Step by Step Guide of Javascript

References: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction>

# What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.).

There are also more advanced server side versions of JavaScript such as Node.js.

* Client-side JavaScript integrate with HTML and respond to events. Such as Mouse click, form input.
* Server-side JavaScript allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on.
* JavaScript is not strictly typed language.
* It supports a runtime system based on a small number of data types representing numeric, Boolean and string values.

# Grammer and Types

* Case-sensitive and uses Unicode character set. For example, the word Früh (which means "early" in German) could be used as a variable name.

let Früh = "foobar"

But, the variable früh is not the same as Früh because JavaScript is case sensitive.

* Statement are separated by semi-colon.
* Comments

// a one line comment

/\* this is a longer,

\* multi-line comment

\*/

/\* You can't, however, /\* nest comments \*/ SyntaxError \*/

Comments behave like whitespace, and are discarded during script execution.

1. HashBang comments :

 #!/usr/bin/env node

This is called HashBang comments syntax, and is a special comments used to specify the path to a particular JavaScript engine that should execute the script.

* Declaration: JavaScript has three kinds of variable declarations.

1. var : Declares a variable, optionally initializing it to a value.

var x =15;

1. let: Declares a block-scoped, local variable, optionally initializing it to a value.

let y = 10;

1. const : Declares a block-scoped, read-only named constant.

const z = 10;

* Variable Hoisting

Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception.

This concept is known as **hoisting.** Variables in JavaScript are, in a sense, "hoisted" (or "lifted") to the top of the function or statement. However, variables that are hoisted return a value of undefined. So even if you declare and initialize after you use or refer to this variable, it still returns undefined.

// code

var a= 10;

console.log(a , b)  //10 undefined

var b =20;

when we declare any variable then JavaScript lifted it up and declare so, we can use it before initialisation and it return is undefined before initialisation.

// compile code

var a;

var b;

a= 10;

console.log(a , b)  //10 undefined

b =20;

* Function Hoisting

In the case of functions, only function declarations are hoisted—but not the function expressions.

foo(); // "bar"

function foo() {

  console.log('bar');

}

/\* Function expression \*/

baz(); // TypeError: baz is not a function

var baz = function() {

  console.log('bar2');

};

* Constants

You can create a read-only, named constant with the [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) keyword.

A constant cannot change value through assignment or be re-declared while the script is running. It must be initialized to a value.

// Example 1

const PI =3.14;

console.log(PI) // 3.14

const f = 10;

function f(){} // SyntaxError: Identifier 'f' has already been declared

However, the properties of objects assigned to constants are not protected

const objects = {key : 'valye'}

console.log(objects)    //{ key: 'valye' }

objects.key = 'new value'

console.log(objects)    // { key: 'new value' }

Also, the contents of an array are not protected, so the following statement is executed without problems.

const MY\_ARRAY = ['HTML','CSS'];

MY\_ARRAY.push('JAVASCRIPT');

console.log(MY\_ARRAY); //logs ['HTML','CSS','JAVASCRIPT'];

## Data Structure and Types

1. The latest ECMAScript standard defines eight data types:

* [Boolean](https://developer.mozilla.org/en-US/docs/Glossary/Boolean). true and false.
* [null](https://developer.mozilla.org/en-US/docs/Glossary/null). A special keyword denoting a null value. (Because JavaScript is case-sensitive, null is *not*the same as Null, NULL, or any other variant.)
* [undefined](https://developer.mozilla.org/en-US/docs/Glossary/undefined). A top-level property whose value is not defined.
* [Number](https://developer.mozilla.org/en-US/docs/Glossary/Number). An integer or floating point number. For example: 42 or 3.14159.
* [BigInt](https://developer.mozilla.org/en-US/docs/Glossary/BigInt). An integer with arbitrary precision. For example: 9007199254740992n.
* [String](https://developer.mozilla.org/en-US/docs/Glossary/String). A sequence of characters that represent a text value. For example: "Howdy"
* [Symbol](https://developer.mozilla.org/en-US/docs/Glossary/Symbol) (new in ECMAScript 2015). A data type whose instances are unique and immutable.

1. and [Object](https://developer.mozilla.org/en-US/docs/Glossary/Object)

# Control Flow and Error Handling

## Block Statement

The most basic statement is a block statement, which is used to group statements. The block is delimited by a pair of curly brackets:

Block statements are commonly used with control flow statements (if, for, while)

// Block scope Declaration

var x = 1;

while (x <= 10) {

    console.log(x)

    x++;

}

## Conditional Statement

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: if...else and switch.

### If-else statement

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false.

if (condition\_1) {

statement\_1;

} else if (condition\_2) {

  statement\_2;

} else if (condition\_n) {

  statement\_n;

} else {

  statement\_last;

}

### Best practises

In general, it's good practice to always use block statements—especially when nesting if statements:

if (condition) {

statement\_1\_runs\_if\_condition\_is\_true;

statement\_2\_runs\_if\_condition\_is\_true;

} else {

statement\_3\_runs\_if\_condition\_is\_false;

statement\_4\_runs\_if\_condition\_is\_false;

}

It's unwise to use simple assignments in a conditional expression, because the assignment can be confused with equality when glancing over the code.

For example, do not write code like this:

// Prone to being misread as "x == y"

if (x = y) {

/\* statements here \*/

}

let .log(a,b) // 9 , 9

if(a==b){ // true

    console.log("in second")

}

### Falsy values

The following values evaluate to false

* false
* undefined
* null
* 0
* Null
* the empty string (“”)

All other values—including all objects—evaluate to true when passed to a conditional statement.

var b = new Boolean(false); // it is object not prmitive

console.log(b)

if(b){ // true

    console.log("in")

}

if(b == true){ // true  // object data type

    console.log("in true")  // this statement will get execute

}

var d = false;

if(d == true){ // prmitive data type

    console.log("iiiiiii") // this statement will not get execute

}

 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Boolean

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean>

### Switch Statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

A switch statement looks like this:

switch (expression) {

case label\_1:

statements\_1

[break;]

case label\_2:

statements\_2

[break;]

…

default:

statements\_def

[break;]

}

The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.

if no matching label is found, the program looks for the optional default clause:

// if default is defined

switch (new Date().getDay()) {

    case 0:

        day = "Sunday";

        break;

    case 1:

        day = "Monday";

        break;

    case 2:

        day = "Tuesday";

        break;

    case 3:

        day = "Wednesday";

        break;

    case 4:

        day = "Thursday";

        break;

    case 5:

        day = "Friday";

        break;

    default:

        day = "Saturday"

}

console.log(day)

If no default clause is found, the program resumes execution at the statement following the end of switch.

// when default conditon is not given

var gender = "Transgender";

var category;

switch(gender){

    case "Male":

    category = "Male";

    break;

    case "Female":

    category = "Female";

    break;

}

console.log(category); // undefined

// when default conditon is not in last

var gender = "Transgender";

var category;

switch(gender){

    case "Male":

    category = "Male";

    break;

    default :

    category = "Transgender";

    case "Female":

    category = "Female";

    break;

}

console.log(category); // Transgender

### Try Catch Statement

If any statement within the try block (or in a function called from within the try block) throws an exception, control *immediately* shifts to the catch block. If no exception is thrown in the try block, the catch block is skipped. The finally block executes after the try and catch blocks execute but before the statements following the try...catch statement.

function getMonthName(mo) {

    mo = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)

    let months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',

                  'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

    if (months[mo]) {

      return months[mo];

    } else {

      throw 'InvalidMonthNo'; // throw keyword is used here

    }

  }

  try { // statements to try

    monthName = getMonthName(myMonth); // function could throw exception

  }

  catch (e) {

    monthName = 'unknown';

    console.log(e) // pass exception object to error handler (i.e. your own function)

  }

getMonthName(14)  //throw 'InvalidMonthNo'; // throw keyword is used here

### Catch Block

You can use a catch block to handle all exceptions that may be generated in the try block.

catch (catchID) {

statements

}

try {

    throw 'testing'

}catch(Error) {

    console.error(Error)

}

### Best Practise for catch Block

When logging errors to the console inside a catch block, using console.error() rather than console.log() is advised for debugging. It formats the message as an error, and adds it to the list of error messages generated by the page.

### Finally Block

The finally block contains statements to be executed after the try and catch blocks execute. Additionally, the finally block executes before the code that follows the try…catch…finally statement

It is also important to note that the finally block will execute whether or not an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

function f() {

try {

console.log(0);

throw 'bogus';

} catch(e) {

console.log(1);

return true; // this return statement is suspended

// until finally block has completed

console.log(2); // not reachable

} finally {

console.log(3);

return false; // overwrites the previous "return"

console.log(4); // not reachable

}

// "return false" is executed now

console.log(5); // not reachable

}

console.log(f());

# Loops and Iteration

Loops offer a quick and easy way to do something repeatedly.

There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

### for statement

A [for](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for) loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C for loop.

A for statement looks as follows:

for ([initialExpression]; [condition]; [incrementExpression])

statement

for(let i=0 ; i<10;i++){

    console.log(i);

}

### do…while statement

The [do...while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/do...while) statement repeats until a specified condition evaluates to false.

A do...while statement looks as follows:

*statement* is always executed once before the condition is checked.

do

statement

while (condition);

let i = 0;

do {

i += 1;

console.log(i);

} while (i < 5);

### While Statement

A [while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/while) statement executes its statements as long as a specified condition evaluates to true. A while statement looks as follows:

The condition test occurs before statement in the loop is executed. If the condition returns true, statement is executed and the *condition* is tested again. If the condition returns false, execution stops, and control is passed to the statement following while.

while (condition)

statement

let n = 0;

let x = 0;

while (n < 3) {

  n++;

  x += n;

  console.log(x);

}

### Labeld Statement

A [label](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/label) provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

label :

statement

let i =0;

markLoop : while(true){

    if(i==5){

        break markLoop;

    }

    else {

        i++;

    }

    console.log(i)

}

### Break Statement

Use the [break](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/break) statement to terminate a loop, switch, or in conjunction with a labeled statement.

* When you use break without a label, it terminates the innermost enclosing while, do-while, for, or switch immediately and transfers control to the following statement.
* When you use break with a label, it terminates the specified labeled statement.

break;

break [label];

Example 1:

let theValue = 11;

let a = [10,11,22,12]

for (let i = 0; i < a.length; i++) {

    console.log(a[i])

    if (a[i] === theValue) {

      break;

    }

  }

Example 2:

let x = 0;

let z = 0;

labelCancelLoops: while (true) {

  console.log('Outer loops: ' + x);

  x += 1;

  z = 1;

  while (true) {

    console.log('Inner loops: ' + z);

    z += 1;

    if (z === 4 && x === 4) {

      break labelCancelLoops;

    } else if (z === 4) {

      break;

    }

  }

}

// Outer loops: 0

// Inner loops: 1

// Inner loops: 2

// Inner loops: 3   // break

// Outer loops: 1

// Inner loops: 1

// Inner loops: 2

// Inner loops: 3   // break

// Outer loops: 2

// Inner loops: 1

// Inner loops: 2

// Inner loops: 3   // break

// Outer loops: 3

// Inner loops: 1

// Inner loops: 2

// Inner loops: 3   // break labelCancelLoops

### Continue Statement

The [continue](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/continue) statement can be used to restart a while, do-while, for, or label statement.

* When you use continue without a label, it terminates the current iteration of the innermost enclosing while, do-while, or for statement and continues execution of the loop with the next iteration. In contrast to the break statement, continue does not terminate the execution of the loop entirely. In a while loop, it jumps back to the condition. In a for loop, it jumps to the increment-expression.
* When you use continue with a label, it applies to the looping statement identified with that label

continue [label];

Example 1:

let i = 0;

let n = 0;

while (i < 5) {

  i++;

  if (i === 3) {

    continue;

  }

  console.log(i)

}

// 1

// 2

// 4

// 5

### For. . . in statement

The [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for...in) statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements. A for...in statement looks as follows

for (variable in object)

statement

let string = "mohit jain"

// for in loop

for(let i in string){   // here i is index

console.log(string[i])  // print character using index

}

### for…of Statement

The [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for...of) statement creates a loop Iterating over [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) (including [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), [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), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/functions/arguments) object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

for (variable of object)

statement

The following example shows the difference between a for...of loop and a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for...in) loop. While for...in iterates over property names, for...of iterates over property values:

const arr = [3, 5, 7];

arr.foo = 'hello';

for (let i in arr) {

   console.log(i); // logs "0", "1", "2", "foo"

}

What to Study Later:

Functions: <https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

Hashbang comments:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Hashbang_comments>

All below Literals:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Array_literals>