**JavaScript**

* **Compiler:** The source code is converted into machine code once and then gets executed.

**Compiler Pros:**

* + Suppose we have a loop that runs 40 times, we do not need to translate the same code again and again. The time it saves is huge.
  + The converted code will be more efficient cause we have more time for optimization.

**Compiler Cons:**

* + It takes a little bit more time to start up because it has to go through that compilation step at the beginning.
* **Interpreter:** Interpreters are quick. We don’t have to go through that whole compilation step before execution. It just starts translating the first line and then executes it.

**Interpreter Pros**

* + Fast start-up times are characteristic of interpreters. That’s why browsers used JavaScript interpreters in the beginning.

**Interpreter Cons**

* + When you’re running the same code more than once. For example, if you’re in a loop. Then you have to do the same translation over and over and over again.
  + The code will be less efficient than the compiler because we have less time for optimization.
* **JIT Compiler:** In short Just in time compiler is nothing but a combination of an interpreter and a compiler. To get rid of the interpreter’s inefficiency, “the interpreter keeps retranslating the same code every time it goes through the loop”. In the JIT compiler, we have a new component called a monitor (aka a profiler). That monitor watches the code as it runs and
  + Identify the hot or warm components of the code eg: repetitive code.
  + Transform those components into machine code during run time.
  + Optimize the generated machine code.
  + Hot swap the previous implementation of the code.
* **Babel:** A trans piler that can convert ES6 code to ES5 code.
* **Typescript:** Typescript is a superset of JavaScript that compiles down to JavaScript.
* **Ecma International:** An organization that creates standards for technologies.
* **ECMA-262:** This is a standard published by Ecma International. It contains the specification for a general-purpose scripting language.
* **Scripting Language:** While programming languages are compiled, scripting languages are mostly interpreted. Even though there are some scripting languages that are both compiled and interpreted, such as Python and Groovy.
* **ECMAScript:** The specification defined in ECMA-262 for creating a general-purpose scripting language.
* **ECMAScript 6:** It is the sixth edition of the ECMA-262 standard, and features major changes and improvements to the ECMAScript specification.
* **JavaScript engine:** A program or interpreter that understands and executes JavaScript code. JavaScript has many JS engines like V8, SpiderMonkey, Chakra etc. JS Engine is also **Synchronous**.
* **JavaScript:** A high-level, general purpose **scripting language** that conforms to the ECMAScript specification and is built into browsers that allows you to implement functionality on web pages/apps. JavaScript is also available in other programming environments, such as Node.
* **Memory Heap:** It’s where the JS Engine allocates memory for variables and Objects.
* **Call Stack:** Stores the variables and functions as code executes and keeps track of where the code is in execution.
* **Stack Overflow:** Too many functions called and stored on call-stack that exceed call stack size.
* **Memory Leak**: When we use all our available memory heap and fill it with data (variables, objects, etc.).

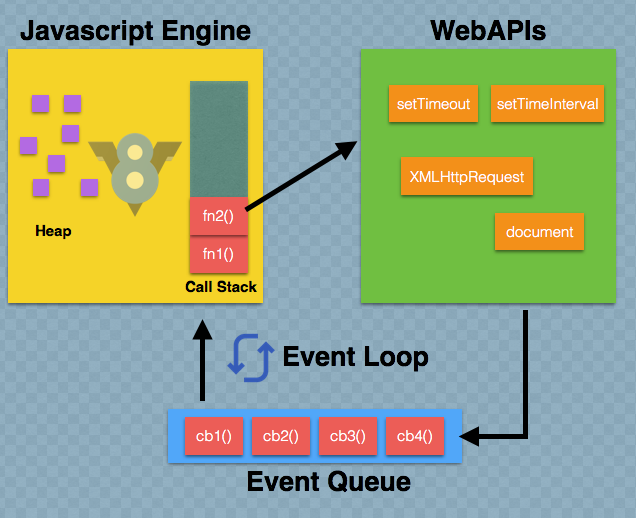
Memory leaks can happen 3 different ways:

1. Global variables: Using too many global variables can lead to memory leak.
2. Event Listeners: Using too many event-listeners
3. Set Intervals: Objects inside the set-intervals

* **Single Threaded:** JavaScript is a **single threaded** language that can execute only 1 set of instructions at a time, can’t perform multiple things. Single threaded language has 1 call stack. JavaScript is also **Synchronous**.
* **Web-Browser and Node.js = JS Runtime + Web API or Backend Node API**
* **JavaScript Runtime: JS Runtime** is the environment in which the JavaScript code runs and is interpreted by a JavaScript engine. JavaScript is **Synchronous** and **single threaded**. JS Engine is also **Synchronous**. It’s the **JS Runtime Environment (Web-Browser and Node.js)** that provides **Web API or Backend Node API (Worker Threads) that allows Asynchronous programming.** The runtime also provides the host objects (HTML, CSS etc.) that JavaScript can operate on and work with.
* For the client side, the JavaScript runtime would be the web browser, where host objects like windows and HTML documents are made available for manipulation. Browser also provides Web Apis that JS uses to become Asynchronous.
* For the server side, the JavaScript runtime is Node.js. Server-related host objects such as the file system, processes, and requests are provided in Node.js. There are other Apis that are provided to allow JS to be Asynchronous.
* **WEB API, Callback Queue and Event Loop**: These are APIs supplied by host environment like **Web-Browser**. So Anytime a **WEB-API** like **setTimeout(), fetch()** comes up which is not part of JavaScript, **CALL-STACK** will assign that to **WEB-API** to take care of in the **background** so that it can move on to next function call on stack.

Once the **WEB-API** is done with work **like fetch() api done with fetching data**, it waits in **CALLABCK QUEUE** where **EVENT LOOP** will keep checking until **CALL STACK** is empty and entire JS file is read. Once call-stack is **empty** the **EVENT LOOP** would push data/function to **Call-Stack**.

WEB-API are constructs built into the browser that sits on top of the JavaScript language and allows you to implement functionality more easily. Example: DOM (Document Object Model) API (To manipulate HTML, CSS), Fetch API and XMLHttpRequest API (To fetch data from the server), Web Storage API and IndexedDB API (Client-side storage APIs) etc.



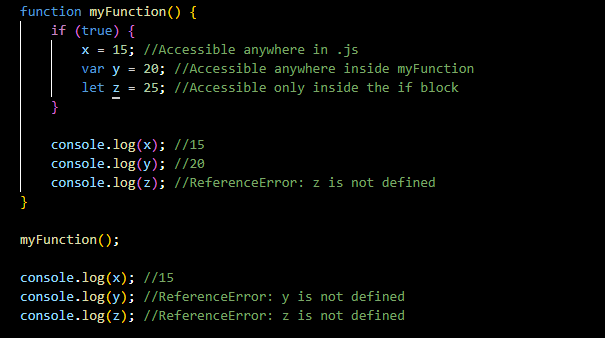
**Node.js:** Just like **WEB-API** in Brower Env Node.js has backend apis inside global which are called **WORKER THREADS** and any **ASYNCHRONOUS tasks** are assigned to worker threads which allows ASYNCHRONOUS programming with SYNCHRONOUS JavaScript and SYNCHRONOUS V8 engine in backend.

**Variable Scoping**

* If a variable is defined without **var**, **let** or **const** anywhere in .js file. It becomes global variable. We should use **"use strict"** to avoid this.
* if a variable is defined with var. It has scope within the function it is defined in.
* if a variable is defined with let or const. It has scope within the block it is defined in.

**var = Immediate function scope**

**let = Immediate Block scope**

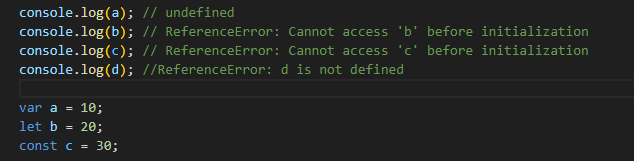


**Hoisting**

* Hoisting Happens only for those that start with words "function" or "var"
* We can avoid hoisting problems by using LET and CONST

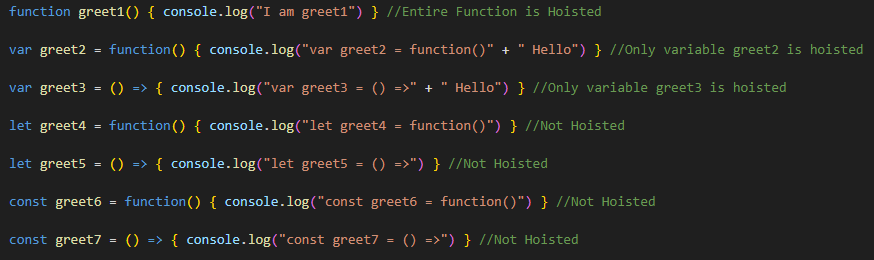
**Variables Hoisting**

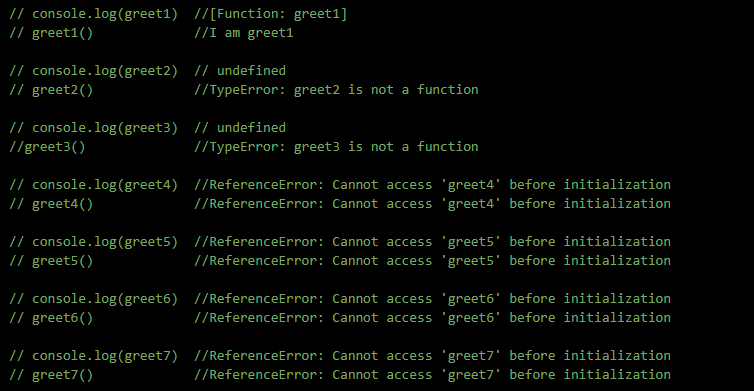
All variables whether they are assigned with VAR or LET or CONST, all are hoisted. var with **undefined** value



**Functions Hoisting**

* Only the **function declarations** are hoisted. Any **Function expression** (LET, CONST) will **not** be hoisted
* For the **Function Expression starting with VAR** only the variable will be hoisted with **undefined** value
* For Example, in case of greet2 and greet3, value will be undefined





**Execution Context**

* Every time **code is run in JS** it’s always run in **execution context**
* **Every function in JS file** has its own **EXECUTION CONTEXT**
* Before even that Every JS File starts with **GLOBAL execution Context**
* Global Execution Context starts with a global variable WINDOWS (GLOBAL with node) and THIS
* Every Execution context has 2 parts. Creation Phase and Execution Phase
* Creation Phase is where hoisting happens
* Execution Phase then runs code.
* We can avoid hoisting problems by using LET and CONST

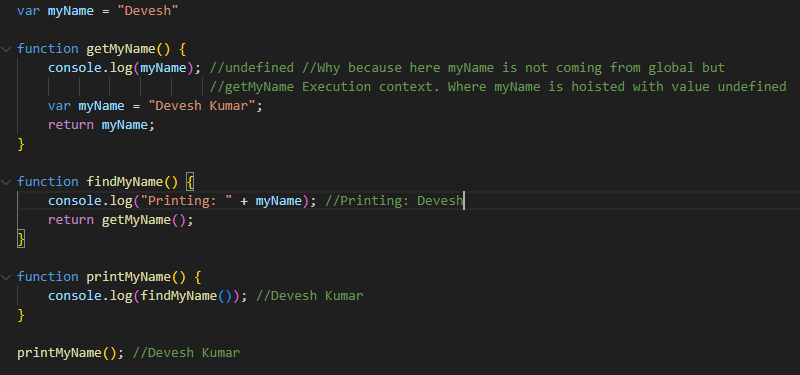
In the example below 5 execution contexts are created. Every Execution Context has its own hoisting. For below code following Execution Contexts are created on Call Stack.

**getMyName** execution context

**findMyName** execution context

**printMyName** execution context

Global execution context



**Lexical Environment or Lexical Scope or Lexical Context**

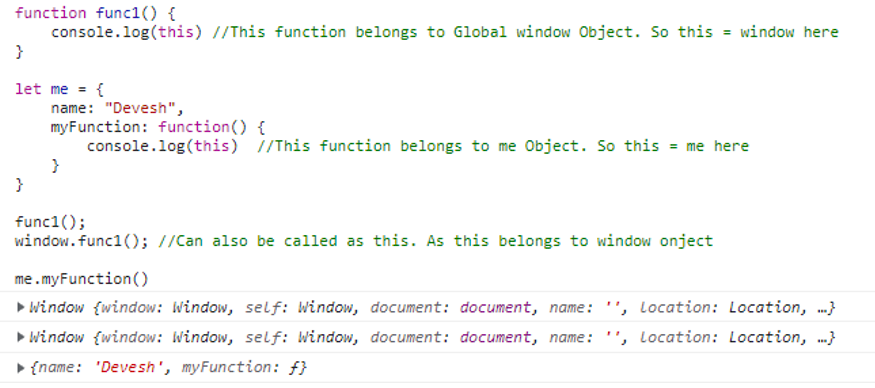
**Lexical Scoping:** Lexical Scope defines that in which **EXECUTION CONTEXT** the code/function is written. Based of function’s location, it is determined which variables/data is accessible. In code below:

|  |  |
| --- | --- |
|  | * printMyName() function's lexical scope is Global Execution Context as it's located inside Global & has access to Global Variables * findMyName() function's lexical scope is Global Execution Context as it's located inside Global & has access to Global Variables * getMyName() function's lexical scope is Global Execution Context as it's located inside Global & has access to Global Variables * getName() function's lexical scope is getMyName() Execution Context as it's location is inside getMyName() & has access to Global/variables inside getMyName() |

**Dynamic Scoping:** Dynamic Scoping uses the location of the function's invocation to determine which variables/data are available. “**this”** keyword in JS is dynamically scoped, which means it can have different values depending on where it is called. Explained in This Keyword Section below.

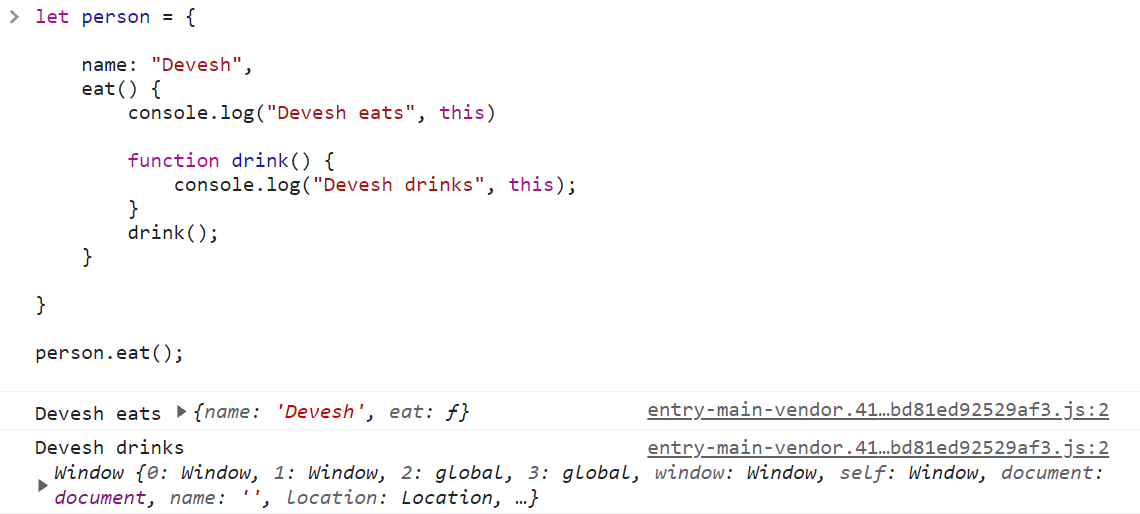
**This Keyword**

* **“this”** in JS is **dynamically scoped**, which means it can have **different values** depending on **where it is called.**
* **"this"** keyword in JavaScript function indicates **the object** that function belongs to

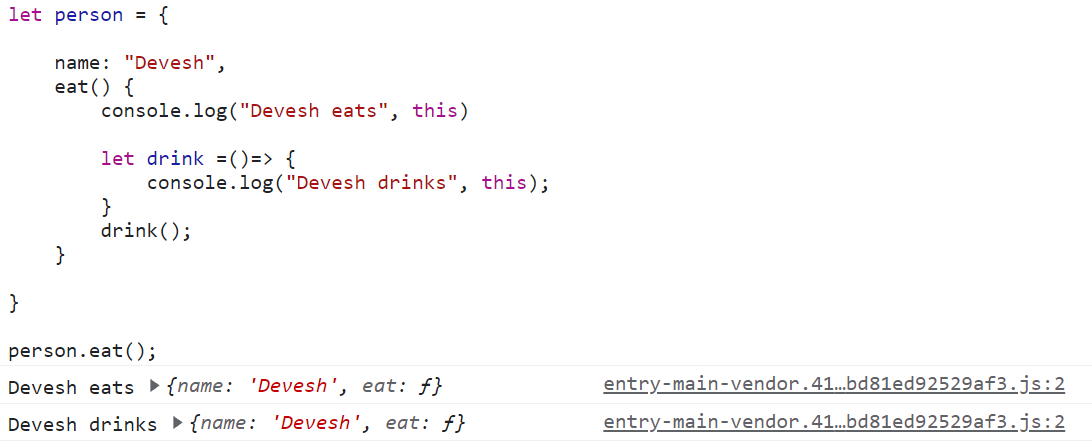


**Dynamic Scope of “this” in JS**

In the example below, value of “this” keyword inside drink function is window even though it’s inside eat() function of person object. This is because while eat is being called by person object person.eat() but drink is being called by no specifies object. So, value of “this” becomes window.



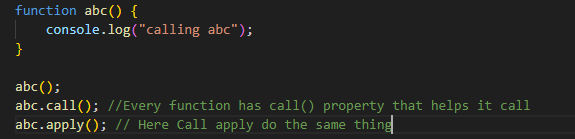
We can solve this issue by using arrow functions then the value of this would be person only.



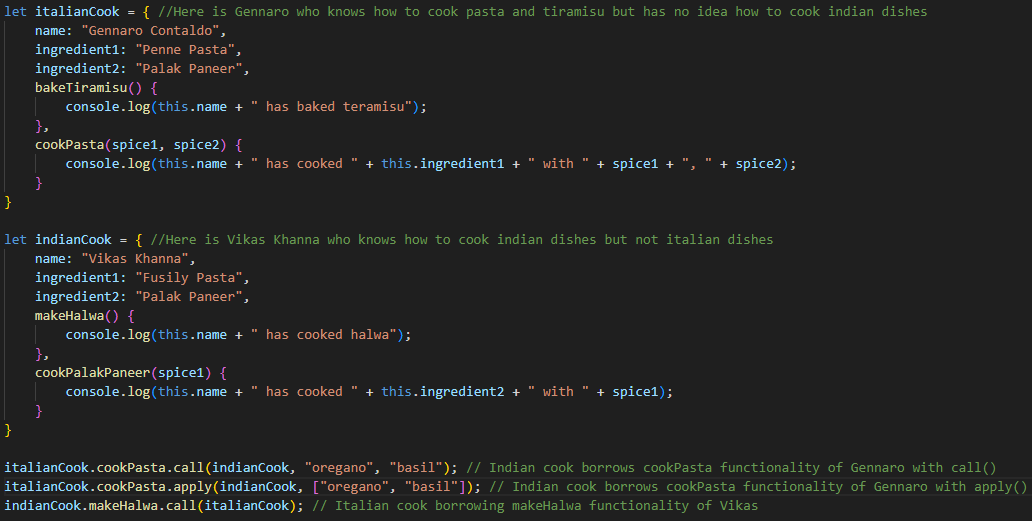
**call(), apply(), bind() (Manipulation of “this” keyword)**

**call(), apply()**

All functions internally use call() to be invoked. apply() only will do the same as call()



* call() and apply() functions allow/enable borrowing of functionalities of other objects by another object
* call() and apply() both can take other object as first argument who will use the functionality of first object
* Other than object, call() can take other arguments as required separated by comma
* Other than object, apply() can take array of other arguments required





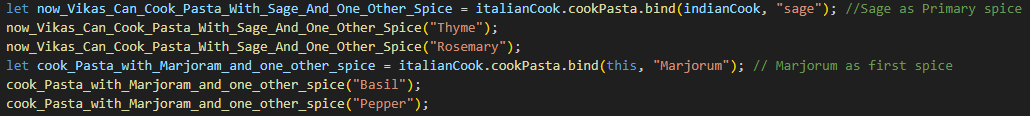
**bind()**

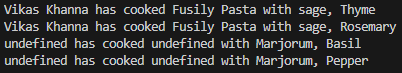
unlike call(), bind() does not call the function right away but returns the borrowed function which can be called later with different this.



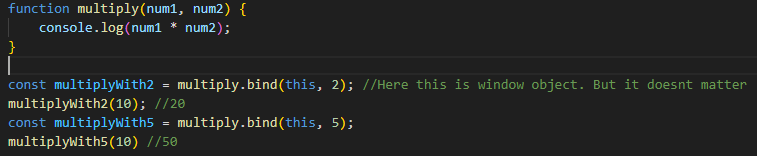
**bind() and Function Currying**

* Creating Multiple versions of a function: Function Currying
* Function Currying is creating a new function with bind() with partial arguments and then passing rest of arguments later





**Another Example of Function Currying**



**bind() can be used to fix "this" keyword dynamic scoping problem**



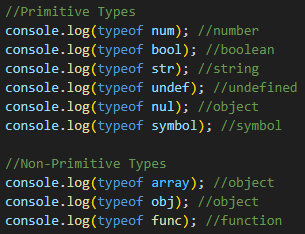
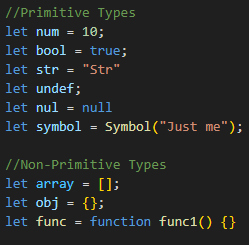
**Data Types**

1: Variable datatype is determined at runtime by value assigned.

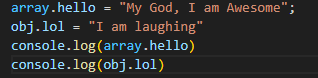
2: 3 Primitive Types - string, number, boolean.

3: Other Datatypes - **undefined**, object, symbol

4: **null** is **object** datatype



**Arrays** & **Functions** are both Object types even though function shows function type. As we can add **properties** to both Arrays and Functions just like Objects.

**Type Coercion and Type Conversion**

JavaScript variables can be converted to a new variable of another data type in 2 ways.

**Type Conversion:** When we do it on purpose by the use of a JavaScript function

**Type Coercion:** Automatically done by JavaScript itself

**Type Conversion**

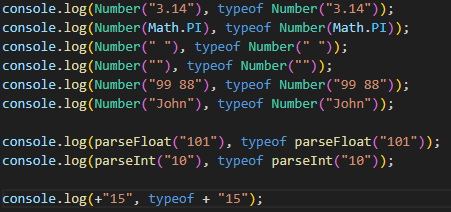
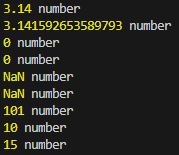
**Converting Strings to Numbers**

**Number():** Returns a number, converted from its argument

**parseFloat():** Parses a string and returns a floating point number

**parseInt():** Parses a string and returns an integer

**Using +:** The unary + operator can be used to convert a variable to a number

**Converting Booleans/Dates to Numbers**

The global method **Number()** can also convert **Booleans** and **dates** to numbers.

**Dates to Number**

d = new Date();

Number(d) // returns 1404568027739

d.getTime() // returns 1404568027739 // The date method **getTime()** does the same.

**Booleans to Number**

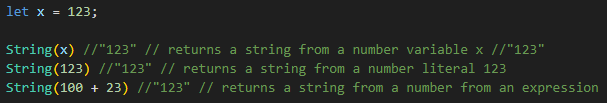
Number(false) // returns 0

Number(true) // returns 1

**Converting Numbers/Booleans/Dates to String**

We can use String() and toString() methods to do same things.

**NUMBER to STRINGS**



**DATES to STRINGS**



**BOOLEAN to STRINGS**



**Automatic Type Conversion (Type Coercion)**

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

5 + null    // returns 5         because null is converted to 0  
"5" + null  // returns "5null"   because null is converted to "null"  
"5" + 2     // returns "52"      because 2 is converted to "2"  
"5" - 2     // returns 3         because "5" is converted to 5  
"5" \* "2"   // returns 10        because "5" and "2" are converted to 5 and 2

**Truthy and Falsy**

|  |  |
| --- | --- |
| **Truthy** | **Falsy** |
| Numbers(Except: 0) , Strings(Except: Empty String “”), Any Array, Any Object, Any Function, +/- Infinity | 0, Empty String (“”), null, undefined, NaN |

**Double == Vs Triple ===**

We should **never use ==** for comparisons to avoid JS Auto-Conversions/Comparisons. We can however **use ==** in one scenario if we need to check for both **null** **and** **undefined**.

**null == undefined // true**

**null === undefined // false**

So if we want to to check **for both null and undefined** and we don’t want to write like if **(x=== null or x===undefined)**. We can simply write **if (x==null)** as it will check for both null and undefined.

**x == null // True if x is null or undefined**

**x === null // Only true is x is null**

**Functions in JavaScript**

**A new Way to create function using Function interface:**



**Functions in JS are Objects. As they can be assigned Properties to them.**



**Functions also have name property.**

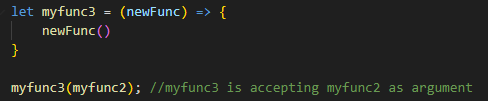


**FUNCTIONS in JS are FIRST CLASS CITIZENS. why?**

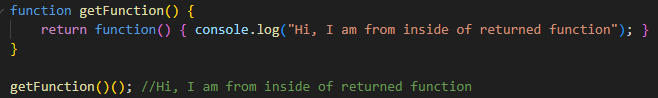
1. **As functions can be ASSIGNED to variables**



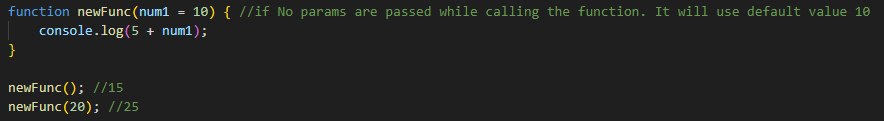
1. **Functions can ACCEPT other FUNCTIONS AS ARGUMENTS and they can be PASSED AS ARGUMENTS like other variables**



1. **Functions can be RETURNED as value from a function too**

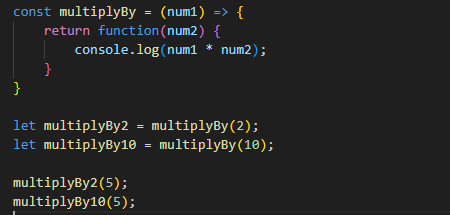


1. **Functions can also accept DEFAULT VARIABLES in case no params are passed**



**Higher Order Functions**

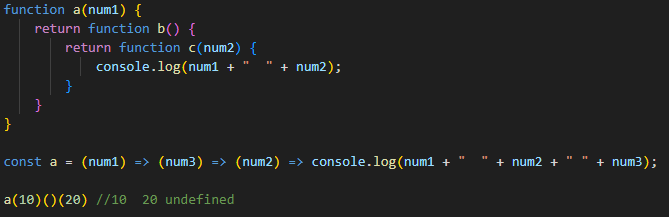
* A “higher-order function” is a function that can **accept** functions as parameters and/or **returns** a function;
* This is Awesome for a number of reasons like code reusability and creating multiple flavours of functions.

In the example to the left, multiply by is returning a new multiplier function which can be used later.

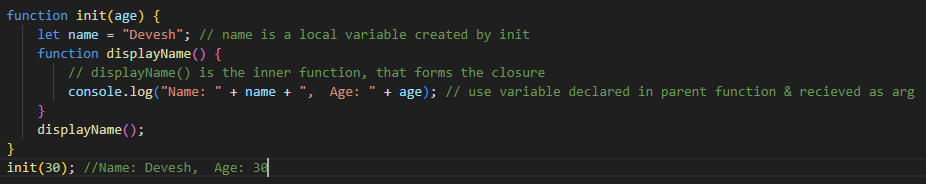
And also, many types of multiplying functions can be created. Example: multiplyBy2, multiplyBy5, multiplyBy10, etc.

**Closures**

* Closure means that an inner function always has access to the variables and parameters of its outer function, even after the outer function has finished executing.
* A closure is the combination of a function and its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.



*function C() has access to all variables num1, num2, num3 due to closure.*



*In the example above displayName() function has access to all variables defined inside init() function or received as arguments.*

**Benefits of Closures**

**1 – Memory Efficient:**



*In the example to the left, every time we use heavyDuty1(). A new Array is created and destroyed. Bad for Memory*

While in case of *heavyDuty2(), array is created only once with heavyDuty2() call. All because of Closure*

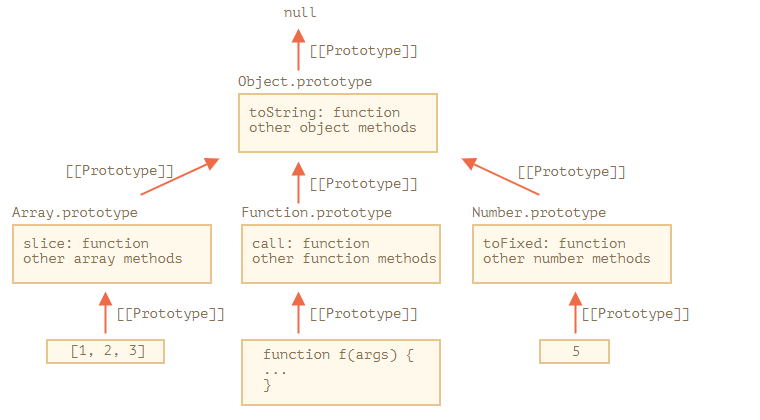
**2 – Encapsulation**

Closure also enables encapsulation in JavaScript. Given below is one such example.

|  |  |
| --- | --- |
|  |  |
| *Because of the closure, view can be set to “Nice View” from outside, we allow init() to be called only once from outside.* | In above example, every time setTimeOut is called by JS. It’s shipped to Web Api. By the time it comes back from callback queue index value is already 4.  When we wrap the setTimeOut inside IIFE and pass the index value through IIFE, It creates a closure and setTimeOut is shipped to web api with index value of the loop. |

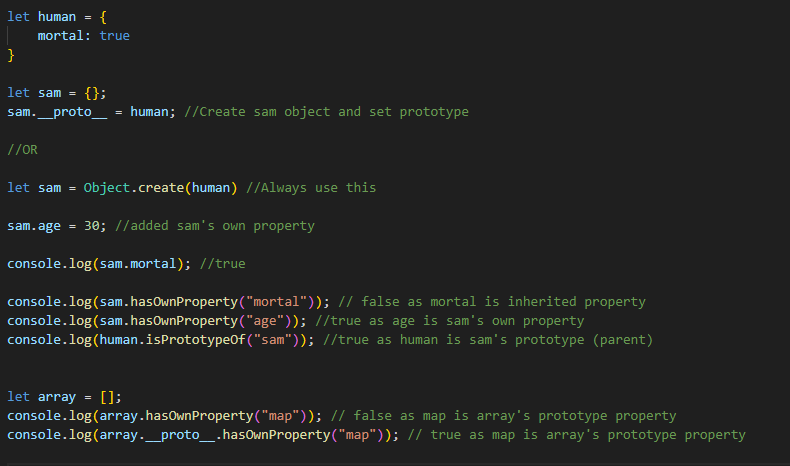
**Prototypical Inheritance**

* In JS each **CHILD OBJECT** in JavaScript has a **\_\_proto\_\_** property which holds a link to **PARENT OBJECT** called “**prototype”**. The Parent Object is also called **Prototype of child object**. Parent object has its own parent (Prototype) until final Parent is Object Prototype. Object class also has its prototype which is **null**. null has no prototype, and acts as the **final link** in this **prototype chain**.
* **Special Note:** Only functions in JavaScript have a **built-in property**, which is called **prototype**.
* So, **Parent object** is also called **prototype** and functions also have property **prototype**.



We can create a prototype of custom object by using **Object.create()** method or **setting the \_\_proto\_\_ property** to another object. **Always use Object.create** though.

We can use **hasOwnProperty**() method to find out object's own properties not inherited ones. We can use **isPrototypeOf**() method to find out if an object is parent of another object as indicated below.



**JavaScript OOP**