JavaScript

1. [Difference between localStorage , sessionStorage and Cookies](#_Difference_between_localStorage,)
2. [Closures](#_Closures)
3. Callback functions
4. Event Loop
5. Prototype – Inheritance
6. Var vs. Let
7. Map vs Filter vs reduce

# Difference between localStorage, sessionStorage and Cookies

**Cookies**

* Has different expiration dates (both the server or client can set up expiration date)
* The Client can't access the Cookies if the HttpOnly flag is true
* Has SSL Support
* Data are transferred on each HTTP request
* 4kb limit

**Local Storage**

* Has no expiration date
* Client only
* Has no SSL support
* Data are not transferred on each HTTP request
* 5 mb limit (check with the browser)

**Session Storage**

* Data is gone when you close the browser tab
* Client only
* Has no SSL support
* Data are not transferred on each HTTP request
* 5-10 mb limit (check with the browser)

# Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Example:

function outer() {

var b = 10;

var c = 100;

function inner() {

var a = 20;

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

a++;

b++;

}

return inner;

}

var X = outer(); // outer() invoked the first time

var Y = outer(); // outer() invoked the second time

//end of outer() function executions

X(); // X() invoked the first time

X(); // X() invoked the second time

X(); // X() invoked the third time

Y(); // Y() invoked the first time

When you run this code, you will see the following output in the console.log:

a=20 b=10

a=20 b=11

a=20 b=12

a=20 b=10

# Callback Functions

In the real world, callbacks are most often used with asynchronous functions.

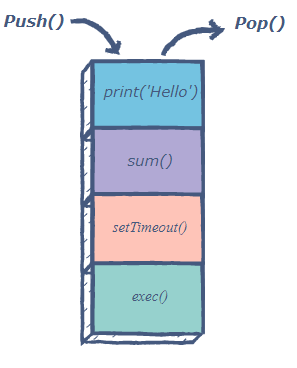
A typical example is JavaScript setTimeout().

setTimeout(myFunction, 3000);  
  
function myFunction() {  
  document.getElementById("demo").innerHTML = "I love You !!";  
}

# Even Loop

The **event loop** is the secret behind JavaScript’s asynchronous programming. JS executes all operations on a single thread, but using a few smart data structures, it gives us the illusion of multi-threading. Let’s take a look at what happens on the back-end.

The **event queue** is responsible for sending new functions to the track for processing. It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.



# Prototype

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

## [Writing doesn’t use prototype](https://javascript.info/prototype-inheritance" \l "writing-doesn-t-use-prototype)

The prototype is only used for reading properties.

Write/delete operations work directly with the object.

In the example below, we assign its own walk method to rabbit:

let animal = {

eats: true,

walk() {

/\* this method won't be used by rabbit \*/

}

};

let rabbit = {

\_\_proto\_\_: animal

};

rabbit.walk = function() {

alert("Rabbit! Bounce-bounce!");

};

rabbit.walk(); // Rabbit! Bounce-bounce!

# Var vs. let

Main difference is scoping rules. Variables declared by var keyword are scoped to the immediate function body (hence the function scope) while let variables are scoped to the immediate enclosing block denoted by { } (hence the block scope).

function run() {

var foo = "Foo";

let bar = "Bar";

console.log(foo, bar); // Foo Bar

{

var moo = "Mooo"

let baz = "Bazz";

console.log(moo, baz); // Mooo Bazz

}

console.log(moo); // Mooo

console.log(baz); // ReferenceError

}

run();

## Hoisting

While variables declared with var keyword are [hoisted](https://dev.to/godcrampy/the-secret-of-hoisting-in-javascript-egi) (initialized with undefined before the code is run) which means they are accessible in their enclosing scope even before they are declared:

function run() {

console.log(foo); // undefined

var foo = "Foo";

console.log(foo); // Foo

}

run();

let variables are not initialized until their definition is evaluated. Accessing them before the initialization results in a ReferenceError. Variable said to be in "temporal dead zone" from the start of the block until the initialization is processed.

function checkHoisting() {

console.log(foo); // ReferenceError

let foo = "Foo";

console.log(foo); // Foo

}

checkHoisting();

# Map vs. Filter vs. reduce

Map, Filter, and Reduce are Array methods which help to create new arrays in various ways. They are all 'higher order' functions because they take user-defined functions as parameters.

**Map:** returns an array of pieces of information from the original array. In the callback function, return the data you wish to be part of the new array.

// Durations are in minutes

const tasks = [

  {

    'name'     : 'Write for Envato Tuts+',

    'duration' : 120

  },

  {

    'name'     : 'Work out',

    'duration' : 60

  },

  {

    'name'     : 'Procrastinate on Duolingo',

    'duration' : 240

  }

];

Let's say we want to create a new array with just the name of each task.

const task\_names = tasks.map(function (task, index, array) {

    return task.name;

});

console.log(task\_names) // [ 'Write for Envato Tuts+', 'Work out', 'Procrastinate on Duolingo' ]

**Filter:** returns a subset of the original array based on custom criteria. In your callback function, return a Boolean value to determine whether or not each item will be included in the new array.

const words = ['Python', 'Javascript', 'Go', 'Java', 'PHP', 'Ruby'];

const result = words.filter(word => word.length < 8);

console.log(result);

**Reduce:** can be used to return almost anything. It is often used to return a single number, like a sum, but it can also be used to combine the logic of Map and Filter to return an array of values matching certain criteria. This can remove unnecessary iterations.

*map* creates a new array by transforming every element in an array, individually. *filter* creates a new array by removing elements that don't belong. *reduce*, on the other hand, takes all of the elements in an array and *reduces* them into a single value.

Just like *map* and *filter*, reduce is defined on *Array.prototype* and so is available on any array, and you pass a callback as its first argument. But it also takes a second argument: the value to start combining all your array elements into.

Let's say you want to find the sum of a list of numbers. Using a loop, it would look like this:

|  |  |
| --- | --- |
|  |  |

let numbers = [1, 2, 3, 4, 5],

    total = 0;

numbers.forEach(function (number) {

    total += number;

});

console.log(total); // 15

First, we call reduce on our list of numbers. We pass it a callback, which accepts the previous value and current value as arguments, and returns the result of adding them together. Since we passed 0 as a second argument to reduce, it'll use that as the value of previous on the first iteration.

With arrow functions, we would write it like this:

const total = [1, 2, 3, 4, 5].reduce((previous, current) => previous+current),0;

console.log(total) // 15