JAVASCRIPT

10 MINUTES CHEAT SHEET TO CRACK JAVASCRIPT TECHNICAL INTERVIEWS

LEVEL 1

JOHN THARAKAN

Preface

There are a hundred books on JavaScript and I don't want to write another one into that pile, but there is a dearth of a revision material that a developer could skim through maybe right before an interview. This book addresses just that. A ready recon-er, a one stop look to refresh your fundamentals on your kindle, while waiting for your turn to be interviewed at a company.

These are very fundamental concepts any JavaScript Developer should understand and are usually the front run questions any interviewer would ask in a level 1 technical interviews of JavaScript. This is documented as crisp as possible with the main intension to only review the basics in less than 10 minutes. No JavaScript technical interview is complete without at least a few of these fundamental concepts or different flavor questions on these same concepts.

1. What is JavaScript? Is it Case Sensitive?

JavaScript is a high-level, dynamic, untyped, interpreted Programming language; It is prototype-based with first-class functions, making JavaScript a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It supports client side scripting as well as server side scripting (NodeJS). JavaScript can provide dynamic interactivity on websites when used along with HTML and CSS, the fundamental building blocks of any webpage.

Yes, it is a case sensitive language.

```
var a=10; var A=20;          /* a and A points to
two different memory    locations. */

console.log(a); // will print 10

console.log(A); // will print 20
```

var is a valid identifier to allocate memories and Var is not understood by the JavaScript Engine console.

```
var Var; // Will define a variable with the name as Var.
```

Good to Know: JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997. ECMA-262 is the official name. ECMAScript 2016 (June 2016) is the latest JavaScript version.

2. How do we use JavaScript on the HTML's?

JavaScript can be scripted into HTML's in three ways

Inline

```
<input type="button" id="hello" value="Hello" onClick="window.alert('Hello world!')"/>
```

Internal JavaScript code

```
<script>
    document.getElementById("demo").innerHTML = "John's First  JavaScript";
</script>
```

External JavaScript Code

```
<!DOCTYPE html>
<html>
    <body>
    <script src="johnTharakan.js"></script>
    </body>
</html>
```

Order of Precedence

Inline code has the highest precedence. This can override a function defined in Internal and well as from an external JS file. However, the precedence order of Internal and external JS code depends on where they have been defined. What has been defined at a later point of the code will have the higher precedence and will override the other.

Case Inline wins

```
<html>
    <body>
        <script src="try1.js"></script>
        <script>
```

```
        function a()
        {alert("a");}
    </script>
    input type="button"
    onClick="window.alert('c')"/> //will alert
    c on screen overriding both the definitions
    of functions a();
    </body>
</html>
```

try1.js having content

```
function a()
{alert("b");}
```

Case Internal JavaScript code wins

```
<html>
    <body>
        <script src="try1.js"></script>
        <script>
            function a()
            {alert("a");}
        </script>
        input type="button" onClick="a()"/> /*
        will print a since the try1.js is included be-
        fore the inline definition of function a(); */
        </body>
</html>
```

try1.js having the same content

```
function a()
{alert("b");}
```

Case External JavaScript Code wins

```
<html>
    <body>
        <script>
            function a()
            {alert("a");}
```

```
        </script>
        <script src="try1.js"></script>
    input type="button" onClick="a()"/> /*
    will print b since the try1.js is included after
    the inline definition of function a(); and
    hence external JS function overrides the ex-
    isting inline function definition. */
        </body>
</html>
```

try1.js having the same content

```
function a()
{alert("b");}
```

The scripts or external script tag can be placed anywhere within the body or the head of the HTML page. Cached JavaScript files can speed up page loads if they are in external files.

We can use the script tag to point to a Google CDN URL as below. This way, you don't have to download anything or maintain a local copy.

```
<script src="https://ajax.googleapis.com/
ajax/libs/angularjs/1.5.6/angular.min.js"></
script>
```

Good to Know:

<filename>.js — These files are non-obfuscated, non-minified, and human-readable by opening them in any editor or browser.

<filename>.min.js — These are minified and obfuscated versions, created with the Closure compiler. Use these versions for production in order to minimize the size of the application that is downloaded by your user's browser.

3. What are the various data types in JavaScript?

JavaScript is a *loosely typed* or a *dynamic* language. That means you don't have to declare the type of a variable ahead of time. The type will get determined automatically while the program is being processed. That also means that you can have the same variable as different types:

```
var foo = 42; // foo is a Number
var foo = "bar"; // foo is now a String
var foo = true; // foo is now a Boolean
```

The latest ECMAScript standard defines seven data types:

Six data types that are primitives:

a. **Boolean** (true, false)
b. **Null** (has exactly one value Null)
c. **Undefined** (A variable that has not been assigned a value has the value undefined)
d. **Number** (integers between $-(2^{53}-1)$ and $2^{53}-1$)

Symbolic values: +Infinity, -Infinity, and NaN (not-a-number)

```
>888/0 //will print Infinity
>-888/0 //will print -Infinity
```

Anything that is not a legal Number is represented as NaN

```
>Infinity- Infinity //will print NaN
>Infinity / -Infinity //will print NaN
```

We can also use the function isNaN() to test if a value is a NaN

```
>isNaN(123) //false
>isNaN(-1.23) //false
>isNaN(3 - 2) //false
>isNaN(0) //false
>isNaN('123') //false
>isNaN('Hello') //true
>isNaN('2017/12/12') //true
>isNaN('') //false
>isNaN(true) //false
>isNaN(undefined) //true
>isNaN('NaN') //true
>isNaN(NaN) //true
>isNaN(0 / 0) //true
```

e. **String**

```
var str ="John Tharakan";
var str ='John Tharakan';
var str ="John's Book";
var str ='John said "Lets Go"';
var str ="John said 'Let's Go'";
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

f. **Symbol** (new in ECMAScript 6) whose instances are unique and immutable. Function Symbol() is a wrapper around the primitive symbol.

```
Symbol("symbol1") !== Symbol("symbol1")
```

g. **Object** (which is a collection of any of the 6 primitives and functions)

```
var person = { firstName:"John", lastName:"Tharakan", age:50, eyeColor: "blue" };
```

Objects are a combination of name value pairs called properties and methods (actions) that are performed on these properties. Properties can be

enumerable(enumerates on **for(prop in obj)** loop),

configurable (delete it or change its other attributes) and

writable (replaceable).

```
e.g: Object.defineProperty(person, 'firstName',
{
  value: "John",
  writable: true,
  enumerable: true,
  configurable: true
});
```

Object Properties are accessed as

objectName.propertyName or **objectName["propertyName"]**

eg: **person.lastName;** or **person["lastName"];**

and methods by **objectName.methodName()** or **name = person.fullName();**

4. What is a prototype?

Prototype is an object that defines the characteristics (its properties and methods) of an object. The **Object.prototype** is on the top of the prototype chain. All objects are created from this prototype, by using either the **Object.create()** method or **new** operator. All JavaScript native objects (number, boolean, string, array, date, math etc.) are all inherited from the **Object.prototype**. On the same lines, for e.g. Objects created with new **Date()** will re-inherit the **Date.prototype**, which was in turn inherited from **object.prototype**, but has properties and methods specific for the date.

The standard way to create an object prototype is to use an object constructor function:

```
function Person(first, last, age, eyecolor) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eyecolor;
}
```

With a constructor function, you can use the **new** keyword to create new objects from the same prototype:

```
var myFather = new Person("John", "Tharakan", 50, "blue");
var myMother = new Person("Mary", "Tharakan", 48, "green");
```

Adding a new property to a prototype:

```
Person.nationality = "Indian";
```

Adding a new method to a prototype:

```
function Person(first, last, age, eyecolor) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eyecolor;
this.name = function() {return this.firstName + " "
+ this.lastName;};
   }
```

or else use the prototype property as below

```
Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
/* Adding a new property/method to a prototype will ensure all the objects created from that prototype will have its property/method updated. (as above: both objects myFather and myMother is updated.)

Adding a new property to an existing object will update the property/method only to that instance of the object. (as below, only the object myFather is updated) */
```

Adding a new property to an existing object:

```
myFather.nationality = "Indian";
```

Adding a new method to an existing object

```
myFather.name = function () {
    return this.firstName + " " + this.lastName;
};
```

5. What is the difference between a=10; and var a=10 and how does the scope vary on them?

```
<script>
var a=10; //Global Scope accessible as window.a
b=20; //Global Scope accessible as window.b
function f1()
{
var c=30; /* Local scope. Only within the function f1 */
        d=20; /* Global Scope. Variables created without the keyword var, are always global, even if they are created inside a function. */
}
</script>
```

Global variables live as long as your application (your window / your web page) lives.

Local variables have short lives. They are created when the function is invoked, and are deleted when the function is terminated.

Also

```
<script>
  console.log(window.a); /* This will be undefined since variables defined with var are not hoisted */
  console.log(window.b); /* will print 20 as b being  global gets hoisted to the top of the scope. */
  var a=10;
  b=20;
</script>
```

Do refer to question 8 on variable hoisting.

6. What is the difference between let a=10 and var a=10?

var is scoped to within the function block its defined and let is scoped to any *enclosing* block its defined (It could be a function, or a looping block). Both **var** and **let** behave as global if outside of any block.

```
let a1 = 10; // globally scoped
var a2 = 10; // globally scoped
```

However, global variables defined with **let** will NOTbe added as properties on the global **window** object like those defined with **var** .

```
console.log(window.a1); // undefined
console.log(window.a2); // 10
```

They behave the same on local scope too

```
function f1() {
  let a1=10; // function block scoped
  var a2=10; // function block scoped
} // Outside this function, both a1 and a2 becomes undefined.
```

Block Scope

```
function f1() {
  // a1 is NOT visible out here
```

```
    for( let a1 = 0; a1 < 5; a1++ ) {
        // a1 is visible only within this for loop;
    }
    // a1 is NOT visible out here
}

function f2() {
    // a2 is visible out here
    for( var a2 = 0; a2 < 5; a2++ ) {
     // a2 is visible here and anywhere within the function f2 as a local variable.
    }
  // a2 is visible out here
}
```

Assuming strict mode, **var** will let you re-declare the same variable in the same scope. On the other hand, **let** will not:

```
'use strict';
let a1 = 'John';
let a1 = 'Tharakan'; /* Syntax Error: Identifier 'a1' has already been declared */
'use strict';
var a2 = 'John';
var a2 = 'Tharakan'; // No problem, 'a2' is replaced.
```

Good to Know: Using strict in JavaScript will make the scripting more secure, just like XHTML does to HTML.

Global

```
<script>
"use strict"; /* If declared in global scope, the entire script becomes strict. */
x = 3.14; // This will cause an error (x is not defined).
</script>
```

Local

```
<script>
```

```
x = 3.14; // This will not cause an error.
myFunction();

function myFunction() {
 "use strict"; //Strict is local within the function
  y = 3.14; // This will cause an error
}
</script>
```

What is not allowed in Strict mode?

```
"use strict";
a=10; // Error. You cannot use a variable without declaring it
z = {p1:10, p2:20}; // Error. You cannot use an object without declaring it.
var x = 3.14;
delete x; // Error. You cannot delete a variable/object
function f1(p1, p2) {};
delete f1; // Error. You cannot delete a function.
var x = 010; // Error. Octa Numerals are not allowed.
var x = \010; /* Error. Escape characters are not allowed:
        and a lot more. */
```

7. What are function closures?

A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined within the function), it has access to the outer function's variables, and it has access to the global variables.

```
function outer (firstName) {
   var nameIntro = "Mr.";
      function inner (theLastName) {
```

```
        /* this inner function has access to the
outer function's variables, including the param-
eters being passed. */
        return nameIntro + firstName + " " + the-
LastName;
        }
return inner;
}
var myName = outer ("John"); /* At this point,
the outer function has returned.
The closure (inner function) is called here after
the outer function has returned above yet, the
closure still has access to the outer function's
variables and parameter */
myName ("Tharakan"); // Mr. John Tharakan
```

Closures are used extensively in Jquey, NodeJS and other associated JavaScript Libraries.

8. What is variable hoisting? What is function hoisting?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope. In JavaScript, a variable can be declared after it has been used. In other words; a variable can be used before it has been declared.

JavaScript Declarations are hoisted.

```
x = 5; // Assign 5 to x
elem = document.getElementById("demo"); //
Find an element
elem.innerHTML = x; // Display x in the ele-
ment
var x; // Declare x
```

is the same as

```
var x; // Declare x
x = 5; // Assign 5 to x
elem = document.getElementById("demo"); //
Find an element
elem.innerHTML = x; // Display x in the element
```

But JavaScript Initializations are Not Hoisted

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); //
Find an element

elem.innerHTML = x + " " + y; /* Displays x as
5 and y as undefined because y is not initialized
yet */

var y = 7; // Initialize y
```

which is the same as

```
var x = 5; // Initialize x

var y; // Declare y

elem = document.getElementById("demo"); //
Find an element

elem.innerHTML = x + " " + y; /* Displays x as
5 and y as undefined because y is not initialized
yet */

y = 7; // Assign 7 to y
```

Function Hoisting is based on the same principle

```
hoistedFunc(); // "Functions hoist too!"

//Lines of code

function hoistedFunc() {

   console.log("Functions hoist too!");

}
```

The function is being used even before it's declared. But this will throw an error if the function declaration is part of an overall expression as below

```
nonHoistedFunc(); // TypeError: Undefined

//Lines of code

var nonHoistedFunc = function () {

   console.log("This will not work!");

};
```

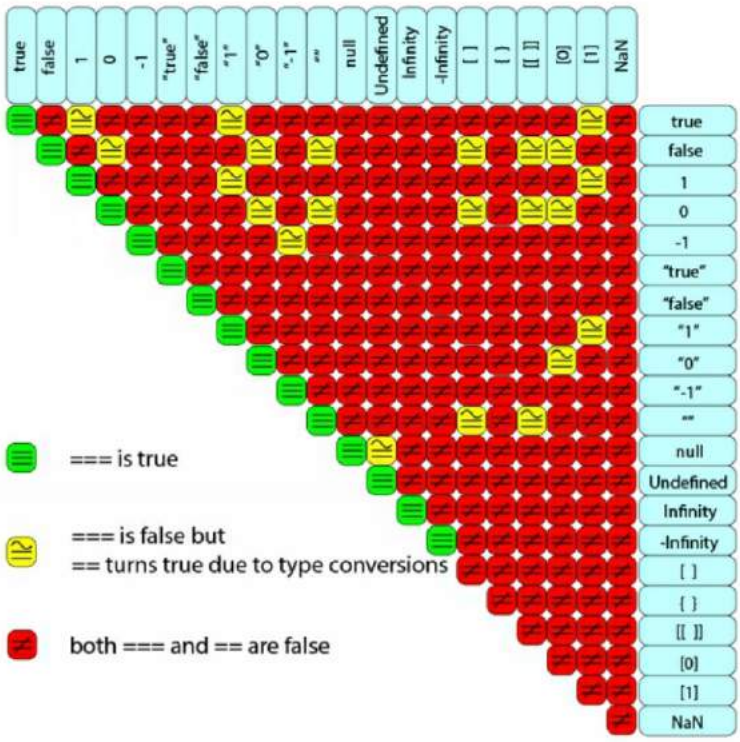
9. What is the difference between == and === operator?

The `==` operator will compare for equality *after doing any necessary type conversions*. The `===` operator will **not** do the conversion, so if two values are not the same type `===` will simply return `false` .

```
true == 1; //true, because 'true' is converted to 1 and then compared

"2" == 2; //true, because "2" is converted to 2 and then compared

true === 1; //false

"2" === 2; //false
```

A detailed mapping of the various combinations can be represented diagrammatically using a truth table below.

10.



What is the output on the console?

```
function f1 () {
    var a=b=c=10;
  }
f1();
console.log(a);
console.log(b);
console.log(c);
```

`console.log(a)` will print "a is not defined"

Reasoning: `var a` isdefined inside the function with-`var` keyword. It is a local variable and not able to access outside the `function f1()` .

console.log(b) will print 10

Reasoning: b is declared without **var** keyword so it is a global variable. (Remember question 5)

console.log(c) will print 10

Reasoning: c is declared without **var** keyword so it is a global variable.

11. What is the output on the console?

```
var foo = 10;
    function helloWorld() {
        foo = 20;
        function foo() {}
    }
helloWorld();
console.log(foo);
```

foo will print 10.

Reasoning: var foo =10; is a global variable, Now the helloWorld function will hoist the function foo() with it to the top of the function. So the function helloWorld () is as good as below

```
function helloWorld() {
var foo = function() {}
foo =20;
}
```

Now, after hoisting foo becomes the local variable of the helloWorld function. and it gets overridden with the value 20, thus making the other global variable foo retain the value of 10.

12. What is the output on the console?

```
console.log(3<2<1);
```

It will print true;

Reasoning: First the engine evaluates 3<2 which evaluates to false. This Boolean is evaluated with <1 . False <1 type converts to 0<1 evaluating to true.

i.e. (3<2<1) evaluates to

((3<2)<1) evaluates to

((false)<1) evaluates to

(0<1) evaluates to true.

13. What is 'this' in JavaScript?

This refers to the current object in scope. On a global context, (outside of all functions) this refers to the window object.

Within any function, it refers to the object the method is called on. If it's a global function call, it refers to the window object provided its NOT in strict mode, but if its in a strict mode, the value of this would remain at whatever it was set to the execution context, meaning this could remain as 'undefined'

Eg:

```
function f2(){
 "use strict"; // strict mode
 console.log(this); // will print undefined
}
```

14. What is call and apply in JavaScript and how do they differ? What is context switching in JavaScript?

Where a function uses this keyword in its body, its value can be bound to a particular object in the call using the call or apply methods which all functions inherit from function.prototype.

```
function add(c, d){
  return this.a + this.b + c + d;
}
var o = {a:1, b:3};
/* The first parameter is the object to use as
 'this', subsequent parameters are passed as
 arguments in the function call */
add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16
/* The first parameter is the object to use as
'this', the second is an array whose
 members are used as the arguments in the
function call */
add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34
```

So here's the difference between call and apply. Both can be called on functions, which they run in the context of

the first argument. In call the subsequent arguments are passed in to the function as they are, while apply expects the second argument to be an array that it unpacks as arguments for the called function. In other words use apply() when you are not sure on the number of arguments the function needs.

We can use call and apply to execute various methods in the context of various objects. The concept of switching the context of execution dynamically is commonly referred to as context switching in JavaScript.

15. **What is the output on the console?**

```
function f1() {
        var a = 10;
        console.log(this.a);
      }
function f2(a) {
         this.a =10;
         console.log(this.a);
       }
  f1();
  f2();
```

Function f1() will print **undefined** on the console, because **a** is local variable and 'this' refers to the window object.

Function f2() gives you the output **10**. "this" will refer the function context and print the value of a.

16. **What is the output on the console?**

```
var array1 = [1,2,3]
  newArray2 = array1;
    array1.push(4);
      console.log(newArray2);
```

newArray2 will be printed as [1,2,3,4]

Reasoning: On executing the statement newArray2 = array1;newArray2 is created as a new reference to array1. Any modification to the array1 will be reflected on the newArray2 and vice versa, since they refer to the same memory location.

the first argument. In call the subsequent arguments are passed in to the function as they are, while apply expects the second argument to be an array that it unpacks as arguments for the called function. In other words use apply() when you are not sure on the number of arguments the function needs.

We can use call and apply to execute various methods in the context of various objects. The concept of switching the context of execution dynamically is commonly referred to as context switching in JavaScript.

15. **What is the output on the console?**

```
function f1() {
        var a = 10;
        console.log(this.a);
      }
function f2(a) {
         this.a =10;
         console.log(this.a);
       }
  f1();
  f2();
```

Function f1() will print **undefined** on the console, because **a** is local variable and 'this' refers to the window object.

Function f2() gives you the output **10**. "this" will refer the function context and print the value of a.

16. **What is the output on the console?**

```
var array1 = [1,2,3]
  newArray2 = array1;
    array1.push(4);
      console.log(newArray2);
```

newArray2 will be printed as [1,2,3,4]

Reasoning: On executing the statement newArray2 = array1;newArray2 is created as a new reference to array1. Any modification to the array1 will be reflected on the newArray2 and vice versa, since they refer to the same memory location.

17. What is the output on the console?

```
var var1 ="John"; // initialized with var
var2 ="Tharakan"; // initialized without var
delete var1;
delete var2;
console.log(var1);
console.log(var2);
```

console.log(var1); // will print John

console.log(var2); // will print var2 is not defined.

Reasoning: var1 is created as a variable, but var2 is created as a property of the window object. Delete can be used to delete the properties of an object but not a variable. Hence the statement delete var1; will print false on the console and delete var2 will print true on the console indicating that it has deleted the property, thus making var2 not defined henceforth.

18. What is chaining? How would you use it to reverse a string in JavaScript?

Chaining is a technique that can be used to simplify code in scenarios that involve calling multiple functions on the same object consecutively.

```
function reverseString(str) {
  return str.split("").reverse().join("");
}
reverseString("John Tharakan"); // Would return "nakarahT nhoJ"
```

Split() splits the string into a string array of individual elements/alphabets, reverse() reverses the order of the elements, and joins() merges the elements into a single entity.

19. What is the value of foo in each case

```
var foo = 10 + '20' /* Addition (+) of a string with other entities yields a string. This yields '1020' a string. */
```

```
var foo = '10' + 20 /* Addition (+) of a string with
other entities  yields a string. This yields '1020' a
string. */

var foo = 10 - '20' // -10

var foo = 10 * '20' // 200

var foo = 10 / '20' // 0.5
```

20. If we can get an output of 8 as return for the function add (3, 5), how can we get the same output by calling the function add () as add (3) (5)?

```
var add = function(x){
    return function(y){
        return x + y;
    }
}
```