JavaScript Variable

Before ES2015, JavaScript had only two types of scope: Global Scope and Function Scope.

Global Scope

Variables declared Globally (outside any function) have Global Scope.

```
var carName = "Volvo";

// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

Global variables can be accessed from anywhere in a JavaScript program.

Function Scope

Variables declared Locally (inside a function) have Function Scope.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Block Scope

Variables declared with the var keyword can not have Block Scope.

Variables declared inside a block {} can be accessed from outside the block.

```
{
  var x = 2;
}
// x CAN be used here
```

Variables declared with the let keyword can have Block Scope.

Variables declared inside a block {} can not be accessed from outside the block

```
{
  let x = 2;
}
// x can NOT be used here
```

Redeclaring Variables

```
var x = 2;        // Allowed
let x = 3;        // Not allowed

{
  var x = 4;    // Allowed
  let x = 5    // Not allowed
}
```

```
let x = 2;        // Allowed
let x = 3;        // Not allowed

{
  let x = 4;    // Allowed
  let x = 5;    // Not allowed
}
```

```
let x = 2;        // Allowed
var x = 3;        // Not allowed

{
  let x = 4;    // Allowed
  var x = 5;    // Not allowed
}
```

```
let x = 2;        // Allowed

{
  let x = 3;    // Allowed
}

{
  let x = 4;    // Allowed
}
```

JavaScript Const

Variables defined with const behave like let variables, except they cannot be reassigned

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

Block Scope

Declaring a variable with const is similar to let when it comes to Block Scope.

The x declared in the block, in this example, is not the same as the x declared outside the block:

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

Constant Objects can Change

You can change the properties of a constant object

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

But you can NOT reassign a constant object

```
const car = {type:"Fiat", model:"500", color:"white"};
car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Constant Arrays can Change

You can change the elements of a constant array

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

But you can NOT reassign a constant array:

```
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

مثال هایی از تغییر مجدد

```
var x = 2;          // Allowed
const x = 2;        // Not allowed
{
  let x = 2;      // Allowed
  const x = 2;    // Not allowed
}
```

```
const x = 2;        // Allowed
const x = 3;        // Not allowed
x = 3;              // Not allowed
var x = 3;          // Not allowed
let x = 3;          // Not allowed

{
  const x = 2;    // Allowed
  const x = 3;    // Not allowed
  x = 3;          // Not allowed
  var x = 3;      // Not allowed
  let x = 3;      // Not allowed
}
```

```
const x = 2;        // Allowed
```

```
{
  const x = 3;   // Allowed
}

{
  const x = 4;   // Allowed
}
```

Arrow Function

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax

Before:

```
hello = function() {
  return "Hello World!";
}
```

With Arrow Function:

```
hello = () => {
  return "Hello World!";
}
```

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

نکته:

اگر در تابع سازنده یک متد وجود داشته باشد که از طریق کلمه کلیدی this بخواهیم به property های تابع سازنده دسترسی پیدا کنیم می بایست از Arrow Functions استفاده کنیم.

```
function Person() {
  this.age = 0;
  setInterval(() => {
    this.age++;
    console.log(this.age);
  }, 5000);
}
let person = new Person(); // 0 1 2 3 4 5 6 7 …
```

اگر بصورت زیر نوشته شود کلمه کلیدی this به property های تابع سازنده دسترسی پیدا نمی کند.

```
function Person() {
  this.age = 0;
  setInterval(function() {
    this.age++;
    console.log(this.age);
  }, 1000);
}
let person = new Person(); // NaN NaN NaN NaN NaN NaN NaN
```

اما در Object ها کاملا برعکس است یعنی اگر در یک Object از طریق کلمه کلیدی this بخواهیم به property های آن Object دسترسی پیدا کنیم می بایست از function استفاده کنیم.

```
let obj = {
  a: 10,
  b: function(){
  console.log(this.a);
}
}
obj.b(); //10
```

Default function parameters

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

```
function multiply(a, b) {
  return a * b
}

multiply(5, 2)  // 10
multiply(5)     // NaN !
```

With default parameters in ES2015, checks in the function body are no longer necessary. Now, you can assign 1 as the default value for b in the function head:

```
function multiply(a, b = 1) {
  return a * b
}
multiply(5, 2)          // 10
multiply(5)             // 5
multiply(5, undefined)  // 5
```

Syntax

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

```
function sum(...theArgs) {
  return theArgs.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum(1, 2, 3));
// expected output: 6
console.log(sum(1, 2, 3, 4));
// expected output: 10
```

Spread syntax

Spread syntax allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

```
function myFunction(x, y, z) { }
const args = [0, 1, 2];
myFunction(...args);
```

Any argument in the argument list can use spread syntax, and the spread syntax can be used multiple times.

```
function myFunction(v, w, x, y, z) { }
const args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

for…of

The for…of statement creates a loop iterating over iterable objects, including: built-in String, Array, array-like objects (e.g., arguments or NodeList), TypedArray, Map, 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.

```
const iterable = [10, 20, 30];
for (const value of iterable) {
  console.log(value);
}
// 10
// 20
// 30
```

You can use let instead of const too, if you reassign the variable inside the block.

```
const iterable = [10, 20, 30];
for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11
// 21
// 31
```

Iterating over a String

```
const iterable = 'boo';
for (const value of iterable) {
  console.log(value);
}
// "b"
// "o"
// "o"
```

Destructuring assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]
```

Array destructuring

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]
```

Basic variable assignment

```
const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

Assignment separate from declaration

```
let a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Default values

```
let a, b;

[a=5, b=7] = [1];
```

```
console.log(a); // 1
console.log(b); // 7
```

Parsing an array returned from a function

```
function f() {
  return [1, 2];
}

let a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
```

Ignoring some returned values

```
function f() {
  return [1, 2, 3];
}

const [a, , b] = f();
console.log(a); // 1
console.log(b); // 3
```

Assigning the rest of an array to a variable

```
const [a, ...b] = [1, 2, 3];
console.log(a); // 1
console.log(b); // [2, 3]
```

Object destructuring

Basic assignment

```
const o = {p: 42, q: true};
const {p, q} = o;

console.log(p); // 42
console.log(q); // true
```

Assignment without declaration

```
let a, b;

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

Assigning to new variable names

```
const o = {p: 42, q: true};
const {p: foo, q: bar} = o;

console.log(foo); // 42
console.log(bar); // true
```

Default values

```
const {a = 10, b = 5} = {a: 3};

console.log(a); // 3
console.log(b); // 5
```

Template literals (Template strings)

Template literals are enclosed by the backtick (` `) (grave accent) character instead of double or single quotes.

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}). The expressions in the placeholders and the text between the backticks (` `) get passed to a function.

Multi-line strings

Any newline characters inserted in the source are part of the template literal.

```
console.log('string text line 1\n' +
'string text line 2');
```

Expression interpolation

In order to embed expressions within normal strings, you would use the following syntax:

```
let a = 5;
let b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// "Fifteen is 15 and
// not 20."
```

Now, with template literals, you are able to make use of the syntactic sugar, making substitutions like this more readable:

```
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Classes

Use the keyword class to create a class, and always add the constructor() method.

The constructor method is called each time the class object is initialized.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
}
mycar = new Car("Ford");
```

Methods

The constructor method is special, it is where you initialize properties, it is called automatically when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not have a constructor method, JavaScript will add an invisible and empty constructor method.

You are also free to make your own methods, the syntax should be familiar:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return "I have a " + this.carname;
  }
}

mycar = new Car("Ford");
document.getElementById("demo").innerHTML = mycar.present();
```

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (any parameters would go inside the parentheses).

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present(x) {
    return x + ", I have a " + this.carname;
  }
}

mycar = new Car("Ford");
document.getElementById("demo").innerHTML = mycar.present("Hello");
```

Static Methods

Static methods are defined on the class itself, and not on the prototype.

That means you cannot call a static method on the object (mycar), but on the class (Car):

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  static hello() {
    return "Hello!!";
  }
}

mycar = new Car("Ford");

//Call 'hello()' on the class Car:
document.getElementById("demo").innerHTML = Car.hello();

//and NOT on the 'mycar' object:
//document.getElementById("demo").innerHTML = mycar.hello();
//this would raise an error.
```

If you want to use the mycar object inside the static method, you can send it as a parameter:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  static hello(x) {
    return "Hello " + x.carname;
  }
}

mycar = new Car("Ford");

document.getElementById("demo").innerHTML = Car.hello(mycar);
//this would raise an error.
```

Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

mycar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = mycar.show();
```

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Getters and Setters

Classes also allows you to use getters and setters.

It can be smart to use getters and setters for your properties, especially if you want to do something special with the value before returning them, or before you set them.

To add getters and setters in the class, use the get and set keywords.

The name of the getter/setter method cannot be the same as the name of the property, in this case carname.

Many programmers use an underscore character _ before the property name to separate the getter/setter from the actual property:

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  }
}

mycar = new Car("Ford");

document.getElementById("demo").innerHTML = mycar.carname;
```

Use a setter to change the carname to "Volvo":

```
mycar = new Car("Ford");
mycar.carname = "Volvo";
document.getElementById("demo").innerHTML = mycar.carname;
```

Objects

New notations in ECMAScript 2015

```
// Shorthand property names (ES2015)
let a = 'foo', b = 42, c = {};
let o = {a, b, c}
// {a: "foo", b: 42, c: {…}}

// Computed property names (ES2015)
let prop = 'foo'
let o = {
  [prop]: 'hey',
  ['b' + 'ar']: 'there'
}
// {foo: "hey", bar: "there"}
```

Symbol

The data type symbol is a primitive data type. The Symbol() function returns a value of type symbol, has static properties that expose several members of built-in objects, has static methods that expose the global symbol registry, and resembles a built-in object class, but is incomplete as a constructor because it does not support the syntax "new Symbol()".

Every symbol value returned from Symbol() is unique. A symbol value may be used as an identifier for object properties; this is the data type's primary purpose, although other use-cases exist, such as enabling opaque data types, or serving as an implementation-supported unique identifier in general.

```
let sym = Symbol('foo')
typeof sym      // "symbol"
```

To create a new primitive symbol, you write Symbol() with an optional string as its description:

```
let sym1 = Symbol()
let sym2 = Symbol('foo')
let sym3 = Symbol('foo')
```

Foo صرفا یک توضیح درباره سمبل است

The above code creates three new symbols. Note that Symbol("foo") does not coerce the string "foo" into a symbol. It creates a new symbol each time:

```
Symbol('foo') === Symbol('foo')  // false
```

در مثال زیر از یک سمبل برای ساخت یک شناسه منحصر به فرد برای یک Object استفاده شده است.

```
let sym = Symbol('foo')

let obj = {
   name: ‘Ali’,
   [sym]: 23
}
```

Symbol.for()

The Symbol.for(key) method searches for existing symbols in a runtime-wide symbol registry with the given key and returns it if found. Otherwise a new symbol gets created in the global symbol registry with this key.

```
let symbol1 = Symbol('fullname')
let symbol2 = Symbol('fullname')
symbol1 == symbol2
//false
```

```
let symbol1 = Symbol.for('fullname')
let symbol2 = Symbol.for('fullname')
symbol1 == symbol2
//true
```

New Array Method

Array.of

The Array.of() method creates a new Array instance from a variable number of arguments, regardless of number or type of the arguments.

The difference between Array.of() and the Array constructor is in the handling of integer arguments: Array.of(7) creates an array with a single element, 7, whereas Array(7) creates an empty array with a length property of 7 (Note: this implies an array of 7 empty slots, not slots with actual undefined values).

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]


Array(7);          // array of 7 empty slots
Array(1, 2, 3);    // [1, 2, 3]
```

Array.from

The Array.from() method creates a new, shallow-copied Array instance from an array-like or iterable object.

```
console.log(Array.from('foo'));
// expected output: Array ["f", "o", "o"]


console.log(Array.from([1, 2, 3], x => x + x));
// expected output: Array [2, 4, 6]
```

Array.find

The find() method returns the value of the first element in the provided array that satisfies the provided testing function.

```
const array1 = [5, 12, 8, 130, 44];

const found = array1.find(element => element > 10);

console.log(found);
// expected output: 12
```

Array.entries

The entries() method returns a new Array Iterator object that contains the key/value pairs for each index in the array.

```
const array1 = ['a', 'b', 'c'];

const iterator1 = array1.entries();

console.log(iterator1.next().value);
// expected output: Array [0, "a"]

console.log(iterator1.next().value);
// expected output: Array [1, "b"]
```

Iterating with index and element

```
const a = ['a', 'b', 'c'];

for (const [index, element] of a.entries())
  console.log(index, element);

// 0 'a'
// 1 'b'
// 2 'c'
```

Using a for…of loop

```
var a = ['a', 'b', 'c'];
var iterator = a.entries();


for (let e of iterator) {
  console.log(e);
}
// [0, 'a']
// [1, 'b']
// [2, 'c']
```

Array.fill

The fill() method changes all elements in an array to a static value, from a start index (default 0) to an end index (default array.length). It returns the modified array.

```
const array1 = [1, 2, 3, 4];


// fill with 0 from position 2 until position 4
console.log(array1.fill(0, 2, 4));
// expected output: [1, 2, 0, 0]


// fill with 5 from position 1
console.log(array1.fill(5, 1));
// expected output: [1, 5, 5, 5]


console.log(array1.fill(6));
// expected output: [6, 6, 6, 6]
```

Array.copyWithin

The copyWithin() method shallow copies part of an array to another location in the same array and returns it without modifying its length.

```
const array1 = ['a', 'b', 'c', 'd', 'e'];

// copy to index 0 the element at index 3
console.log(array1.copyWithin(0, 3, 4));
// expected output: Array ["d", "b", "c", "d", "e"]

// copy to index 1 all elements from index 3 to the end
console.log(array1.copyWithin(1, 3));
// expected output: Array ["d", "d", "e", "d", "e"]
```

نکته: `array1.copyWithin(0, 3, 4)` یعنی از ایندکس ۳ تا ایندکس ۴ (خود ایندکس ۴ محاسبه نمی شود) را کپی و از ایندکس ۰ مقادیر را قرار دهید.

New Number Method

isNaN

The isNaN() function determines whether a value is NaN or not. Note, coercion inside the isNaN function has interesting rules; you may alternatively want to use Number.isNaN(), as defined in ECMAScript 2015.

```
function milliseconds(x) {
  if (isNaN(x)) {
    return 'Not a Number!';
  }
  return x * 1000;
}
console.log(milliseconds('100F'));
// expected output: "Not a Number!"
console.log(milliseconds('0.0314E+2'));
// expected output: 3140
```

isInteger

The Number.isInteger() method determines whether the passed value is an integer.

```
function fits(x, y) {
  if (Number.isInteger(y / x)) {
    return 'Fits!';
  }
  return 'Does NOT fit!';
}

console.log(fits(5, 10));
// expected output: "Fits!"

console.log(fits(5, 11));
// expected output: "Does NOT fit!"
```

Math.sin

The Math.sin() function returns the sine of a number.

```
let num1 = 12.5;
let num2 = -8;
let num3 = 0;

console.log(Math.sign(num1));
// expected output: 1
console.log(Math.sign(num2));
// expected output: -1
console.log(Math.sign(num3));
// expected output: 0
```

Math.trunc

The Math.trunc() function returns the integer part of a number by removing any fractional digits.

```
console.log(Math.trunc(13.37));
// expected output: 13

console.log(Math.trunc(42.84));
// expected output: 42

console.log(Math.trunc(0.123));
// expected output: 0

console.log(Math.trunc(-0.123));
// expected output: -0
```

New String Method

Includes

The includes() method determines whether one string may be found within another string, returning true or false as appropriate.

```
const str = 'To be, or not to be, that is the question.'

console.log(str.includes('To be'))        // true
console.log(str.includes('question'))     // true
console.log(str.includes('nonexistent'))  // false
console.log(str.includes('To be', 1))     // false
console.log(str.includes('TO BE'))        // false
console.log(str.includes(''))             // true
```

```
'Blue Whale'.includes('blue')  // returns false
```

startsWith

The startsWith() method determines whether a string begins with the characters of a specified string, returning true or false as appropriate.

```
const str1 = 'Saturday night plans';

console.log(str1.startsWith('Sat'));
// expected output: true

console.log(str1.startsWith('Sat', 3));
// expected output: false
```

```
//startswith
let str = 'To be, or not to be, that is the question.'

console.log(str.startsWith('To be'))          // true
console.log(str.startsWith('not to be'))      // false
console.log(str.startsWith('not to be', 10))  // true
```

endsWith

The endsWith() method determines whether a string ends with the characters of a specified string, returning true or false as appropriate.

```
const str1 = 'Cats are the best!';

console.log(str1.endsWith('best', 17));
// expected output: true

const str2 = 'Is this a question';

console.log(str2.endsWith('?'));
// expected output: false
```

New Object Method

Assign

The Object.assign() method copies all enumerable own properties from one or more source objects to a target object. It returns the target object.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign({}, target, source);

console.log(target);
// expected output: Object { a: 1, b: 4 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

Iterator

The well-known Symbol.iterator symbol specifies the default iterator for an object. Used by for...of.

```
const iterable1 = {};

iterable1[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

console.log([...iterable1]);
// expected output: Array [1, 2, 3]
```

Generator

The Generator object is returned by a generator function and it conforms to both the iterable protocol and the iterator protocol.

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generator(); // "Generator { }"
```

Example

```
function* infinite() {
    let index = 0;

    while (true) {
        yield index++;
    }
}

const generator = infinite(); // "Generator { }"

console.log(generator.next().value); // 0
console.log(generator.next().value); // 1
console.log(generator.next().value); // 2
// ...
```

Callback function

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name.');
  callback(name);
}

processUserInput(greeting);
```

Promise

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

```
function getData(url) {
  return new Promise((resolve , reject) => {
       const httpRequest = new XMLHttpRequest();
       httpRequest.open("GET", url);
       httpRequest.onreadystatechange = function() {
           console.log(XMLHttpRequest.DONE);
           if(this.readyState == XMLHttpRequest.DONE) {
               if(this.status == 200) {
                   resolve(this.responseText)
               } else if(this.status == 404) {
                   reject("data not found")
               } else {
                   reject("something goes wrong")
               }
           }
       }
       httpRequest.send();
  })
}
function paresToJson(dataText) {
   return new Promise((resolve , reject) => {
       setTimeout(() => {
           try {
               resolve(JSON.parse(dataText));
           } catch (error) {
               reject(error);
           }
       }, 2000);
   })
}
getData("https://jsonplaceholder.typicode.com/todos")
   .then(data => paresToJson(data) )
   .then((json) => {
       console.log(json)
   })
   .catch(err => console.log(err))
```