**Basic JavaScript - JavaScript for Beginners -** [Fahad Ahmad](https://fahadahmd-bs.medium.com/?source=post_page-----5bff43a0570b--------------------------------) May 15, 2022

**What are the different data types present in JavaScript?**

To know the type of a JavaScript variable, we can use the typeof operator.

**Primitive types** :

**String** — It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

Example :

var str = "Vivek Singh Bisht"; //using double quotes   
var str2 = 'John Doe'; //using single quotes

**Number** — It represents a number and can be written with or without decimals.

Example :

var x = 3; //without decimal   
var y = 3.6; //with decimal

**BigInt**— This data type is used to store numbers that are above the limitation of the Number data type. It can store large integers and is represented by adding “n” to an integer literal.

Example :

var bigInteger = 234567890123456789012345678901234567890;

**Boolean** — It represents a logical entity and can have only two values: true or false. Booleans are generally used for conditional testing.

Example :

var a = 2;   
var b = 3;   
var c = 2; (a == b) // returns false(a == c) //returns true

**Undefined** — When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

Example :

var x; // value of x is undefined   
var y = undefined; // we can also set the value of a variable as undefined

**Null** — It represents a non-existent or invalid value.

Example :

var z = null;

**Symbol**— It is a new data type introduced in the ES6 version of JavaScript. It is used to store an anonymous and unique value.

Example :

var symbol1 = Symbol('symbol');

typeof of primitive types :

typeof "John Doe" // Returns "string"   
typeof 3.14 // Returns "number"   
typeof true // Returns "boolean"   
typeof 234567890123456789012345678901234567890n // Returns bigint typeof undefined // Returns "undefined"   
typeof null // Returns "object" (kind of a bug in JavaScript)   
typeof Symbol('symbol') // Returns Symbol

**Non-primitive types**

Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used. Object — Used to store collection of data.

Example:

// Collection of data in key-value pairs   
var obj1 = {   
 x: 43,   
 y: "Hello world!",   
 z: function(){   
 return this.x;   
 }   
} // Collection of data as an ordered list   
var array1 = [5, "Hello", true, 4.1];

**Explain Hoisting in JavaScript.**

Hoisting is a default behaviour of JavaScript where all the variable and function declarations are moved on top.

Diagram

Description automatically generated with low confidence

This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.

Example 1:

hoistedVariable = 3;   
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is init ialized   
var hoistedVariable;

Example 2:

hoistedFunction(); // Outputs " Hello world! " even when the function is declared after calling   
function hoistedFunction(){   
 console.log(" Hello world! ");   
}

Example 3:

// Hoisting takes place in the local scope as well   
function doSomething(){   
 x = 33;   
 console.log(x);   
 var x;   
}

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

**Note** — Variable initializations are not hoisted, only variable declarations are hoisted:

var x;   
console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted   
x = 23;

**Note**— To avoid hoisting, you can run JavaScript in strict mode by using “use strict” on top of the code:

"use strict";   
x = 23; // Gives an error since 'x' is not declared  
var x;

**Difference between “ == “ and “ === “ operators.**

Both are comparison operators. The difference between both the operators is that,“==” is used to compare values whereas, “ === “ is used to compare both value and types.

Example:

var x = 2;   
var y = "2";   
(x == y) // Returns true since the value of both x and y is the same (x === y) // Returns false since the typeof x is "number" and typeof y is "string"

**Explain Implicit Type Coercion in JavaScript.**

Implicit type coercion in JavaScript is automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

**String coercion**

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type. Example 1:

var x = 3;   
var y = "3";   
x + y // Returns "33"

Example 2:

var x = 24;   
var y = "Hello";   
x + y // Returns "24Hello";

**Note** — ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string:

var name = "Vivek";   
var surname = " Bisht";   
name + surname // Returns "Vivek Bisht"

Let’s understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types ( one being a number type and the other being a string type ) , it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

**Note**— Type coercion also takes place when using the ‘ — ‘ operator, but the difference while using ‘ — ‘ operator is that, a string is converted to a number and then subtraction takes place.

var x = 3;   
Var y = "3";   
x - y //Returns 0 since the variable y (string type) is converted to a number type

**Boolean Coercion**

Boolean coercion takes place when using logical operators, ternary operators, if statements and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.

Truthy values are those which will be converted (coerced) to true . Falsy values are those which will be converted to false . All values except false, 0, 0n, -0, “”, null, undefined and NaN are truthy values.

**If statements:**

Example:

var x = 0;   
var y = 23;   
   
if(x) { console.log(x) } // The code inside this block will not run since the value of x is 0 (Falsy) if(y) { console.log(y) } // The code inside this block will run since the value of y is 23 ( Truthy)

**Logical operators:**

Logical operators in JavaScript, unlike operators in other programming languages, do not return true or false. They always return one of the operands.

**OR ( | | ) operator**— If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

**AND ( && ) operator** — If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

Example:

var x = 220;   
var y = "Hello";   
var z = undefined;   
 x | | y // Returns 220 since the first value is truthy   
 x | | z // Returns 220 since the first value is truthy   
 x && y // Returns "Hello" since both the values are truthy   
 y && z // Returns undefined since the second value is falsy   
if( x && y ){   
 console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)   
}   
if( x || z ){ console.log("Code runs"); // This block runs because x || y returns 220(Truthy)   
}

**Equality Coercion**

Equality coercion takes place when using ‘ == ‘ operator. As we have stated before

**The ‘ == ‘ operator compares values and not types**

While the above statement is a simple way to explain == operator, it’s not completely true The reality is that while using the ‘==’ operator, coercion takes place. The ‘==’ operator, converts both the operands to the same type and then compares them.

Example:

var a = 12;   
var b = "12";   
a == b // Returns true because both 'a' and 'b' are converted to the same type and then co mpared. Hence the operands are equal.

Coercion does not take place when using the ‘===’ operator. Both operands are not converted to the same type in the case of ‘===’ operator.

Example:

var a = 226;   
var b = "226";   
a === b // Returns false because coercion does not take place and the operands are of dif ferent types. Hence they are not equal.

**Is JavaScript a statically typed or a dynamically typed language?**

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during run-time in contrast to statically typed language, where the type of a variable is checked during compile-time.

Graphical user interface

Description automatically generated with medium confidence

Since JavaScript is a loosely(dynamically) typed language, variables in JS are not associated with any type. A variable can hold the value of any data type. For example, a variable which is assigned a number type can be converted to a string type:

var a = 23;   
var a = "Hello World!";

**What is NaN property in JavaScript?**

NaN property represents “**Not-a-Number**” value. It indicates a value which is not a legal number.

typeof of a **NaN** will return a **Number**.

To check if a value is NaN, we use the isNaN() function,

**Note**- isNaN() function converts the given value to a Number type, and then equates to NaN

isNaN("Hello") // Returns true isNaN(345) // Returns false isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a nu mber)   
isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)   
isNaN(false) // Returns false   
isNaN(undefined) // Returns true

**Explain passed by value and passed by reference.**

**In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.**

For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

var x = 2;

In the above example, we created a variable x and assigned it a value “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non primitive data types,

**Assign operator dealing with primitive types:**

Diagram

Description automatically generated

var y = 234;   
var z = y;

In the above example, assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address.

Therefore, variable z is not pointing to the location of variable y, instead it is pointing to a new location in the memory.

var y = #8454; // y pointing to address of the value 234   
var z = y;   
var z = #5411; // z pointing to a completely new address of the value 234// Changing the value of y   
y = 23;   
console.log(z); // Returns 234, since z points to a new address in the memory so change s in y will not effect z

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

**Assign operator dealing with non-primitive types:**

Graphical user interface, diagram, text, application

Description automatically generated

var obj = { name: "Vivek", surname: "Bisht" };   
var obj2 = obj;

In the above example, the assign operator, directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

var obj = #8711; // obj pointing to address of { name: "Vivek", surname: "Bisht" }  
var obj2 = obj;   
var obj2 = #8711; // obj2 pointing to the same address   
// changing the value of obj1   
obj1.name = "Akki";   
console.log(obj2);   
// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same address.

From the above example, we can see that while passing non-primitive data types, the assign operator directly passes the address (reference). Therefore, non-primitive data types are always **passed by reference.**

**What is an Immediately Invoked Function in JavaScript?**

An Immediately Invoked Function ( known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.

Syntax of IIFE :

(function(){   
// Do something;   
})();

To understand IIFE, we need to understand the two sets of parentheses which are added while creating an IIFE :

First set of parenthesis:

(function (){   
 //Do something;   
})

While executing JavaScript code, whenever the compiler sees the word “function”, it assumes that we are declaring a function in the code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

function() {   
 //Do something;   
} // Compiler gives an error since the syntax of declaring a function is wrong in the code abov e.

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it’s a function expression.

Second set of parenthesis:

(function (){   
 //Do something;   
})();

From the definition of an IIFE, we know that our code should run as soon as it is defined. A function runs only when it is invoked. If we do not invoke the function, the function declaration is returned:

(function (){   
// Do something;   
}) // Returns the function declaration

Therefore to invoke the function, we use the second set of parenthesis. .

**Explain Higher Order Functions in JavaScript.**

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. Higher order functions are a result of functions being first-class citizens in JavaScript.

Examples of higher order functions:

function higherOrder(fn) {   
 fn();   
}   
higherOrder(function() { console.log("Hello world") });function higherOrder2() {   
 return function() {   
 return "Do something";   
 }   
}   
var x = higherOrder2();   
x() // Returns "Do something"

**Explain “this” keyword.**

The “this” keyword refers to the object that the function is a property of.

The value of “this” keyword will always depend on the object that is invoking the function.

Confused? Let’s understand the above statements by examples:

function doSomething() {   
 console.log(this);   
}   
doSomething();

What do you think the output of the above code will be?

Note — Observe the line where we are invoking the function. Check the definition again:

**The “this” keyword refers to the object that the function is a property of.**

In the above code, function is a property of which object? Since the function is invoked in the global context, the function is a property of the global object.

Therefore, the output of the above code will be the global object. Since we ran the above code inside the browser, the global object is the window object.

Example 2:

var obj = {   
 name: "vivek",   
 getName: function(){   
 console.log(this.name);   
 }   
} obj.getName();

In the above code, at the time of invocation, the getName function is a property of the object obj , therefore, the this keyword will refer to the object obj , and hence the output will be “vivek”.

Example 3:

var obj = {   
 name: "vivek",   
 getName: function(){   
 console.log(this.name);   
 }   
}   
var getName = obj.getName;   
var obj2 = {  
 name:"akshay",   
 getName   
};   
obj2.getName();

Can you guess the output here?

The output will be “akshay”.

Although the getName function is declared inside the object obj , at the time of invocation, getName() is a property of obj2 , therefore the “this” keyword will refer to obj2 .

The silly way to understanding the this keyword is, whenever the function is invoked, check the object before the dot . The value of this . keyword will always be the object before the dot . If there is no object before the dot like in example1, the value of this keyword will be the global object.

Example 4:

var obj1 = {   
 address : "Mumbai,India",   
 getAddress: function(){   
 console.log(this.address);   
}   
}  
 var getAddress = obj1.getAddress;   
 var obj2 = {name:"akshay"};   
 obj2.getAddress();

Can you guess the output?

The output will be an error.

Although in the code above, the this keyword refers to the object obj2 , obj2 does not have the property “address”‘, hence the getAddress function throws an error.

**Explain Scope and Scope Chain in JavaScript.**

Scope in JS, determines the accessibility of variables and functions at various parts in one’s code.

In general terms, the scope will let us know at a given part of code, what are the variables and functions that we can or cannot access. There are three types of scopes in JS:

1. Global scope
2. Local or Function Scope
3. Block Scope

**Global Scope:**

Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

var globalVariable = "Hello world";   
function sendMessage(){   
 return globalVariable; // can access globalVariable since it's written in global space   
} function sendMessage2(){   
 return sendMessage(); // Can access sendMessage function since it's written in global sp ace   
 }   
sendMessage2(); // Returns “Hello world”

**Function Scope**

Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

function awesomeFunction(){   
 var a = 2;   
 var multiplyBy2 = function(){   
 console.log(a\*2); // Can access variable "a" since a and multiplyBy2 both are written i nside the same function   
 }   
}   
console.log(a); // Throws reference error since a is written in local scope and cannot be ac cessed outside multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope

**Block Scope**

Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope.

Block scope tells us that any variable declared inside a block { }, can be accessed only inside that block and cannot be accessed outside of it.

{   
 let x = 45;   
}   
console.log(x); // Gives reference error since x cannot be accessed outside of the block for(let i=0; i

**Scope Chain**

JavaScript engine also uses Scope to find variables. Let’s understand that using an example:

var y = 24;   
function favFunction(){   
 var x = 667;   
 var anotherFavFunction = function(){   
 console.log(x); // Does not find x inside anotherFavFunction, so looks for variable insi de favFunction, outputs 667   
}  
 var yetAnotherFavFunction = function(){   
 console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable i nside favFunction and does not find it, so looks for variable in global scope, finds it and outp uts 24   
 } anotherFavFunction();   
 yetAnotherFavFunction(); }   
favFunction();

As you can see in the code above, if the JavaScript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.

If the variable is not found in the global space as well, reference error is thrown.

**What is recursion in a programming language?**

Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it arrives at a result.

function add(number) {  
if (number <= 0) {  
return 0;  
} else {  
return number + add(number - 1);  
}  
}  
add(3) => 3 + add(2)  
3 + 2 + add(1)  
3 + 2 + 1 + add(0)  
3 + 2 + 1 + 0 = 6

Example of a recursive function:

The following function calculates the sum of all the elements in an array by using recursion:

function computeSum(arr){  
 if(arr.length === 1){  
 return arr[0];  
}  
 else{  
 return arr.pop() + computeSum(arr);  
 }  
}  
computeSum([7, 8, 9, 99]); // Returns 123

**What is DOM?**

DOM stands for Document Object Model.

DOM is a programming interface for HTML and XML documents.

When the browser tries to render a HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document. Example of how HTML code gets converted to DOM:

Diagram

Description automatically generated

**Differences between declaring variables using var, let and const.**

Before the ES6 version of JavaScript, only the keyword var was used to declare variables. With the ES6 Version, keywords let and const were introduced to declare variables.

Table

Description automatically generated

Let’s understand the differences with examples:

Let’s understand the differences with examples:

var variable1 = 23;let variable2 = 89;function catchValues(){console.log(variable1);console.log(variable2);// Both the variables can be accessed anywhere since they are declared in the global scope}window.variable1; // Returns the value 23  
window.variable2; // Returns undefined

The variables declared with the let keyword in the global scope behave just like the variable declared with the var keyword in the global scope. Variables declared in the global scope with var and let keywords can be accessed from anywhere in the code.

But, there is one difference! Variables that are declared with the var keyword in the global scope are added to the window/global object. Therefore, they can be accessed using window.variableName. Whereas, the variables declared with the let keyword are not added to the global object, therefore, trying to access such variables using window.variableName results in an error.

**var vs let in functional scope**

function varVsLetFunction(){let awesomeCar1 = "Audi";var awesomeCar2 = "Mercedes";}console.log(awesomeCar1); // Throws an errorconsole.log(awesomeCar2); // Throws an error

Variables declared in a functional/local scope using var and let keywords behave exactly the same, meaning , they cannot be accessed from outside of the scope.

{var variable3 = [1, 2, 3, 4];}console.log(variable3); // Outputs [1,2,3,4]{let variable4 = [6, 55, -1, 2];}console.log(variable4); // Throws errorfor(let i = 0; i < 2; i++){//Do something}console.log(i); // Throws errorfor(var j = 0; j < 2; i++){// Do something}console.log(j) // Outputs 2

In JavaScript, a block means the code written inside the curly braces {} . Variables declared with var keyword do not have block scope. It means a variable declared in block scope {} with the var keyword is the same as declaring the variable in the global scope. Variables declared with let keyword inside the block scope cannot be accessed from outside of the block. Const keyword Variables with the const keyword behave exactly like a variable declared with the let keyword with only one difference, any variable declared with the const keyword cannot be reassigned.

Example:

const x = {name:"Vivek"};x = {address: "India"}; // Throws an errorx.name = "Nikhil"; // No error is thrownconst y = 23;y = 44; // Throws an error

In the code above, although we can change the value of a property inside the variable declared with const keyword, we cannot completely reassign the variable itself.

**Guess the outputs of the following codes:**

// Code 1:function func1(){setTimeout(()=>{console.log(x);console.log(y);},3000);var x = 2;let y = 12;}func1();// Code 2:function func2(){for(var i = 0; i < 3; i++){setTimeout(()=> console.log(i),2000);}}func2();// Code 3:(function(){setTimeout(()=> console.log(1),2000);console.log(2);setTimeout(()=> console.log(3),0);console.log(4);})();

Answers:

Code 1 — Outputs 2 and 12 . Since, even though let variables are not hoisted, due to async nature of JavaScript, the complete function code runs before the setTimeout function. Therefore, it has access to both x and y.

Code 2 — Outputs 3 , three times since variable declared with var keyword does not have block scope. Also, inside the for loop, the variable i is incremented first and then checked.

Code 3 — Output in the following order:

2  
4  
3  
1 // After two seconds

Even though the second timeout function has a waiting time of zero seconds, the JavaScript engine always evaluates the setTimeout function using the Web API and therefore, the complete function executes before the setTimeout function can execute.

**Guess the outputs of the following code:**

// Code 1:let x= {}, y = {name:"Ronny"},z = {name:"John"};x[y] = {name:"Vivek"};x[z] = {name:"Akki"};console.log(x[y]);// Code 2:function runFunc(){console.log("1" + 1);console.log("A" - 1);console.log(2 + "-2" + "2");console.log("Hello" - "World" + 78);console.log("Hello"+ "78");}runFunc();// Code 3:let a = 0;let b = false;   
console.log((a == b));   
console.log((a === b));

Answers:

Code 1 — Output will be {name: “Akki”}. Adding objects as properties of another object should be done carefully. Writing x[y] = {name:”Vivek”} , is same as writing x[‘object Object’] = {name:”Vivek”} , While setting a property of an object, JavaScript coerces the parameter into a string. Therefore, since y is an object, it will be converted to ‘object Object’. Both x[y] and x[z] are referencing the same property. Code 2 — Outputs in the following order:

11   
Nan   
2-22   
NaN   
Hello78

**Code 3** — Output in the following order due to equality coercion:

true   
false

**Guess the output of the following code:**

var x = 23;(function(){var x = 43;(function random(){x++;console.log(x);var x = 21;})();})();

Answer:

Output is NaN. random() function has functional scope, since x is declared and hoisted in the functional scope.

Rewriting the random function will give a better idea about the output:

function random(){var x; // x is hoistedx++; // x is not a number since it is not initialized yetconsole.log(x); // Outputs NaNx = 21; // Initialization of x}

**Guess the outputs of the following code:**

**Note** — Code 2 and Code 3 require you to modify the code, instead of guessing the output.

// Code 1(function(a){return (function(){console.log(a);a = 23;})()})(45);// Code 2// Each time bigFunc is called, an array of size 700 is being created,// Modify the code so that we don't create the same array again and againfunction bigFunc(element){let newArray = new Array(700).fill('♥');return newArray[element];

Answers — Code 1 — Outputs 45 . Even though a is defined in the outer function, due to closure the inner functions have access to it. Code 2 — This code can be modified by using closures,

function bigFunc(){let newArray = new Array(700).fill('♥');return (element) => newArray[element];}let getElement = bigFunc(); // Array is created only oncegetElement(599);getElement(670);

Code 3 — Can be modified in two ways:

Using let keyword:

function randomFunc(){for(let i = 0; i < 2; i++){setTimeout(()=> console.log(i),1000);}  
}  
randomFunc();

Using closure:

function randomFunc(){for(var i = 0; i < 2; i++){(function(i){setTimeout(()=>console.log(i),1000);})(i);}}randomFunc();

**Implement a function that returns an updated array with r right rotations on an array of integers a .**

Example:

Given the following array:

[2,3,4,5,7]

Perform 3 right rotations:

First rotation : [7,2,3,4,5] , Second rotation : [5,7,2,3,4] and, Third rotation: [4,5,7,2,3]

return [4,5,7,2,3]

Answer:

function rotateRight(arr,rotations){if(rotations == 0) return arr;for(let i = 0; i < rotations;i++){let element = arr.pop();arr.unshift(element);}return arr;}rotateRight([2, 3, 4, 5, 7], 3); // Return [4,5,7,2,3]rotateRight([44, 1, 22, 111], 5); // Returns [111,44,1,22]

Here is some basic concept of JavaScript with a Practice example. I hope you like this article like, share, and give any suggestions.