserName.toUpperCase()) {

browserName = navigator.appName;

}

}

**return** browserName;

}

console.log('Browser name = ' + getBrowserName());

# Javascript Array Methods: Unshift(), Shift(), Push(), And Pop()

### Push() Method

The push() method can append one or more elements to the end of an array. This alters the array on which the method was called.

var data = [ "X" ];

data.push( "A" );

data.push( "B", "C" );

console.log( data );

When we run the above code, we get the following console output:

*["X", "A", "B", "C"]*

### Pop() Method

The pop() method pulls the last element off of the given array and returns it. This alters the array on which the method was called.

var data = [ "A", "B", "C" ];

console.log( data.pop() );

console.log( data );

When we run the above code, we get the following console output:

*C  
["A", "B"]*

### Unshift() Method

The unshift() method is like the push() method, only it works at the beginning of the array. The unshift() method can prepend one or more elements to the beginning of an array. This alters the array on which the method was called.

var data = [ "X" ];

data.unshift( "A" );

data.unshift( "B", "C" );

console.log( data );

When we run the above code, we get the following console output:

["B", "C", "A", "X"]

### Shift() Method

The shift() method is like the pop() method, only it works at the beginning of the array. The shift() method pulls the first element off of the given array and returns it. This alters the array on which the method was called.

var data = [ "A", "B", "C" ];

console.log( data.shift() );

console.log( data );

If you call shift() on an empty array, it returns an undefined value.

# map, filter and reduce

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

Sometimes we have an array of objects that we wish to modify/add properties of each object, other times we might have an array of strings that we which to turn all of them lower case. In reality there might be countless situations in which **map** is your savior and it’s really simple to use.

# 

# As you can see, ****map**** receives a callback as an argument. That callback is then given the ****current value**** of the iteration, the ****index**** of the iteration and the ****original array**** from which map was called. There is also an optional second argument for map (after the callback) that is the value to use as “****this”**** inside the callback.

# 

# 

# 

# [Difference between .forEach() and .map()](https://stackoverflow.com/questions/34426458/javascript-difference-between-foreach-and-map)

They are not one and the same. Let me explain the difference.

forEach: This iterates over a list and applies some operation with side effects to each list member (example: saving every list item to the database)

Returns undefined

map: This iterates over a list, transforms each member of that list, and returns another list of the same size with the transformed members (example: transforming list of strings to uppercase)

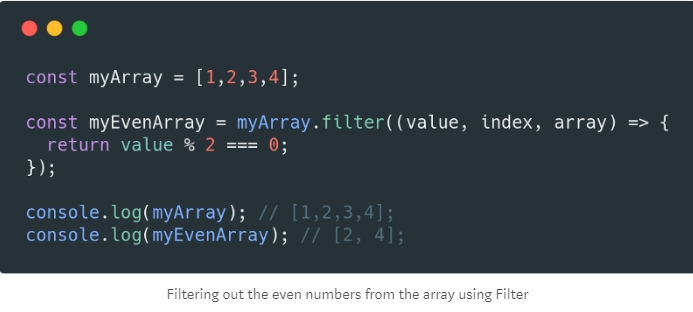
Returns new array with tranformed elements leaving back original array unchanged

### [Filter](https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

I’m pretty confident that in your entire path as a programmer you’ve had to filter some items out of an existing array, as it is one of the most common exercises. This used to be quite annoying to do yourself but now there is no need to sweat, **filter** is here to help!

#### How to use it?

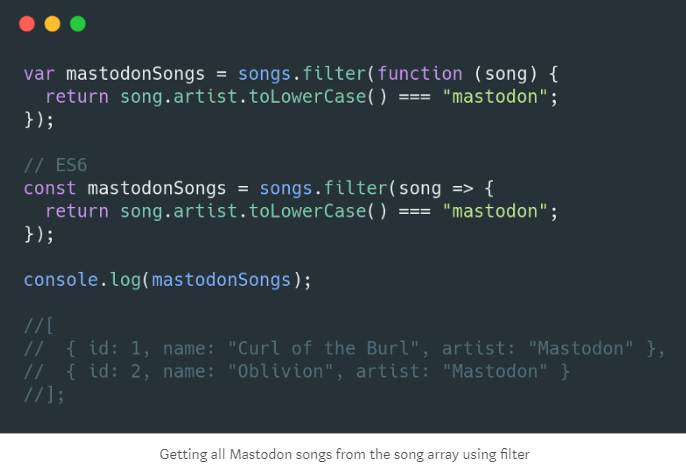
Pretty similar to **map**. If you know **map**, you know **filter**!



**Filter** receives the same arguments as **map**, and works very similarly. The only difference is that the callback needs to return either**true** or **false**. If it returns **true**then the array keeps that element and if it returns **false** the element is filtered out.







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

Last but not least, **reduce**takes an array and reduces it into a single value. For example, with an array of numbers you can easily find the average of all values.

#### How to use it?

It is similar to both **map** and **filter** but, it differs in the callback arguments. The callback now receives the **accumulator** (it accumulates all the return values. Its value is the accumulation of the previous returned accumulations), the current value, the current index and finally the whole array.

# 

# 

# forEach()

# The forEach() prototype\* method, introduced in ES5, executes a provided callback function (a function passed to another function as an argument) once for each value in Array, Map (made up of key-value pairs), Set, NodeList (an array-like object often returned by Document.querySelectorAll()), or HTMLCollection (an array-like object often returned by Document.getElementsByClassName()) objects, in ascending order.

# myArray.forEach(function (value) {

# // Do stuff

# });

# It takes a function callback as a parameter, with the current value being processed in the array as its argument. (There are also three optional arguments: the current index value, the array forEach() is being applied to, and a value to use as this [the reference Object].)

# It’s somewhat more succinct than the classic for loop (which has been around for much longer). However, you can’t use the statements break to exit the loop, continue to skip an iteration, or return to return a value (and exit the loop). It returns undefined.

# To use forEach() to loop over a NodeList or HTMLCollection object, note that you’ll need to first convert it to an Array object using Array.from().

# 

# O/P:

# 

**for..in**

# Use for…in to iterate over the properties of an object (the object keys):

# 

You can also use for…in to iterate over the index values of an iterable like an array or a string:

# 

# for..of

# The for...of statement creates a loop iterating over [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#The_iterable_protocol), including: built-in [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), Array-like objects (e.g., [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) or [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList)), [TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray), [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

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

let iterable = [10, 20, 30];

for (let value of iterable) {

value += 1;

console.log(value);

}

// 11

// 21

// 31

### **Iterating over a**[**String**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

let iterable = 'boo';

for (let value of iterable) {

console.log(value);

}

// "b"

// "o"

// "o"

### **Iterating over a**[**Map**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map)

let iterable = new Map([['a', 1], ['b', 2], ['c', 3]]);

for (let entry of iterable) {

console.log(entry);

}

// ['a', 1]

// ['b', 2]

// ['c', 3]

for (let [key, value] of iterable) {

console.log(value);

}

// 1

// 2

// 3

### **Iterating over a DOM collection**

Iterating over DOM collections like [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList): the following example adds a read class to paragraphs that are direct descendants of an article:

// Note: This will only work in platforms that have

// implemented NodeList.prototype[Symbol.iterator]

let articleParagraphs = document.querySelectorAll('article > p');

for (let paragraph of articleParagraphs) {

paragraph.classList.add('read');

}

## A Comparison

|  | **for..in** | **for..of** |
| --- | --- | --- |
| Applies to | Enumerable Properties | Iterable Collections |
| Use with Objects? | Yes | No |
| Use with Arrays? | Yes, but not advised | Yes |
| Use with Strings? | Yes, but not advised | Yes |

The [for..of loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) only supports iterable objects like arrays, not objects.

To iterate over the values of an object, use:

for (var key in test) {

var item = test[key];

}

# Rest parameters and spread operator

## **[Rest parameters ...](https://javascript.info/rest-parameters-spread-operator" \l "rest-parameters)**

A function can be called with any number of arguments, no matter how it is defined.

Like here:

function sum(a, b) {

return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted.

The rest parameters can be mentioned in a function definition with three dots .... They literally mean “gather the remaining parameters into an array”.

For instance, to gather all arguments into array args:

function sumAll(...args) { // args is the name for the array

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

We can choose to get the first parameters as variables, and gather only the rest.

Here the first two arguments go into variables and the rest go into titles array:

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Julius Caesar

// the rest go into titles array

// i.e. titles = ["Consul", "Imperator"]

alert( titles[0] ); // Consul

alert( titles[1] ); // Imperator

alert( titles.length ); // 2

}

showName("Julius", "Caesar", "Consul", "Imperator");

**The rest parameters must be at the end**

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!

// error

}

The ...rest must always be last.

**spread operator**

It looks similar to rest parameters, also using ..., but does quite the opposite.

When ...arr is used in the function call, it “expands” an iterable object arr into the list of arguments.

For Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)

We also can pass multiple iterables this way:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8

We can even combine the spread operator with normal values:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Also, the spread operator can be used to merge arrays:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)

For instance, here we use the spread operator to turn the string into array of characters:

let str = "Hello";

alert( [...str] ); // H,e,l,l,o

## **[Summary](https://javascript.info/rest-parameters-spread-operator" \l "summary)**

When we see "..." in the code, it is either rest parameters or the spread operator.

There’s an easy way to distinguish between them:

* When ... is at the end of function parameters, it’s “rest parameters” and gathers the rest of the list of arguments into an array.
* When ... occurs in a function call or alike, it’s called a “spread operator” and expands an array into a list.

Use patterns:

* Rest parameters are used to create functions that accept any number of arguments.
* The spread operator is used to pass an array to functions that normally require a list of many arguments.

# Promise

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

The constructor syntax for a promise object is:

let promise = new Promise(function(resolve, reject) {

// executor (the producing code, "singer")

});

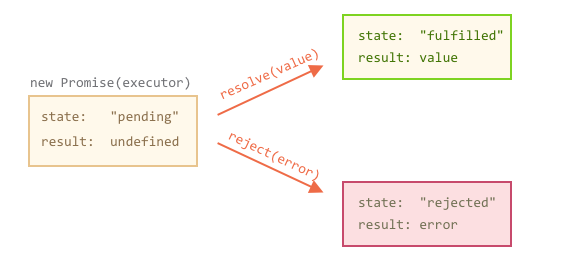
The function passed to new Promise is called the *executor*. When the promise is created, this executor function runs automatically. It contains the producing code, that should eventually produce a result. In terms of the analogy above: the executor is the “singer”.

The resulting promise object has internal properties:

* state — initially “pending”, then changes to either “fulfilled” or “rejected”,
* result — an arbitrary value of your choosing, initially undefined.

When the executor finishes the job, it should call one of the functions that it gets as arguments:

* resolve(value) — to indicate that the job finished successfully:
  + sets state to "fulfilled",
  + sets result to value.
* reject(error) — to indicate that an error occurred:
  + sets state to "rejected",
  + sets result to error.



**example of a successful job completion, a “fulfilled promise”:**

let promise = new Promise(function(resolve, reject) {

// the function is executed automatically when the promise is constructed

// after 1 second signal that the job is done with the result "done"

setTimeout(() => resolve("done"), 1000);

});

**example of the executor rejecting the promise with an error:**

let promise = new Promise(function(resolve, reject) {

// after 1 second signal that the job is finished with an error

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

**There can be only a single result or an error**

The executor should call only one resolve or one reject. The promise’s state change is final.

All further calls of resolve and reject are ignored:

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // ignored

setTimeout(() => resolve("…")); // ignored});

The idea is that a job done by the executor may have only one result or an error.

Also, resolve/reject expect only one argument (or none) and will ignore additional arguments.

**The state and result are internal**

The properties state and result of the Promise object are internal. We can’t directly access them from our “consuming code”. We can use the methods .then/.catch/.finally for that. They are described below.

**Consumers: then, catch, finally**

### **[then](https://javascript.info/promise-basics" \l "then)**

The most important, fundamental one is .then.

The syntax is:

promise.then(

function(result) { /\* handle a successful result \*/ },

function(error) { /\* handle an error \*/ }

);

The first argument of .then is a function that:

1. runs when the Promise is resolved, and
2. receives the result.

The second argument of .then is a function that:

1. runs when the Promise is rejected, and
2. receives the error.

For instance, here’s a reaction to a successfuly resolved promise:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve runs the first function in .then

promise.then(

result => alert(result), // shows "done!" after 1 second

error => alert(error) // doesn't run

);

If we’re interested only in successful completions, then we can provide only one function argument to .then:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // shows "done!" after 1 second

### **[catch](https://javascript.info/promise-basics" \l "catch)**

If we’re interested only in errors, then we can use null as the first argument: .then(null, errorHandlingFunction). Or we can use .catch(errorHandlingFunction), which is exactly the same:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// .catch(f) is the same as promise.then(null, f)

promise.catch(alert); // shows "Error: Whoops!" after 1 second

The call .catch(f) is a complete analog of .then(null, f), it’s just a shorthand.

### **[finally](https://javascript.info/promise-basics" \l "finally)**

Just like there’s a finally clause in a regular try {...} catch {...}, there’s finally in promises.

The call .finally(f) is similar to .then(f, f) in the sense that it always runs when the promise is settled: be it resolve or reject.

finally is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed any more, no matter what the outcome is.

Like this:

new Promise((resolve, reject) => {

/\* do something that takes time, and then call resolve/reject \*/

})

// runs when the promise is settled, doesn't matter successfully or not

.finally(() => stop loading indicator)

.then(result => show result, err => show error)

# Prototypal inheritance

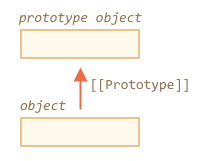
## **[[[Prototype]]](https://javascript.info/prototype-inheritance" \l "prototype)**

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.

Nearly all objects in JavaScript are instances of [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) which sits on the top of a prototype chain.

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of *method overriding*).

When an inherited function is executed, the value of [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) points to the inheriting object, not to the prototype object where the function is an own property.



The property [[Prototype]] is internal and hidden, but there are many ways to set it.

One of them is to use \_\_proto\_\_, like this:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

## Overview — an example of ES6 class syntax and ES5 equivalent

**class** Vehicle {  
   
 **constructor** (name, type) {  
 this.name = name;  
 this.type = type;  
 }  
   
 getName () {  
 return this.name;  
 }  
   
 getType () {  
 return this.type;  
 }  
   
}

let car = new Vehicle('Tesla', 'car');  
console.log(car.getName()); // Tesla  
console.log(car.getType()); // car

ES5 equivalent could be something like this:

function Vehicle (name, type) {  
 this.name = name;  
 this.type = type;  
};  
   
Vehicle.prototype.getName = function getName () {  
 return this.name;  
};  
   
Vehicle.prototype.getType = function getType () {  
 return this.type;  
};

var car = new Vehicle('Tesla', 'car');  
console.log(car.getName()); // Tesla  
console.log(car.getType()); // car

## inheritance

..to it and start from ES5 example:

function Vehicle (name, type) {  
 this.name = name;  
 this.type = type;  
};  
   
Vehicle.prototype.getName = function getName () {  
 return this.name;  
};  
   
Vehicle.prototype.getType = function getType () {  
 return this.type;  
};

function Car (name) {  
 Vehicle.call(this, name, ‘car’);  
}

Car.prototype = Object.create(Vehicle.prototype);  
Car.prototype.constructor = Car;  
Car.parent = Vehicle.prototype;  
Car.prototype.getName = function () {  
 return 'It is a car: '+ this.name;  
};

var car = new Car('Tesla');  
console.log(car.getName()); // It is a car: Tesla  
console.log(car.getType()); // car

And now look at the ES6 version:

class Vehicle {  
   
 constructor (name, type) {  
 this.name = name;  
 this.type = type;  
 }  
   
 getName () {  
 return this.name;  
 }  
   
 getType () {  
 return this.type;  
 }  
   
}

class Car **extends** Vehicle {  
   
 constructor (name) {  
 **super**(name, 'car');  
 }  
   
 getName () {  
 return 'It is a car: ' + **super**.getName();  
 }  
   
}

let car = new Car('Tesla');  
console.log(car.getName()); // It is a car: Tesla  
console.log(car.getType()); // car

We see how easy is to implement inheritance with ES6. It’s finally looking like in other OO programming languages. We use **extends** to inherit from another class and the **super** keyword to call the parent class (function). Moreover, **getName()** method was overridden in subclass **Car**.

**\_\_proto\_\_ is a historical getter/setter for [[Prototype]]**

Please note that \_\_proto\_\_ is *not the same* as [[Prototype]]. That’s a getter/setter for it.

It exists for historical reasons, in modern language it is replaced with functions Object.getPrototypeOf/Object.setPrototypeOf that also get/set the prototype. We’ll study the reasons for that and these functions later.

By the specification, \_\_proto\_\_ must only be supported by browsers, but in fact all environments including server-side support it. For now, as \_\_proto\_\_ notation is a little bit more intuitively obvious, we’ll use it in the examples.

If we look for a property in rabbit, and it’s missing, JavaScript automatically takes it from animal.

For instance:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal; // (\*)

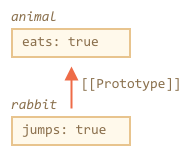
// we can find both properties in rabbit now:

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

Here the line (\*) sets animal to be a prototype of rabbit.

Then, when alert tries to read property rabbit.eats (\*\*), it’s not in rabbit, so JavaScript follows the [[Prototype]] reference and finds it in animal (look from the bottom up):



Here we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal".

So if animal has a lot of useful properties and methods, then they become automatically available in rabbit. Such properties are called “inherited”.

If we have a method in animal, it can be called on rabbit:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

// walk is taken from the prototype

rabbit.walk(); // Animal walk

There are actually only two limitations:

1. The references can’t go in circles. JavaScript will throw an error if we try to assign \_\_proto\_\_ in a circle.
2. The value of \_\_proto\_\_ can be either an object or null, other types (like primitives) are ignored.

Also it may be obvious, but still: there can be only one [[Prototype]]. An object may not inherit from two others.

## [**Summary**](https://javascript.info/prototype-inheritance#summary)

* In JavaScript, all objects have a hidden [[Prototype]] property that’s either another object or null.
* We can use obj.\_\_proto\_\_ to access it (a historical getter/setter, there are other ways, to be covered soon).
* The object referenced by [[Prototype]] is called a “prototype”.
* If we want to read a property of obj or call a method, and it doesn’t exist, then JavaScript tries to find it in the prototype. Write/delete operations work directly on the object, they don’t use the prototype (unless the property is actually a setter).
* If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.

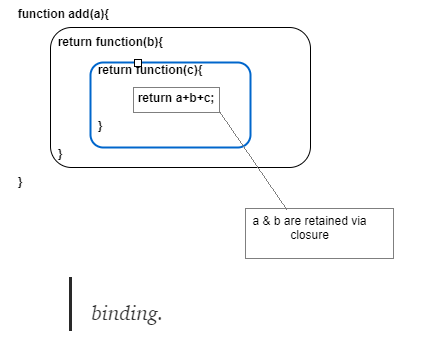
# Currying in JavaScript

Currying is a technique of evaluating function with multiple arguments, into sequence of function with single argument.

In other words, when a function, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth, until all arguments have been fulfilled.

That is, when we turn a function call add(1,2,3) into add(1)(2)(3) . By using this technique, the little piece can be configured and reused with ease.

Currying works by natural closure. The closure created by the nested functions to retain access to each of the arguments. So inner function have access to all arguments.



# [How can I make var a = add(2)(3); //5 work?](https://stackoverflow.com/questions/2272902/how-can-i-make-var-a-add23-5-work)

var add = function(x) {

return function(y) { return x + y; };

}

**OR**

const add = x => y => x + y;

# Array​.from()

Array.from(arrayLike[, mapFn[, thisArg]])

### **Parameters**

**arrayLike**

An array-like or iterable object to convert to an array.

**mapFn**Optional

Map function to call on every element of the array.

**thisArg**Optional

Value to use as this when executing mapFn.

### **Return value**

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

**Description:**

Array.from() lets you create Arrays from:

* array-like objects (objects with a length property and indexed elements) or
* [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) (objects where you can get its elements, such as [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)).

Array.from() has an optional parameter mapFn, which allows you to execute a [map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) function on each element of the array (or subclass object) that is being created. More clearly,Array.from(obj, mapFn, thisArg) has the same result as Array.from(obj).map(mapFn, thisArg), except that it does not create an intermediate array.

With the aid of *Array.from* you can get an *Array*instance from:

* array-like objects (objects which has a *length*property)

**Array**-**Like Objects**. Some **objects** in **JavaScript** look **like** an **array**, but they aren't one. That usually means that they have indexed access and a length property, but none of the **array** methods. Examples include the special variable arguments, DOM node lists, and strings

* iterable objects (objects which has an iterator).

In order to be **iterable**, an object must implement the **@@iterator** method, meaning that the object (or one of the objects up its [prototype chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)) must have a property with a **@@iterator**key which is available via constant [Symbol.iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator):

### **Array from a String**

Array.from('foo');

// [ "f", "o", "o" ]

# Promise​.all()

The Promise.all() method returns a single Promise that resolves when all of the promises passed as an iterable have resolved or when the iterable contains no promises. It rejects with the reason of the first promise that rejects.

var promise1 = Promise.resolve(3);

var promise2 = 42;

var promise3 = new Promise(function(resolve, reject) {

setTimeout(resolve, 100, 'foo');

});

Promise.all([promise1, promise2, promise3]).then(function(values) {

console.log(values); });

**Difference between null, undefined and not defined in javascript**

**Undefined:- typeof(undefined): "undefined"**

There are certain cases when undefined value is returned in javascript as follows:-

1)Whenever we declare a variable without assigning any value to it, javascript implicitly assigns its value as undefined.

let name;  
console.log(name); //undefined

2) When value is not assigned in array or object

let numArray = [1,2,,4];  
console.log(numArray);   
//[1, 2, empty, 4]

typeof(numArray[2])  
//"undefined"

3) When functions don’t have a return statement but are called for assigning a value to a variable.

let add = (a,b) => {  
 let c = a+b;  
 //return c;  
}

let sum = add(2,3);

console.log(sum);   
//Output: undefined

**Null:-** **typeof(null): "object"**

null is a reserved keyword in javascript. We can assign a null value to a variable explicitly using the keyword null. Null essentially represents a non-existent or an empty value i.e. we explicitly tell javascript interpreter that the variable has no value.

let life = null;  
console.log(life); //null

**Not defined:-**

A not defined is a variable which is not declared at a given point of time with declaration keyword like var, let or const.

console.log(a);  
var a = 5;  
//Output:- undefined

While if we don’t use var keyword above the output will be:-

console.log(b);  
b = 5;  
//Output:- "ReferenceError: b is not defined

ReferenceError: "x" is not defined

## What went wrong?

There is a non-existent variable referenced somewhere. This variable needs to be declared, or you need make sure it is available in your current script or [scope](https://developer.mozilla.org/en-US/docs/Glossary/scope).

### **Variable not declared**

foo.substring(1); // ReferenceError: foo is not defined

The "foo" variable isn't defined anywhere. It needs to be some string, so that the [String.prototype.substring()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring) method will work.

var foo = 'bar';

foo.substring(1); // "ar"

### **Wrong scope**

A variable needs to be available in the current context of execution. Variables defined inside a [function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions) cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function

function numbers() {

var num1 = 2,

num2 = 3;

return num1 + num2;

}

console.log(num1); // ReferenceError num1 is not defined.

However, a function can access all variables and functions defined inside the scope in which it is defined. In other words, a function defined in the global scope can access all variables defined in the global scope.

var num1 = 2,

num2 = 3;

function numbers() {

return num1 + num2;

}

console.log(numbers()); // 5

Differences:-

The type of null and undefined are different as follows:

console.log(typeof(undefined)); //"undefined"  
console.log(typeof(null)); //"object"

null is an object with a valid non-existent value and it is non-mutable while object type of an undefined value is in itself undefined.

Also any arithmetic operation with null value will result in integer value while any arithmetic operation with undefined with result in value of variable being changed to NaN.

let a = 7 + null;  
console.log(a); // 7

let b = 7 \* null;  
console.log(b); // 0

let c = 8 + undefined;  
console.log(c); //NaN

let d = 8 \* undefined;  
console.log(d); //NaN

## **Toggle Class**

##### **Step 1) Add HTML:**

<button onclick="myFunction()">Try it</button>  
  
<div id="myDIV">  
  This is a DIV element.  
</div>

##### **Step 3) Add JavaScript:**

Get the <div> element with id="myDIV" and toggle between the "mystyle" class:

function myFunction() {  
  var element = document.getElementById("myDIV");  
  element.classList.toggle("mystyle");  
}

**Add a Class**

##### **Step 1) Add HTML:**

<button onclick="myFunction()">Try it</button>  
  
<div id="myDIV">  
  This is a DIV element.  
</div>

##### **Step 3) Add JavaScript:**

Get the <div> element with id="myDIV" and add the "mystyle" class to it:

function myFunction() {  
  var element = document.getElementById("myDIV");  
  element.classList.add("mystyle");  
}

# [Is it possible to divide a number with string in javascript?](https://stackoverflow.com/questions/26277979/is-it-possible-to-divide-a-number-with-string-in-javascript)

Unary operators will attempt to convert the string to a number. It will only produce NaN if the string cannot by converted to a number.

You'll notice the same if you try the following, which will convert from hex to 10:

## [What is an unary operator?](https://scotch.io/tutorials/javascript-unary-operators-simple-and-useful#toc-what-is-an-unary-operator-)

A unary operator is one that takes a single operand/argument and performs an operation.

A unary operation is an operation with only one operand. This operand comes either before or after the operator. Unary operators are more efficient than standard JavaScript function calls. Additionally, unary operators can not be overridden, therefore their functionality is guaranteed.

## [Unary plus (+)](https://scotch.io/tutorials/javascript-unary-operators-simple-and-useful#toc-unary-plus-)

This operator precedes the operand and tries to convert it to a number.

It can convert all string representations of numbers, boolean values(true and false) and null to numbers. Numbers will include both integers, floats, hexadecimal, scientific (exponent) notation and Infinity.

If the operand cannot be converted into a number, the unary plus operator will return **NaN**.

+3 // returns 3

+'-3' // returns -3

+'3.14' // returns 3.14

+'3' // returns 3

+'0xFF' // returns 255

+true // returns 1

+'123e-5' // returns 0.00123

+false // returns 0

# JavaScript String repeat() method

var str = "Hello world!";  
str.repeat(2);

Output: Hello world!Hello world!

**Array destructuring**

### **Basic variable assignment**

var foo = ['one', 'two', 'three'];

var [one, two, three] = foo;

console.log(one); // "one"

console.log(two); // "two"

console.log(three); // "three"

### [**Parsing an array returned from a function**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Parsing_an_array_returned_from_a_function)

function f() {

return [1, 2];

}

var a, b;

[a, b] = f();

console.log(a); // 1, console.log(b); // 2

**Object destructuring**

var o = {p: 42, q: true};

var {p, q} = o;

console.log(p); // 42

console.log(q); // true

### **Assignment without declaration**

var a, b;

({a, b} = {a: 1, b: 2});

**Notes**: The parentheses ( ... ) around the assignment statement are required when using object literal destructuring assignment without a declaration.

{a, b} = {a: 1, b: 2} is not valid stand-alone syntax, as the {a, b} on the left-hand side is considered a block and not an object literal.

However, ({a, b} = {a: 1, b: 2}) is valid, as is var {a, b} = {a: 1, b: 2}

Your ( ... ) expression needs to be preceded by a semicolon or it may be used to execute a function on the previous line.

### **Assigning to new variable names**

var o = {p: 42, q: true};

var {p: foo, q: bar} = o;

console.log(foo); // 42

console.log(bar); // true

# Async/await

# [Async functions](https://javascript.info/async-await" \l "async-functions)

Let’s start with the async keyword. It can be placed before a function, like this:

async function f() {

return 1;

}

The word “async” before a function means one simple thing: a function always returns a promise. Even If a function actually returns a non-promise value, prepending the function definition with the “async” keyword directs JavaScript to automatically wrap that value in a resolved promise.

For instance, the code above returns a resolved promise with the result of 1, let’s test it:

async function f() {

return 1;

}

f().then(alert); // 1

…We could explicitly return a promise, that would be the same as:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

So, async ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, await, that works only inside async functions, and it’s pretty cool.

# [Await](https://javascript.info/async-await#await)

The syntax:

// works only inside async functions

let value = await promise;

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here’s an example with a promise that resolves in 1 second:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait till the promise resolves (\*)

alert(result); // "done!"

}f();

The function execution “pauses” at the line (\*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in one second.

Let’s emphasize: await literally makes JavaScript wait until the promise settles, and then go on with the result. That doesn’t cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc.

It’s just a more elegant syntax of getting the promise result than promise.then, easier to read and write.

**Can’t use await in regular functions**

If we try to use await in non-async function, there would be a syntax error:

function f() {

let promise = Promise.resolve(1);

let result = await promise; // Syntax error

}

We will get this error if we do not put async before a function. As said, await only works inside an async function.

The async keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows to use await in it.

The await keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it’s an error, the exception is generated, same as if throw error were called at that very place.
2. Otherwise, it returns the result, so we can assign it to a value.

# JavaScript String split() Method

The split() method is used to split a string into an array of substrings, and returns the new array.

**Tip:** If an empty string ("") is used as the separator, the string is split between each character.

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

## **Syntax**

*string*.split(separator, limit)

## **Parameter Values**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| separator | Optional. Specifies the character, or the regular expression, to use for splitting the string. If omitted, the entire string will be returned (an array with only one item) |
| limit | Optional. An integer that specifies the number of splits, items after the split limit will not be included in the array |

var str = "How are you doing today?";

var res = str.split("");

console.log(res); // ["H", "o", "w", " ", "a", "r", "e", " ", "y", "o", "u", " ", "d", "o", "i", "n", "g", " ", "t", "o", "d", "a", "y", "?"]

var str = "How are you doing today?";

var res = str.split(" "); // ["How", "are", "you", "doing", "today?"]

var str = "How are you doing today?";

var res = str.split(" ", 3);

console.log(res); // ["How", "are", "you"]

# JavaScript Array join() Method

The join() method joins the elements of an array into a string, and returns the string.

The elements will be separated by a specified separator. The default separator is comma (,).

*array*.join(separator)

|  |  |
| --- | --- |
| *separator* | Optional. The separator to be used. If omitted, the elements are separated with a comma |

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

var res = fruits.join();

console.log(res); // Banana,Orange,Apple,Mango

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

var res = fruits.join(" and ");

console.log(res); // Banana and Orange and Apple and Mango

# [JavaScript adding a string to a number](https://stackoverflow.com/questions/16522648/javascript-adding-a-string-to-a-number)

> 3 + 4 + "5"

"75"

> 3 + 4 + +"5" // unary operator

12

"2"+1= 21

1+"2" = 12

"2"+1+2 = 212

"2"+"5"+1+5 = 2515

"2"-1 = 1

1+2-"1" = 2

Since you have a **string data** type first anything that comes after the “+” operator will be **concatenated** to your data, i.e “123”.

However, if you were to have the string data type at the end in place of the number , i.e1+2+ “3” your result would be “33” as “+” on two numbers would result in an addition operation and then that result would be concatenated to your string.

# Debouncing in JavaScript

Debouncing in JavaScript is a practice used to improve browser performance. There might be some functionality in a web page which requires time-consuming computations. If such a method is invoked frequently, it might greatly affect the performance of the browser, as JavaScript is a single threaded language. Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, that it stalls the performance of the web page. In other words, it limits the rate at which a function gets invoked.

**Application:**  
Debouncing can be applied in implementing suggestive text, where we wait for the user to stop typing for a few seconds before suggesting the text. thus, on every keystroke, we wait for some seconds before giving out suggestions.  
Another application of debouncing is in content-loading webpages like Facebook and Twitter where the user keeps on scrolling. In these scenarios, if the scroll event is fired too frequently, there might be a performance impact, as it contains lots of videos and images. Thus the scroll event must make use of debouncing.

# ES5 functions vs. ES6 ‘fat arrow’ functions

Arrow functions in ES6 allow you to remove some of that syntax along with refining how to handle ‘this’.

function sayHi(greeting){

return greeting;

}

sayHi('Hello!');

// 'Hello!'

Syntactically, the previous function can be reduced down to one line.

var sayHi = greeting => greeting

sayHi('Hello!');

// 'Hello!'

As you can see, the ‘function’ keyword is no longer necessary. By using the fat arrow syntax ( => ) ES6 knows you are using a function. There are a few other bits of syntax magic going on here as well.

If you have only one argument, you don’t need to wrap it in ­parentheses.

If the code block has only one expression in it, you can eliminate the curly braces around it.

If you do eliminate those curly braces then you can also throw out the ‘return’ keyword since you will now have an implicit return.



Calling ‘this.owner’ from within that function doesn’t work. Why? Because ‘this’ is referring to the function that called it which doesn’t know what ‘owner’ is. What needs to happen is that ‘this’ needs to refer to the ‘pets’ object, not the function. In ES5 there are a couple of ways around this.

First you could .bind() this to the function.



Or, you could just pass along the ‘this’ context you want to use as an argument to the function.



Or you could even set the value of ‘this’ to a variable inside the first function and reference it in the second function



All of those work just fine. But an arrow function does away with both of those and makes it so you don’t have to figure out what exactly ‘this’ means when calling it in a function.

Just refactor the first example to an arrow function and you get…



How does this work? I like to think of it as fat arrow functions don’t have their own or don’t change the context of ‘this’. They leave it alone so that it stays the same as the context in which the function was created.

Arrow functions don't have their own this or arguments binding. Instead, those identifiers are resolved in the lexical scope like any other variable. That means that inside an arrow function, this and arguments refer to the values of this and arguments in the environment the arrow function is defined in (i.e. "outside" the arrow function):

// Example using a function expression

function createObject() {

console.log('Inside `createObject`:', this.foo);

return {

foo: 42,

bar: function() {

console.log('Inside `bar`:', this.foo);

},

};

}

createObject.call({foo: 21}).bar(); // override `this` inside createObject

// Example using a arrow function

function createObject() {

console.log('Inside `createObject`:', this.foo);

return {

foo: 42,

bar: () => console.log('Inside `bar`:', this.foo),

};

}

createObject.call({foo: 21}).bar(); // override `this` inside createObject

**Arrow functions cannot be called with new**

ES2015 distinguishes between functions that are callable and functions that are constructable. If a function is constructable, it can be called with new, i.e. new User(). If a function is callable, it cannot be called without new (i.e. normal function call).

Functions created through function declarations / expressions are both constructable and callable.  
Arrow functions (and methods) are only callable. class constructors are only constructable.

If you are trying to call a non-callable function or to construct a non-constructable function, you will get a runtime error.

# Pass By Value And Pass By Reference In JavaScript

In Pass by Value, Function is called by directly passing the value of the variable as the argument. Changing the argument inside the function doesn’t affect the variable passed from outside the function.

**Javascript always pass by value** so changing the value of the variable never changes the underlying primitive (String or number).

function callByValue(varOne, varTwo) {   
 console.log("Inside Call by Value Method");   
 varOne = 100;   
 varTwo = 200;   
 console.log("varOne =" + varOne +"varTwo =" +varTwo);   
}

let varOne = 10;   
let varTwo = 20;

console.log("Before Call by Value Method");   
console.log("varOne =" + varOne +"varTwo =" +varTwo);

callByValue(varOne, varTwo)

console.log("After Call by Value Method");   
console.log("varOne =" + varOne +"varTwo =" +varTwo);

output will be :   
---------------   
Before Call by Value Method   
varOne =10 varTwo =20   
Inside Call by Value Method   
varOne =100 varTwo =200   
After Call by Value Method   
varOne =10 varTwo =20

However, when a variable refers to an object which includes array, the value is the reference to the object.

#### Pass by Reference:

In Pass by Reference, Function is called by directly passing the reference/address of the variable as the argument. Changing the argument inside the function affect the variable passed from outside the function. In Javascript objects and arrays follows pass by reference.

function callByReference(varObj) {   
 console.log("Inside Call by Reference Method");   
 varObj.a = 100;   
 console.log(varObj);   
}   
let varObj = {a:1};

console.log("Before Call by Reference Method");   
console.log(varObj);

callByReference(varObj)

console.log("After Call by Reference Method");   
console.log(varObj);

output will be :   
---------------

Before Call by Reference Method   
{a: 1}   
Inside Call by Reference Method   
{a: 100}   
After Call by Reference Method   
{a: 100}

so if we are passing object or array as an argument to the method, then there is a possibility that value of the object can change.

* Javascript is *always* pass by value, but when a variable refers to an object (including arrays), the "value" is a reference to the object.
* Changing the value of a variable *never* changes the underlying primitive or object, it just points the variable to a new primitive or object.
* However, changing a *property* of an object referenced by a variable does change the underlying object.

So, to work through some of your examples:

function f(a,b,c) {

// Argument a is re-assigned to a new value.

// The object or primitive referenced by the original a is unchanged.

a = 3;

// Calling b.push changes its properties - it adds

// a new property b[b.length] with the value "foo".

// So the object referenced by b has been changed.

b.push("foo");

// The "first" property of argument c has been changed.

// So the object referenced by c has been changed (unless c is a primitive)

c.first = false;

}

var x = 4;

var y = ["eeny", "miny", "mo"];

var z = {first: true};

f(x,y,z);

console.log(x, y, z.first); // 4, ["eeny", "miny", "mo", "foo"], false

var a = ["1", "2", {foo:"bar"}];

var b = a[1]; // b is now "2";

var c = a[2]; // c now references {foo:"bar"}

a[1] = "4"; // a is now ["1", "4", {foo:"bar"}]; b still has the value

// it had at the time of assignment

a[2] = "5"; // a is now ["1", "4", "5"]; c still has the value

// it had at the time of assignment, i.e. a reference to

// the object {foo:"bar"}

console.log(b, c.foo); // "2" "bar"

# Understanding Deep and Shallow Copy in Javascript

#### Shallow copy

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

#### Deep copy

A deep copy copies all fields makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

### Lets take an example

Shallow Copy: It makes a copy of the reference to X into Y. Think about it as a copy of X’s Address. So, the addresses of X and Y will be the same i.e. they will be pointing to the same memory location.

Deep copy: It makes a copy of all the members of X, allocates different memory location for Y and then assigns the copied members to Y to achieve deep copy. In this way, if X vanishes Y is still valid in the memory.

The correct term to use would be cloning, where you know that they both are totally the same, but yet different (i.e. stored as two different locations in the memory space).

Consider this example:

var employeeDetailsOriginal = { name: 'Manjula', age: 25, Profession: 'Software Engineer' };

Let’s say you want to create a duplicate of this, so that even if you change the original values, you can always return to the original.

I can do this:

var employeeDetailsDuplicate = employeeDetailsOriginal; //Shallow copy!

If we change a value:

employeeDetailsDuplicate.name = 'NameChanged';

This statement will also change name from employeeDetailsOriginal, since we have a shallow copy, or a reference to var employeeDetailsOriginal. This means, you’re losing the original data as well.

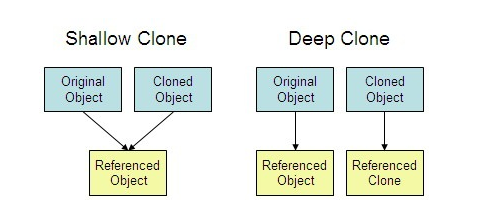
But, creating a brand new variable by using the properties from the original employeeDetailsOriginal variable, you can create a deep copy.

var employeeDetailsDuplicate = {

name: employeeDetailsOriginal.name, age: employeeDetailsOriginal.age, Profession: employeeDetailsOriginal.Profession

}; //Deep copy!

Now if you change employeeDetailsDuplicate.name, it will only affect employeeDetailsDuplicate and not employeeDetailsOriginal



What is Babel in JavaScript?

**Babel** is a **JavaScript** transpiler that converts edge **JavaScript** into plain old ES5 **JavaScript** that can run in any browser (even the old ones). It makes available all the syntactical sugar that was added to **JavaScript** with the new ES6 specification, including classes, fat arrows and multiline strings.

# [What is setTimeout doing when set to 0 milliseconds?](https://stackoverflow.com/questions/33955650/what-is-settimeout-doing-when-set-to-0-milliseconds)

A few useful facts might help clarify what's happening:

1. JavaScript is single-threaded. Asynchronous callbacks are assigned to a *message* placed in a *message queue*.
2. When no code is currently executing, the *event loop* polls the message queue, requesting the next message in line to be processed (executed).
3. setTimeout adds a message (with the callback provided) to the end of this queue after the specified delay has elapsed.

(Note: this means the delay in a setTimeout call is not a sure thing; it is the **minimum delay** before the callback is executed. The actual time taken depends on how long it takes to process any messages ahead of it in the queue.)

So what happens if the delay is set to 0? A new message is added to the queue immediately, and will be processed when the currently executing code is finished and any previously-added messages have been processed.

**What's happening in your code**

When you invoke setTimeout…

setTimeout(function() {

console.log('AAA');

}, 0);

…a message gets added to the queue with the specified callback. The rest of your code…

for (i = 0; i < 1000; i++) {

console.log('BBB');

}

// etc.

…continues executing synchronously. Once it has completely finished, the event loop polls the message queue for the next message and finds the one with your setTimeout callback, which is then processed (the callback is run).

The callback only ever gets executed *after* the currently executing code has finished, no matter how long that takes.

**console.log(1);**

**setTimeout(function() {**

**console.log('AAA');**

**}, 0);**

**function xyz() {**

**console.log(3);}**

**xyz();**

**O/P:** 1,3,AAA

# Understanding rxjs Subjects

### Subjects

A Subject is like an Observable. It can be subscribed to, just like you normally would with Observables. It also has methods like next(), error() and complete() just like the observer you normally pass to your Observable creation function.

The main reason to use Subjects is to multicast. An Observable by default is unicast. Unicasting means that each subscribed observer owns an independent execution of the Observable. To demonstrate this:

|  |
| --- |
| import \* as Rxfrom "rxjs"; |
|  |  |
|  | const observable = Rx.Observable.create((observer) => { |
|  | observer.next(Math.random()); |
|  | }); |
|  |  |
|  | // subscription 1 |
|  | observable.subscribe((data) => { |
|  | console.log(data); // 0.24957144215097515 (random number) |
|  | }); |
|  |  |
|  | // subscription 2 |
|  | observable.subscribe((data) => { |
|  | console.log(data); // 0.004617340049055896 (random number) |
|  | }); |
|  |  |

While Observables are unicast by design, this can be pretty annoying if you expect that each subscriber receives the same values. Subjects can help us overcome this issue. As mentioned before, Subjects can multicast. Multicasting basically means that one Observable execution is shared among multiple subscribers.

Subjects are like EventEmitters, they maintain a registry of many listeners. When calling subscribe on a Subject it does not invoke a new execution that delivers data. It simply registers the given Observer in a list of Observers.

### So how to use Subjects to multicast

Multicasting is a characteristic of a Subject. You don’t have to do anything special to achieve this behaviour. This is a small multicast demonstration:

|  |
| --- |
| import \* as Rx from "rxjs"; |
|  |  |
|  | const subject = new Rx.Subject(); |
|  |  |
|  | // subscriber 1 |
|  | subject.subscribe((data) => { |
|  | console.log(data); // 0.24957144215097515 (random number) |
|  | }); |
|  |  |
|  | // subscriber 2 |
|  | subject.subscribe((data) => { |
|  | console.log(data); // 0.24957144215097515 (random number) |
|  | }); |
|  |  |
|  | subject.next(Math.random()); |

Nice! Now i got two subscriptions getting the same data. This however is not all that Subjects can do.

Whereas Observables are solely data producers, Subjects can both be used as a data producer and a data consumer. By using Subjects as a data consumer you can use them to convert Observables from unicast to multicast. Here’s a demonstration of that:

|  |
| --- |
| import \* as Rx from "rxjs"; |
|  |  |
|  | const observable = Rx.Observable.create((observer) => { |
|  | observer.next(Math.random()); |
|  | }); |
|  |  |
|  | const subject = new Rx.Subject(); |
|  |  |
|  | // subscriber 1 |
|  | subject.subscribe((data) => { |
|  | console.log(data); // 0.24957144215097515 (random number) |
|  | }); |
|  |  |
|  | // subscriber 2 |
|  | subject.subscribe((data) => { |
|  | console.log(data); // 0.24957144215097515 (random number) |
|  | }); |
|  |  |
|  | observable.subscribe(subject); |

We pass our Subject to the subscribe function and let it take the values that come out of the Observable (data consuming). All the subscribers to that Subject will then all immediately receive that value.

### The BehaviorSubject

One of the variants of the Subject is the BehaviorSubject. The BehaviorSubject has the characteristic that it stores the “current” value. This means that you can always directly get the last emitted value from the BehaviorSubject.

There are two ways to get this last emited value. You can either get the value by accessing the .value property on the BehaviorSubject or you can subscribe to it. If you subscribe to it, the BehaviorSubject will directly emit the current value to the subscriber. Even if the subscriber subscribes much later than the value was stored. See the example below:

|  |
| --- |
| import \* as Rx from "rxjs"; |
|  |  |
|  | const subject = new Rx.BehaviorSubject(); |
|  |  |
|  | // subscriber 1 |
|  | subject.subscribe((data) => { |
|  | console.log('Subscriber A:', data); |
|  | }); |
|  |  |
|  | subject.next(Math.random()); |
|  | subject.next(Math.random()); |
|  |  |
|  | // subscriber 2 |
|  | subject.subscribe((data) => { |
|  | console.log('Subscriber B:', data); |
|  | }); |
|  |  |
|  | subject.next(Math.random()); |
|  |  |
|  | console.log(subject.value) |
|  |  |
|  | // output |
|  | // Subscriber A: 0.24957144215097515 |
|  | // Subscriber A: 0.8751123892486292 |
|  | // Subscriber B: 0.8751123892486292 |
|  | // Subscriber A: 0.1901322109907977 |
|  | // Subscriber B: 0.1901322109907977 |
|  | // 0.1901322109907977 |

There are a few things happening here:

1. We first create a subject and subscribe to that with Subscriber A. The Subject then emits it’s value and Subscriber A will log the random number.
2. The subject emits it’s next value. Subscriber A will log this again
3. Subscriber B starts with subscribing to the subject. Since the subject is a BehaviorSubject the new subscriber will automatically receive the last stored value and log this.
4. The subject emits a new value again. Now both subscribers will receive the values and log them.
5. Last we log the current Subjects value by simply accessing the .valueproperty. This is quite nice as it’s synchronous. You don’t have to call subscribe to get the value.

Last but not least, you can create BehaviorSubjects with a start value. When creating Observables this can be quite hard. With BehaviorSubjects this is as easy as passing along an initial value. See the example below:

|  |
| --- |
| import \* as Rx from "rxjs"; |
|  |  |
|  | const subject = new Rx.BehaviorSubject(Math.random()); |
|  |  |
|  | // subscriber 1 |
|  | subject.subscribe((data) => { |
|  | console.log('Subscriber A:', data); |
|  | }); |
|  |  |
|  | // output |
|  | // Subscriber A: 0.24957144215097515 |

# Promise vs Observable

* a *Promise*is eager, whereas an *Observable*is lazy,
* a *Promise*is always asynchronous, while an *Observable*can be either synchronous or asynchronous,
* a *Promise*can provide a single value, whereas an *Observable*is a stream of values (from 0 to multiple values),
* you can apply RxJS operators to an *Observable*to get a new tailored stream.

Primitive data types include the following:

* Number — e.g. 1
* String — e.g. 'Hello’
* Boolean — e.g. true
* Undefined
* Null

# [What is event bubbling and capturing?](https://stackoverflow.com/questions/4616694/what-is-event-bubbling-and-capturing)

Event bubbling and capturing are two ways of event propagation in the HTML DOM API, when an event occurs in an element inside another element, and both elements have registered a handle for that event. The event propagation mode determines in [which order the elements receive the event](http://www.quirksmode.org/js/events_order.html).

With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements.

With capturing, the event is first captured by the outermost element and propagated to the inner elements.

On the other hand, the [performance of event bubbling may be slightly lower](https://stackoverflow.com/a/10335117/1269037) for complex DOMs.

## **Example**

<div>

<ul>

<li></li>

</ul>

</div>

In the structure above, assume that a click event occurred in the li element.

In capturing model, the event will be handled by the div first (click event handlers in the div will fire first), then in the ul, then at the last in the target element, li.

In the bubbling model, the opposite will happen: the event will be first handled by the li, then by the ul, and at last by the div element.

We can use the addEventListener(type, listener, useCapture) to register event handlers for in either bubbling (default) or capturing mode. To use the capturing model pass the third argument as true.

In the example below, if you click on any of the highlighted elements, you can see that the capturing phase of the event propagation flow occurs first, followed by the bubbling phase.

var logElement = document.getElementById('log');

function log(msg) {

logElement.innerHTML += ('<p>' + msg + '</p>');

}

function capture() {

log('capture: ' + this.firstChild.nodeValue.trim());

}

function bubble() {

log('bubble: ' + this.firstChild.nodeValue.trim());

}

function clearOutput() {

logElement.innerHTML = "";

}

var divs = document.getElementsByTagName('div');

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

divs[i].addEventListener('click', capture, true); // capturing

divs[i].addEventListener('click', bubble, false); // bubbling

}

var clearButton = document.getElementById('clear');

clearButton.addEventListener('click', clearOutput);

p {

line-height: 0;

}

div {

display:inline-block;

padding: 5px;

background: #fff;

border: 1px solid #aaa;

cursor: pointer;

}

div:hover {

border: 1px solid #faa;

background: #fdd;

}

<div>1

<div>2

<div>3

<div>4

<div>5</div>

</div>

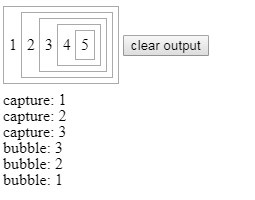
</div>

</div>

</div>

<button id="clear">clear output</button>

<section id="log"></section>



# JavaScript’s Prototype

## Objects Inherit from Objects

If you’ve ever read **anything** about inheritance in JS, then you’ve almost certainly heard that objects inherit from other objects.

This is true, and once you already understand it, it’s a good way to think about things. But my experience has been that this explanation alone isn’t really sufficient.

So I’m going to break from tradition here and not actually try to explain how inheritance works right up front. Instead, we’ll start with some simpler stuff that will hopefully provide a bit of context later on.

## Function prototypes

Here’s a fun fact: In JavaScript, all functions are also objects, which means that they can have properties. And as it so happens, they all have a property called `prototype`, which is also an object.

function foo() {}

typeof foo.prototype // ‘object’

That’s pretty simple, right? Any time you create a function, it will automatically have a property called **prototype**, which will be initialized to an empty object.

## Constructors

In JavaScript, there’s really no difference between a “regular” function and a constructor function. They’re actually all the same. But as a convention, functions that are meant to be used as constructors are generally capitalized.

By the way, if you don’t know what I mean when I say “constructor”, that’s totally okay. We’ll get there.

So let’s say that we want to make a constructor function called **Dog**, because explaining inheritance using animals is a time-honored tradition and I’m kind of a nostalgic dude.

function Dog() {}

If I want to make an instance of **Dog**, I use the **new** keyword. That’s really what I mean when I talk about constructors — I’m using the function to construct a new object. Any time you see the **new** keyword, it means that the following function is being used as a constructor.

var fido = new Dog();

So now we have **fido**, who’s a **Dog**. But he doesn’t really do anything.

## Methods

It’s time to make our **Dog** a little more dog-like:

function Dog() {}

Dog.prototype.bark = function() {  
 console.log(‘woof!’);  
};

You should remember from earlier that all functions automatically get initialized with a **prototype** object. In the example above, we tacked a function onto it called **bark**.

Now let’s make ourselves a new **fido**.

function Dog() {}

Dog.prototype.bark = function() {  
 console.log(‘woof!’);  
};

var fido = new Dog();

fido.bark(); // ‘woof!’

the important thing to take away right now is that by placing **bark** on **Dog.prototype**, we made it available to all instances of **Dog**.

## Differential Inheritance

JavaScript uses an inheritance model called “differential inheritance”. What that means is that methods aren’t copied from parent to child. Instead, children have an “invisible link” back to their parent object

For example, **fido** doesn’t actually have its own method called **bark()** (in other words, **fido.hasOwnProperty(‘bark’) === false**).

1. The JS engine looks for a property called **bark** on our **fido** object.  
2. It doesn’t find one, so it looks “up the prototype chain” to **fido**’s parent, which is **Dog.prototype**.  
3. It finds **Dog.prototype.bark**, and calls it with **this** bound to **fido**.

There’s really no such property as **fido.bark**. It doesn’t exist. Instead, **fido**has *access* to the **bark()** method on **Dog.prototype** because it’s an instance of **Dog**. This is the “invisible link” I mentioned. More commonly, it’s referred to as the “prototype chain”.

## Object.create()

JavaScript has had a cool little function called **Object.create()**.

Here’s how it works.

var parent = {  
 foo: function() {  
 console.log(‘bar’);  
 }  
};

var child = Object.create(parent);

child.hasOwnProperty(‘foo’); // false  
child.foo(); // ‘bar’

So what is it doing?

Essentially, it creates a new, empty object that has **parent** in its prototype chain. That means that even though **child** doesn’t have its own **foo()**method, it has access to the **foo()** method from **parent**.

## **JavaScript Scope and Closures**

### Scope

A scope in JavaScript defines what variables you have access to. There are two kinds of scope – global scope and local scope.

#### Global scope

If a variable is declared outside all functions or curly braces ({}), it is said to be defined in the **global scope**.

const globalVariable = 'some value'

Once you’ve declared a global variable, you can use that variable anywhere in your code, even in functions.



Although you can declare variables in the global scope, it is advised not to. This is because there is a chance of naming collisions, where two or more variables are named the same. If you declared your variables with constor let, you would receive an error whenever a name collision happens. This is undesirable.

*// Don't do this!*

let thing = 'something'

let thing = 'something else' *// Error, thing has already been declared*

If you declare your variables with var, your second variable overwrites the first one after it is declared. This also undesirable as you make your code hard to debug.

*// Don't do this!*

var thing = 'something'

var thing = 'something else' *// perhaps somewhere totally different in your code*

console.log(thing) *// 'something else'*

So, you should always declare local variables, not global variables.

#### Local Scope

Variables that are usable only in a specific part of your code are considered to be in a local scope. These variables are also called **local variables**.

In JavaScript, there are two kinds of local scope: function scope and block scope.

Let’s talk about function scopes first.

#### Function scope

When you declare a variable in a function, you can access this variable only within the function. You can’t get this variable once you get out of it.

In the example below, the variable hello is in the sayHello scope:

function sayHello () {

const hello = 'Hello CSS-Tricks Reader!'

console.log(hello)

}

sayHello() *// 'Hello CSS-Tricks Reader!'*

console.log(hello) *// Error, hello is not defined*

#### Block scope

When you declare a variable with const or let within a curly brace ({}), you can access this variable only within that curly brace.

In the example below, you can see that hello is scoped to the curly brace:

{

const hello = 'Hello CSS-Tricks Reader!'

console.log(hello) *// 'Hello CSS-Tricks Reader!'*

}

console.log(hello) *// Error, hello is not defined*

The block scope is a subset of a function scope since functions need to be declared with curly braces (unless you’re using [arrow functions](https://zellwk.com/blog/es6/#arrow-functions) with an implicit return).

#### Function hoisting and scopes

Functions, when declared with a function declaration, are always hoisted to the top of the current scope. So, these two are equivalent:

*// This is the same as the one below*

sayHello()

function sayHello () {

console.log('Hello CSS-Tricks Reader!')

}

*// This is the same as the code above*

function sayHello () {

console.log('Hello CSS-Tricks Reader!')

}

sayHello()

When declared with a function expression, functions are not hoisted to the top of the current scope

sayHello() *// Error, sayHello is not defined*

const sayHello = function () {

console.log(aFunction)

}

Because of these two variations, function hoisting can potentially be confusing, and should not be used. Always declare your functions before you use them.

#### Functions do not have access to each other’s scopes

Functions do not have access to each other’s scopes when you define them separately, even though one function may be used in another.

In this example below, second does not have access to firstFunctionVariable.

function first () {

const firstFunctionVariable = `I'm part of first`

}

function second () {

first()

console.log(firstFunctionVariable) *// Error, firstFunctionVariable is not defined* }

#### Nested scopes

When a function is defined in another function, the inner function has access to the outer function’s variables. This behavior is called **lexical scoping**.

However, the outer function does not have access to the inner function’s variables.

function outerFunction () {

const outer = `I'm the outer function!`

function innerFunction() {

const inner = `I'm the inner function!`

console.log(outer) *// I'm the outer function!*

}

console.log(inner) *// Error, inner is not defined*

}

To visualize how scopes work, you can imagine one-way glass. You can see the outside, but people from the outside cannot see you.

### Closures

Whenever you create a function within another function, you have created a closure. The inner function is the closure. This closure is usually returned so you can use the outer function’s variables at a later time.

function outerFunction () {

const outer = `I see the outer variable!`

function innerFunction() {

console.log(outer)

}

return innerFunction

}

outerFunction()() *// I see the outer variable!*

Since closures have access to the variables in the outer function, they are usually used for two things:

1. To control side effects
2. To create private variables

# Var, Let, and Const – What's the Difference?

## **Var**

Before the advent of ES6, var declarations ruled. There are issues associated with variables declared with var, though. That is why it was necessary for new ways to declare variables to emerge. First, let's get to understand var more before we discuss those issues.

### **Scope of var**

**Scope** essentially means where these variables are available for use. var declarations are globally scoped or function/locally scoped.

The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.

var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

To understand further, look at the example below.

var greeter = "hey hi";

function newFunction() {

var hello = "hello";

}

Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So we cannot access the variable hello outside of a function. So if we do this:

var tester = "hey hi";

function newFunction() {

var hello = "hello";

}

console.log(hello); // error: hello is not defined

We'll get an error which is as a result of hello not being available outside the function.

### var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

var greeter = "hey hi";

var greeter = "say Hello instead";

and this also

var greeter = "hey hi";

greeter = "say Hello instead";

### **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"

So, since times > 3 returns true, greeter is redefined  to "say Hello instead". While this is not a problem if you knowingly want greeter to be redefined, it becomes a problem when you do not realize that a variable greeter has already been defined before.

If you have used greeter in other parts of your code, you might be surprised at the output you might get. This will likely cause a lot of bugs in your code. This is why letand const are necessary.

## **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 me explain this with an example:

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello);// "say Hello instead"

}

console.log(hello) // hello is not defined

We see that using hello outside its block (the curly braces where it was defined) returns an error. This is because let variables are block scoped

### **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"; // error: Identifier 'greeting' has already been declared

However, if the same variable is defined in different scopes, there will be no error:

let greeting = "say Hi";

if (true) {

let greeting = "say Hello instead";

console.log(greeting); // "say Hello instead"

}

console.log(greeting); // "say Hi"

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

This fact makes let a better choice than var. When using let, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope.

Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with var does not happen.

### **Hoisting of let**

Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to 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.

### **const declarations are block scoped**

Like let declarations, const declarations can only be accessed within the block they were declared.

This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared. So if we declare a variable with const, we can neither do this:

const greeting = "say Hi";

greeting = "say Hello instead";// error: Assignment to constant variable.

nor this:

const greeting = "say Hi";

const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared

Every const declaration, therefore, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a const object as this:

const greeting = {

message: "say Hi",

times: 4

}

while we cannot do this:

const greeting = {

words: "Hello",

number: "five"

} // error: Assignment to constant variable.

we can do this:

greeting.message = "say Hello instead";

### **Hoisting of const**

Just like let, const declarations are hoisted to the top but are not initialized.

So just in case you missed the differences, here they are:

* var declarations are globally scoped or function scoped while let and const are block scoped.
* var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.
* They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.
* While var and let can be declared without being initialized, const must be initialized during declaration.

# LocalStorage, sessionStorage

What’s interesting about them is that the data survives a page refresh (for sessionStorage) and even a full browser restart (for localStorage).

We already have cookies. Why additional objects?

* Unlike cookies, web storage objects are not sent to server with each request. Because of that, we can store much more. Most browsers allow at least 2 megabytes of data (or more) and have settings to configure that.
* Unlike cookies, the server can’t manipulate storage objects via HTTP headers. Everything’s done in JavaScript.
* The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or subdomains infer different storage objects, they can’t access data from each other.

Both storage objects provide same methods and properties:

* setItem(key, value) – store key/value pair.
* getItem(key) – get the value by key.
* removeItem(key) – remove the key with its value.
* clear() – delete everything.
* key(index) – get the key on a given position.
* length – the number of stored items.

## **[localStorage demo](https://javascript.info/localstorage" \l "localstorage-demo)**

The main features of localStorage are:

* Shared between all tabs and windows from the same origin.
* The data does not expire. It remains after the browser restart and even OS reboot.

For instance, if you run this code…

localStorage.setItem('test', 1);

…And close/open the browser or just open the same page in a different window, then you can get it like this:

alert( localStorage.getItem('test') ); // 1

We only have to be on the same origin (domain/port/protocol), the url path can be different.

The localStorage is shared between all windows with the same origin, so if we set the data in one window, the change becomes visible in another one.

## **[Looping over keys](https://javascript.info/localstorage" \l "looping-over-keys)**

As we’ve seen, the methods provide “get/set/remove by key” functionality. But how to get all saved values or keys?

Unfortunately, storage objects are not iterable.

One way is to loop over them as over an array:

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

let key = localStorage.key(i);

alert(`${key}: ${localStorage.getItem(key)}`);

}

## **[Strings only](https://javascript.info/localstorage" \l "strings-only)**

Please note that both key and value must be strings.

If were any other type, like a number, or an object, it gets converted to string automatically:

sessionStorage.user = {name: "John"};

alert(sessionStorage.user); // [object Object]

We can use JSON to store objects though:

sessionStorage.user = JSON.stringify({name: "John"});

// sometime later

let user = JSON.parse( sessionStorage.user );

alert( user.name ); // John

Save to local storage

localStorage.setItem('currentUser', JSON.stringify({ token: token, name: name }));

Load from local storage

var currentUser = JSON.parse(localStorage.getItem('currentUser'));

var token = currentUser.token; // your token

## **[sessionStorage](https://javascript.info/localstorage" \l "sessionstorage)**

The sessionStorage object is used much less often than localStorage.

Properties and methods are the same, but it’s much more limited:

* The sessionStorage exists only within the current browser tab.
  + Another tab with the same page will have a different storage.
  + But it is shared between iframes in the same tab (assuming they come from the same origin).
* The data survives page refresh, but not closing/opening the tab.

sessionStorage.setItem('test', 1);

…Then refresh the page. Now you can still get the data:

alert( sessionStorage.getItem('test') ); // after refresh: 1

…But if you open the same page in another tab, and try again there, the code above returns null, meaning “nothing found”.

That’s exactly because sessionStorage is bound not only to the origin, but also to the browser tab. For that reason, sessionStorage is used sparingly.

| **localStorage** | **sessionStorage** |
| --- | --- |
| Shared between all tabs and windows with the same origin | Visible within a browser tab, including iframes from the same origin |
| Survives browser restart | Survives page refresh (but not tab close) |

# apply(), call(), and bind()

##### **Basic rules worth remembering:**

1. “this” always refers to an object.
2. “this” refers to an object which calls the function it contains.
3. In the global context “this” refers to either window object or is undefined if the ‘strict mode’ is used.

var car = {

registrationNumber: "GA12345",

brand: "Toyota",

displayDetails: function() {

console.log(this.registrationNumber + " " + this.brand);

}

}

# The above will work perfectly fine as long as we use it this way:

# car.displayDetails(); // GA12345 Toyota

# But what if we want to borrow a method?

# var myCarDetails = car.displayDetails;

# myCarDetails();

# Well, this won’t work as the “this” will be now assigned to the global context which doesn’t have neither the registrationNumber nor the brand property.

### **The bind() Method**

# For such cases we can use the ECMAScript 5 bind() method of the Function.prototype property. This means bind() can be used by every single function.

# var myCarDetails = car.displayDetails.bind(car); myCarDetails(); // GA12345 Toyota

The bind() method creates a new function where “this” refers to the parameter in the parenthesis in the above case “car”. This way the bind() method enables calling a function with a specified “this” value.

What if we would like to pass a parameter to the displayDetails function? We can use the bind method again. The following argument of the bind() method will provide an argument to the function bind() is called on.

Let me rewrite the car object:

var car = { registrationNumber: "GA12345",

brand: "Toyota",

displayDetails: function(ownerName) {

console.log(ownerName + ", this is your car: " + this.registrationNumber + " " + this.brand);

}

}

Example of passing arguments with bind():

var myCarDetails = car.displayDetails.bind(car, "Vivian"); // Vivian, this is your car: GA12345 Toyota

#### **call() and apply() methods**

Similar but slightly different usage provide the call() and apply() methods which also belong to the Function.prototype property.

This time there is a car object without the displayDetails function, which is located in the global context.

var car = {

registrationNumber: "GA12345",

brand: "Toyota" }

function displayDetails(ownerName) {

console.log(ownerName + ", this is your car: " + this.registrationNumber + " " + this.brand);

}

We can use the apply() function:

displayDetails.apply(car, ["Vivian"]); // Vivian, this is your car: GA12345 Toyota

Or

displayDetails.call(car, "Vivian"); // Vivian, this is your car: GA12345 Toyota

Note that when using the apply() function the parameter must be placed in an array. Call() accepts both an array of parameters and a parameter itself. Both are great tools for borrowing functions in JavaScript.