# JavaScript

## Best Practices

Use === to not do any type conversion.

Always ‘use strict mode’;

Use if(typeof x !== ‘undefined’) to check for undeclared/undefined variables.

Put vars and functions at top of scope.

Object.defineProperty(): Precise control over a property within an object.

You can only delete a property of an object, not a variable nor an object.

If you start a number with 0, JS engine treats it as an octal. However if you want to properly use octal then use **parseInt**(num, 8);

Callbacks:

Use named functions as opposed to anonymous funcs

Be consistent with the functions (ie always check for errors first and pass error first in params)

Return your callbacks

## Primitives and Object Types

**Primitives**: Numbers, Strings\*, Booleans, Undefined, Null and Symbol.

All are immutable except String.

All are objects except null and undefined.

Primitives are compared by value.

A copy of the type is passed around when used in functions.

**Objects**: All other types, such as Arrays, Functions, Objects, Dates are objects.

These are mutable.

Objects are compared by reference.

A copy of the reference (the memory address) is passed around when used in functions. Thus a copy of an object changed will affect the original object.

**let** is only scoped within a block. **var** is within a function

=== **does not** do any type conversion, whereas == does convert types.

**===** is equal to. **!==** is not equal to.

**not** is also known as the **bang operator(!)**

A value is **truthy** if it’s not **falsey**.

Falsey values:

0  **Note** negative numbers are truthy!!

false

“” or ‘’

null

undefined

Nan

***let defaultName = username || 'Stranger'****;* defaultName will be username if its truthy otherwise Stranger(when username is empty or undefined).

const can get its value from a function. const x = getNumber();

## Arrays

**array .unshift(element)** adds item to front of list

o = **array.shift()** removed first item from array

**array.slice(1,3)** returns items 2,3,4 from array

Pass by reference: If the array is mutated inside the function, that change will be maintained outside the function as well.

**splice()** changes entries in array

**join**(separator): joins the array entrys into 1 string.

**indexOf(obj)**: returns index of first occurrence. -1 otherwise

**foreach(func):** Runs a function on each entry

**map(func) :** Manipulates each row and returns array

**filter(func) :** filters rows and returns array

**findIndex(func)**: using a func returns index of first occurrence. -1 otherwise

**fil(func, initValue)**: Reduces array to single value

**some(func)** true if any item passes the func

**every(func)** true if all item passes the func

## Functions

Functions are objects. Every function is linked to **Function.prototype** (Which is linked to **Object.prototype**).

**Invoking functions:** When a function is invoked 2 extra params are also passed, ***this*** and **arguments**. Arguments are all the params. ***this*** is the function context and depends on which type of invocation.

**Method Invocation:** Function is contained within an object. ***this*** is bound to that object. ie Myobject.increment()

**Function Invocation:** Happens when function is not part of object, then its part of global scope. ***This*** corresponds to that global scope.

**Constructor Invocation:** Function is invoked with the new keyword. ***this*** then refers to the variable that has been assigned the new function. The ***this*** below refers to the variable x. (This is also known as the Function constructor)

### var AStatus = function (string) {

### this.status = string;

### };

### var x = new AStatus(“in progress”)

**Apply/Call Invocation**: Functions have the following methods (from its prototype**) Apply** and **Call. Apply** and **Call** allow a ***this*** object to be passed in to replace the functions context.

### Let car = {

### carId: 123,

### getId = function(prefix, seperator){

### console.log(`${prefix } ${ seperator } ${this.carId}`)

### }

### }

### Let newCar = {carId: 666}

### Console.log(car.getId.call(newCar,’cid’, ‘::’)) //displays cid:: 666

### Console.log(car.getId.apply(newCar,[’ID’, ‘++’])) //displays ID++ 666

Difference between **Call** and **Apply** is **Apply** allows you to pass in array of arguments, **Call** requires args to be passed in explicitly.

**Note**: if null is passed as this in **Call/Apply** then this is deemed to be the global context.

**Bind**: Makes a copy of a function and also changes the ***this*** value.

### Let newFn = car.getId.bind(newCar);

### console.log(newFn());

**Parameters:** If the incorrect # of params are passed in the function will still run.If functions params are not passed in then they are defined as undefined.

**Default parameters**: These exist since ES6.

**Arguments** are a hidden property that each function has that is an array of params passed. Thus if a function is expecting 0 params, you can still pass in some and them access them using the arguments array.

Default parameters occur if you give a value to a func when defining the func. function power(base, exponent = 2)

**Closures**: Allows inner function to access outer functions variables, even after outer function returns.

**Cascade:** If functions return undefined then return this. This will enable you to chain multiple functions of an object together. (like LINQ)

**IIFE (immediately invoked function expression)**. Firing the function straight away.

**Arrow Functions**: Just like C#. Note. They don’t have their own ***this*** value. ***this*** refers to the enclosing context.

## Scope

Scope in JavaScript refers to the accessibility or visibility of variables.

**Function scope**: The parameters and variables defined in a function are not visible outside of the function. However a function contained within a function or object has access to its outer vars. This is **Closure**.

**Block scope**: ES6: Scope within curly braces {}. Only works with **let** and **const**.

**Global scope**: When neither of the above

**Catch scope**: A scope within a catch block of Try/Catch

**StrictMode**: If a var is used but not defined an error will only occur if **‘Use Strict’** is on, otherwise it will define that var.

**ScopeChain:** If var is not in current scope, then engine looks at parent scope, up to the global scope.

Lexical Scoping: Lexical means where something is positioned in code. Basically it means an inner function gets access to the scope of its outer functions/execution contexts

**Execution context**

Execution context is a stack that contains 1 **global execution** **context** at the bottom and any number of **function execution** **contexts** (can also have an execution context based on the **eval** function). The function contexts get popped off when a function ends.

The execution context is single threaded and synchronous.

When a new execution context is added to the Execution stack 2 stages occur:

1. **Creation stage**(when function is called, but before it executes)
   1. Scope chain gets initialized
   2. Variables and functions within context get created: This is the variableObject below.
   3. ***this*** gets determined
2. **Code execution stage**
   1. Values/ references get assigned to variables/functions
   2. Code is executed

Execution context can be represented conceptually as an object with 3 properties

executionContextObj = {

'scopeChain': { /\* variableObject + all parent execution context's variableObject \*/ },

'variableObject': { /\* function arguments / parameters, inner variable and function declarations \*/ },

'this': {}

}

Looking at the **creation** and **execution** stages.

1. Initialize the scope chain (this is detailed below)
2. Create the variable object
   1. Creates arguments object containing the parameters.
   2. Scans for functions. For each function creates a property in **variableObject** which contains function name and a pointer to its location in memory. If function name already exists it’s overwritten.
   3. Scans for variables. Creates a prop in in **variable object** and sets its value to undefined. If variable already exists it ignores the latest variable.
3. Determines this keyword. \*See below
4. Runs the code and assigns the variable values as the code is executed line by line.

Looking at an example

### function foo(i) {

### var a = 'hello';

### var b = function privateB() { };

### function c() { }

### }

### foo(22);

When foo(22) is called the creation stage looks like this:

fooExecutionContext = {

scopeChain: { ... },

variableObject: {

arguments: {

0: 22,

length: 1

},

i: 22,

c: pointer to function c()

a: undefined,

b: undefined

},

this: { ... }

}

After the execution stage has run the fooExecutionContext looks like this (**Note** I’ve only shown the changed values)

fooExecutionContext = {

…

a: ‘hello’

b: pointer to function privateB

…

}

**Scope chain:**

Every function has an execution context which contains a variable object (variableObject see above) which contains the params, functions and variables of that function. The scope chain of each execution context is simply the current contexts variableObject and all its parents variableObjects.

These exist even if a function within the chain has ran and is popped off the execution stack. Looking at the below function we can see 4 execution contexts.

### function one() { => one VO

### var a = 1;

### two();

### function two() { => two VO

### var b = 2;

### three();

### function three() { => three VO

### var c = 3;

### alert(a + b + c); // 6

### }

### }

### }

### one()​; => Global VO

Thus the scope chain for function three is ‘*three VO + two VO + one VO + Global VO’*. Ie Function 3 can access the VO of Function 2, function 1 and the global context.. The scope chain for function one is ‘*one VO + Global VO’*. Ie it can access the global context.

This is how closures work.

When a new execution context is added to the Execution stack (when you enter a new function A) an entry is also added to the corresponding scope chain [A VO + Global VO.] This entry contains the variables used in this context. When the function A is finished the execution context dies and gets popped off the execution stack. Its scope chain also dies. However if that function A also contained a function B then it also got a scope chain [B VO + A VO + Global VO.] B still exists therefore its scope chain exists and thus it can still access A’s Variable object even though A has been popped off the execution stack!

**Determining this keyword. Remember** this happens every time a function is called.

1. If ***this*** is defined in the global context then it equals the global context!

### // global scope

### foo = 'abc';

### alert(foo); // abc

### this.foo = 'def';

### alert(foo); // def

1. If ***this*** is defined in a function then 2 things determine ***this***
   1. What is the parent scope in which the function is called.
   2. How the function call syntax is written. If no LHS then its the scope that the call is made in. If LHS then that object is ***this.***

### function bar() {

### alert(this);

### }

### bar(); // global - because bar() belongs to the global object when invoked and no LHS exists

### //bar() = RHS. LHS is empty

### var foo = {

### baz: function() {

### alert(this);

### }

### }

### foo.baz(); // foo - because the method baz() belongs to foo when invoked as it’s the LHS

### //foo = LHS baz() = RHS

1. Finally if ***this*** is called within an event handler then it always equals the element that triggered it.

**Closures:**

**When to use:**

**Encapsulation**: Allows us to hide implementation details from outside contexts. This is known as the module pattern

**Callbacks:** Defer invocation of function in a non blocking way.

**Closures as arguments: …**

**When not to use:**

**Multiple nested functions:**

## Objects

**Prototypes**. Allows an object to have additional properties that will be available to all objects of that type. Each object has a prototype property that allows for inheritance.

Its mainly only methods that are added to the prototype, not props.

**Prototype Chain**: person->person.prototype->object.prototype. The engine goes thru the chain looking for properties/methods that it needs to call.

### Person.prototype.DisplayName = function(){return ‘marcus’);

**HasOwnProperty**: Func that returns true if prop is not part of prototype.

**InstanceOf**: Operator that return whether object is part of a prototype chain. myCar instanceOf Vehicle; // true;

**Object.create**: Way to create a new object (as opposed to function constructors), using an existing object as the prototype of the newly created object. You pass in the prototype followed by the object itself. It builds an object that inherits directly from the prototype.

**Higher-order functions** are functions that accept other functions as arguments and/or return functions as output

## Classes

Modules: Allows code to be split into different files. Need to use **Export** and **Import** keywords.

## Hoisting

All variable declarations are hoisted to the top of the containing scope. (var x;)

All function declarations are hoisted to the top of the containing scope above variable declarations. (function xx(){})

However function expressions (var x = function(){..}) are treated as a variable declaration

Function declarations override variable declarations unless there’s an initialization of the variable. So

### function value(){   return 1; }

### var value;

### alert(typeof value);    //"function"

### function value (){ return 1;}

### var value =2 ;

### alert (typeof value);    //"number"

If more than 1 function declaration has the same name (in the same scope) the **last** function always wins.

If more than 1 variable declaration has the same name (in the same scope) the **first** variable always wins.