Javascript

JavaScript ES6 (also known as ECMAScript 2015 or ECMAScript 6) is **the newer version of JavaScript that was introduced in 2015**. ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

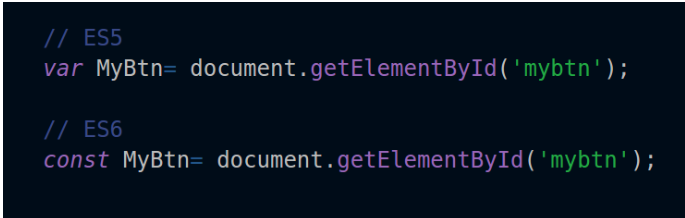
https://www.freecodecamp.org/news/write-less-do-more-with-javascript-es6-5fd4a8e50ee2/ -- important link

JavaScript ES6 brings new syntax and new awesome features to make your code more modern and more readable. It allows you to write less code and do more. ES6 introduces us to many great features like arrow functions, template strings, class destruction, Modules… and more.

**const and let**

const is a new keyword in ES6 for declaring variables. const is more powerful than var. Once used, the variable can’t be reassigned. In other words, it’s an **immutable variable**except when it used with objects.

This is really useful for targeting the selectors. For example, when we have a single button that fires an event, or when you want to select an HTML element in JavaScript, use const instead of var. This is because var is ‘hoisted’. It’s always preferable to use const when don’t want to reassign the variable .



https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/

Functions

**All JavaScript functions return something**. If an explicit return is omitted, undefined is returned automatically instead.

 When a function returns, its instance is wiped out from memory, which also frees all the variables in its scope to be wiped out if nothing else points to them.

In JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. What distinguishes them from other objects is that functions can be called. In brief, they are [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) objects.

Every function in JavaScript is a Function object. See [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) for information on properties and methods of Function objects.

To return a value other than the default, a function must have a [return](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return) statement that specifies the value to return. A function without a return statement will return a default value. In the case of a [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor) called with the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) keyword, the default value is the value of its this parameter. For all other functions, the default return value is [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined).

Classes

Classes were introduced in [EcmaScript 2015](https://262.ecma-international.org/6.0/) (ES6) to provide a cleaner way to follow object-oriented programming patterns.

https://www.freecodecamp.org/news/javascript-classes-how-they-work-with-use-case/

constructor functions

//In JavaScript, a constructor function is used to create objects.

Technically speaking, a constructor function is a regular [function](https://www.javascripttutorial.net/javascript-function/) with the following convention:

* The name of a constructor function starts with a capital letter like Person, Document, etc.
* A constructor function should be called only with the new operator.

Note that ES6 introduces the [class](https://www.javascripttutorial.net/es6/javascript-class/) keyword that allows you to define a custom type. And classes are just syntactic sugar over the constructor functions with some enhancements.

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

let person = new Person("ABC", "XYZ", 20, "green");

Basically, the new operator does the following:

* Create a new empty object and assign it to the this variable.
* Assign the arguments 'John' and 'Doe' to the firstName and lastName properties of the object.
* Return the this value.

Function() constructor

We see that a function is indeed an object. JavaScript functions are a special type of objects, called *function objects*. A function object includes a string which holds the actual code -- the function body -- of the function. The code is literally just a string. Although not recommended, you can create a new function object by passing the built-in Function constructor a string of code, like so:

The Function**constructor** creates a new Function **object**.

**var body = "return Math.PI \* radius \* radius";**

**var circle = new Function("radius", body);**

**console.log(circle(5)); // => 78.5398..**

const sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 6));

// expected output: 8

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

new Function(functionBody)

new Function(arg1, functionBody)

new Function(arg1, ... argN, functionBody)

Prototype

## https://www.w3schools.com/js/js\_object\_prototypes.asp

## Prototype Inheritance

All JavaScript objects inherit properties and methods from a prototype:

* Date objects inherit from Date.prototype
* Array objects inherit from Array.prototype
* Person objects inherit from Person.prototype

https://chamikakasun.medium.com/javascript-prototype-and-prototype-chain-explained-fdc2ec17dd04

**Prototype** is basically a property of a JavaScript function. At each time we create a function in JavaScript, JavaScript engine adds an extra property called **prototype** to the created function. This prototype property is an object (called as prototype object) has a **constructor** property by default. This constructor property points back(i.e blue arrow) to the function object on which the prototype object is a property.

https://chamikakasun.medium.com/javascript-factory-functions-vs-constructor-functions-585919818afe

https://www.freecodecamp.org/news/a-beginners-guide-to-javascripts-prototype/

Objects are key/value pairs. The most common way to create an object is with curly braces {} and you add properties and methods to an object using dot notation.

let animal = {}

animal.name = 'Leo'

animal.energy = 10

animal.eat = function (amount) {

console.log(`${this.name} is eating.`)

this.energy += amount

}

animal.sleep = function (length) {

console.log(`${this.name} is sleeping.`)

this.energy += length

}

animal.play = function (length) {

console.log(`${this.name} is playing.`)

this.energy -= length

}

Simple. Now odds are in our application we'll need to create more than one animal. Naturally the next step for this would be to encapsulate that logic inside of a function that we can invoke whenever we needed to create a new animal. We'll call this pattern Functional Instantiation and we'll call the function itself a "constructor function" since it's responsible for "constructing" a new object.

#### Functional Instantiation

function Animal (name, energy) {

let animal = {}

animal.name = name

animal.energy = energy

animal.eat = function (amount) {

console.log(`${this.name} is eating.`)

this.energy += amount

}

animal.sleep = function (length) {

console.log(`${this.name} is sleeping.`)

this.energy += length

}

animal.play = function (length) {

console.log(`${this.name} is playing.`)

this.energy -= length

}

return animal

}

const leo = Animal('Leo', 7)

const snoop = Animal('Snoop', 10)

Data types

https://www.edureka.co/blog/data-types-in-javascript/#:~:text=JavaScript%20doesn't%20have%20a,instead%2C%20it%20returns%20a%20function.

## ****Difference Between Primitive and Non-Primitive Data Types in JavaScript****

JavaScript primitive data types are data types that refer to a single value.

**E.g. var a =5;**

Here the variable ‘a’ is an integer data type and has a single integer value. The variable ‘a’ refers to a single value in memory. If we want to change the value of a, we would have to assign a new value to a.  *Primitive data types are not mutable.*

When we create a variable, it reserves a space for itself in the memory. The variable ‘a’ has space in memory which holds its value. When we try to change the value of ‘a’ by assigning another value like var a = 6, it doesn’t alter the value of the original a, it just creates a new variable ‘a’ with the new value 6.

We can assign the value of ‘a’ to another variable like this:

**var a1=a;**

Here the variable ‘a1’ is assigned the value of ‘a’, not the address of ‘a’ in memory.

Thus ‘a1’ now holds the same value as ‘a’.

We can compare the two variables ‘a’ and ‘a1’ as the two variables refer to the same value now.

Using the comparison operator will return a Boolean value of ‘true’.

**a===a1 // equals to ‘true’ as ‘===’** checks both the value and type of these two variables are true.

JavaScript non-primitive types are objects. An object holds a reference/address of a single key-value pair or many key-value pairs. Whenever we refer to an object, we refer to an address in memory which contains the key-value pair. If we assign an object ‘object1’ to another object ‘object2’, we are actually assigning the address of ‘object1’ to ‘object2’ instead of the key-value pair which the ‘object1’ contains in memory. Let’s see below”.

**var object1= {a:5, a1:6};**

**var object2 = object1;**

The above statement assigns the address of object2 to object1, and not the value {a:5, a1:6}. Thus with this assignment ‘object1’ and ‘object2’ refer to the same address in memory.

When we compare these two objects, we find that both of them refer to the same address in memory.

object1===object2 //will return true, as both refer to the same address. If we compare two separate objects like below:

**var object1= {a:5, a1:6};**

**var object2 = {a:5, a1:6};**

This statement object1===object2 // would return a false because both the objects refer to two separate addresses in memory. When we compare two objects, we compare their addresses, not their values.

We have seen above in case of primitive data types, that when we assign the variable ‘a’ to variable ‘a1’, the value of ‘a’ is copied to ‘a1’, and not its address which happens in non-primitive data types.

Thus primitive data types refer to a ‘single value’ in an address in memory whereas non-primitive data types refer to the ‘address’ in memory which contains single or multiple key-value pair/s.

**Undefined:**

Undefined data type means a variable that is not defined. The variable is declared but doesn’t contain any value.

|  |  |
| --- | --- |
| 1  2 | var a;  console.log(a); // This will return undefined. |

The variable ‘a’ has been declared but hasn’t been assigned a value yet.  
We can assign a value to a:

|  |  |
| --- | --- |
| 1  2 | a=5;  console.log(a); // This will return 5 |

**Null:**

The null in JavaScript is a data type that is represented by only one value, the ‘null’ itself. A null value means no value.

Something like this:

|  |  |
| --- | --- |
| 1  2 | var a = null;  console.log(a);   // This returns null |

If we check the data type of a using the typeof operator, we get:

|  |  |
| --- | --- |
| 1 | typeof(a);         // This returns object |

This means the type of a null value is an object, not null.

**Symbol:**

The ‘symbol’ data type is new in es6. It is one of the new features of es6. The symbol data type defines a property of an object which is private to the object. It refers to the ‘key’ of the key-value pair of an object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var object1 = {     name: ‘Shalini’,     age: 25,     city: ‘Mumbai’  }  var occupation=Symbol(‘engineer’); |

The function Symbol() is used to create a new symbol. Here we have created a symbol ‘occupation’ with the value ‘engineer’ for the above object ‘object1’.

Every symbol is unique. Two Symbols even with the same key values are not same.

|  |  |
| --- | --- |
| 1  2 | var occupation=Symbol(‘engineer’);  var occupation=Symbol(); |

occupation===occupation // returns false. Thus both the above ‘occupation’ symbols are different. Each one is a unique property of the object.

We cannot create a symbol object using the ‘new’ operator because the Symbol() cannot be used as a constructor.

The string description inside Symbol() function is optional.  
Checking the type of ‘occupation’ symbol:

|  |  |
| --- | --- |
| 1 | typeof(occupation);  // returns symbol |

/ There are 3 ways to create objects.

// By object literal

// By creating instance of Object directly (using new keyword)

// By using an object constructor (using new keyword)

Program: dataTypes.js

**Function:**

JavaScript doesn’t have a function data type but when we find the data type of a function using the typeof operator, we find that it returns a function. This is because a function is an object in JavaScript.

Let’s define a function named a:

|  |  |
| --- | --- |
| 1 | function a(){ } |

Now let’s find the data type of a by using the typeof operator:

|  |  |
| --- | --- |
| 1 | typeof(a); // This will return data type function |

## ****Difference Between Undefined and Null****

These are two different data types of JavaScript. When a variable is declared but has not been assigned a value yet, the data type of that variable is undefined.

**Null** is a data type in JavaScript but it is mainly used as a value that is assigned to a variable. Assigning a null to a variable means that the variable contains no value, similar to the meaning of ‘undefined’ which also means empty value.

The two differences between null and undefined:

* ‘**null**’ can be assigned as a value to a variable whereas undefined can also be assigned as a value to a variable but it is not recommended or a good practice.

A variable with no value simply means it is undefined.

|  |  |
| --- | --- |
| 1  2 | var a;  console.log(a); // returns undefined |

‘a’ doesn’t contain any value, the data type is undefined.

|  |  |
| --- | --- |
| 1  2 | var a=undefined; // possible but not recommended  console.log(a) // returns undefined |

Let’s take another variable:

|  |  |
| --- | --- |
| 1 | var b=null; |

* The data type of a variable which doesn’t contain a value is ‘undefined’, but the data type of a variable which contains a ‘null’ value is ‘object’ instead of ‘null’.

## [Date object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#date_object)

JavaScript does not have a date data type. However, you can use the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

4

There is no "Date Type", [ECMASCript Types](https://262.ecma-international.org/" \l "sec-typeof-operator) are:

1. Undefined
2. Null
3. Boolean
4. String
5. Symbol
6. Number
7. BigInt
8. Object

Note that the value returned by [typeof](https://262.ecma-international.org/" \l "sec-typeof-operator) does not necessarily match the Type of the value, e.g.

1. typeof null returns "object"
2. typeof someFun returns "function", where someFn is a an object that implements an [internal call method](https://262.ecma-international.org/#table-additional-essential-internal-methods-of-function-objects) (i.e. is a function)

ECMAScript has a built–in Date object that is a function that can also be called as a constructor. So:

typeof Date

returns "function" even though it's an Object Type, whereas using the typeof operator on a Date instance:

typeof new Date()

returns "object", as it's also an Object Type. The term "Date object" is commonly used to refer to a Date instance, and "Date constructor" to refer to the built–in Date object/function/constructor.

# Regular expressions

Regular expressions are **patterns used to match character combinations in strings**. In JavaScript, regular expressions are also objects.

## Functions Can Return Other Functions

A function that returns another function has a special name. It's called a higher-order function.

This is the foundation on which closures stand. Here's our first example of a higher-order function.

function getGreeter() {

return function() {

return 'Hi, Jerome!';

};

}

https://www.freecodecamp.org/news/learn-javascript-closures-in-n-minutes/#:~:text=We%20Used%20a%20Closure&text=When%20a%20function%20returns%2C%20its,after%20the%20parent%20passes%20on. ---important

JavaScript doesn't support private variables ([yet](https://github.com/tc39/proposal-class-fields)), but we can use closures!

closures

https://www.w3schools.com/js/js\_function\_closures.asp - important

https://www.javascripttutorial.net/javascript-closure/ --important

## Introduction to JavaScript closures

In JavaScript, a closure is a [function](https://www.javascripttutorial.net/javascript-function/) that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

function getGreeter(name) {

return function() {

return `Hi, ${name}!`;

};

}

## We Used a Closure

The outer function takes name, but the inner function uses it later. This is the power of closures.

When a function returns, its lifecycle is complete. It can no longer perform any work, and its local variables are cleaned up.

Unless it returns another function. If that happens, then the returned function still has access to those outer variables, even after the parent passes on.

IIFE

https://www.freecodecamp.org/news/iife-in-javascript-what/

https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/

A JavaScript immediately invoked function expression is a [function](https://www.javascripttutorial.net/javascript-function/) defined as an expression and executed immediately after creation.

(function(){

//....

})()

IIFE with data privacy

https://javascript.plainenglish.io/4-practical-use-cases-for-iifes-in-javascript-6481dcb0ba7d

Callback functions and Asynchronous programming

[https://medium.com/codingthesmartway-com-blog/async-programm ing-with-javascript-callbacks-promises-and-async-await-980e3f144185](https://medium.com/codingthesmartway-com-blog/async-programming-with-javascript-callbacks-promises-and-async-await-980e3f144185)

<https://medium.com/front-end-weekly/ajax-async-callback-promise-e98f8074ebd7>

Fetch

<https://www.javascripttutorial.net/javascript-fetch-api/>

Arrays

https://www.freecodecamp.org/news/complete-introduction-to-the-most-useful-javascript-array-methods/

spread vs rest

https://www.freecodecamp.org/news/javascript-rest-vs-spread-operators/

JavaScript uses three dots (...) for both the rest and spread operators. But these two operators are not the same.

The main difference between rest and spread is that the rest operator puts the rest of some specific user-supplied values into a JavaScript array. But the spread syntax expands iterables into individual elements.

Map

https://www.javascripttutorial.net/es6/javascript-map/

classes

https://www.javascripttutorial.net/es6/javascript-getters-and-setters/

What is a Promise in JavaScript?

A Promise is a special JavaScript object. It produces a value after an asynchronous (aka, async) operation completes successfully, or an error if it does not complete successfully due to time out, network error, and so on.

Webpack

<https://www.freecodecamp.org/news/an-intro-to-webpack-what-it-is-and-how-to-use-it-8304ecdc3c60/>