**JS MILESTONE 6**

Mutable / Immutable concepts - How it works

Primitive and Reference Values

● Dynamic Properties

● Copying Values

● Argument Passing

● Determining Type

Execution Context and Scope

● Scope Chain Augmentation

● No Block-Level Scopes

1. Mutable/ Immutable concepts:

* Mutability refers to data types that can be accessed and changed after they've been created and stored in memory.
* Immutability, on the other hand, refers to data types that you can't change after creating them – but that you can still access in the memory.
* Primitive datatypes are immutable.
* Non-primitive/ Reference data types are mutable.
* Primitive datatypes are built-in datatypes provided by JS. String, number, Boolean, null, symbol, undefined are primitive datatypes in JS
* Non-primitive datatypes can store collection of values or more complex entities. Functions, arrays, objects
* Reference data does not copy values, but rather pointers.
* So, when a variable is created its ‘variable name’ is stored in stack and it is pointed to its value (ie., object/ array) in heap.

EX: const staff = {

name: "Strengthened",

age: 43,

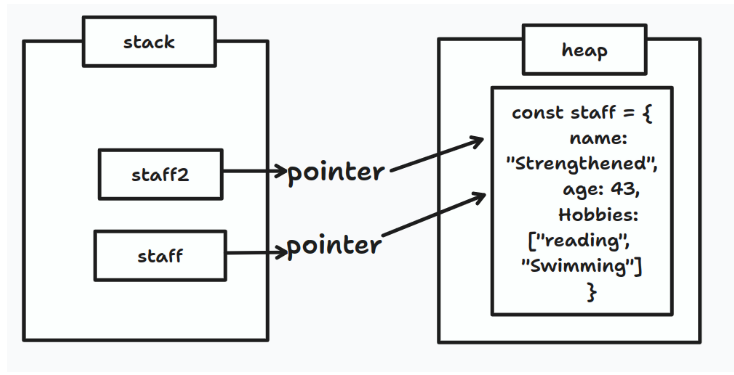
Hobbies: ["reading", "Swimming"]

}

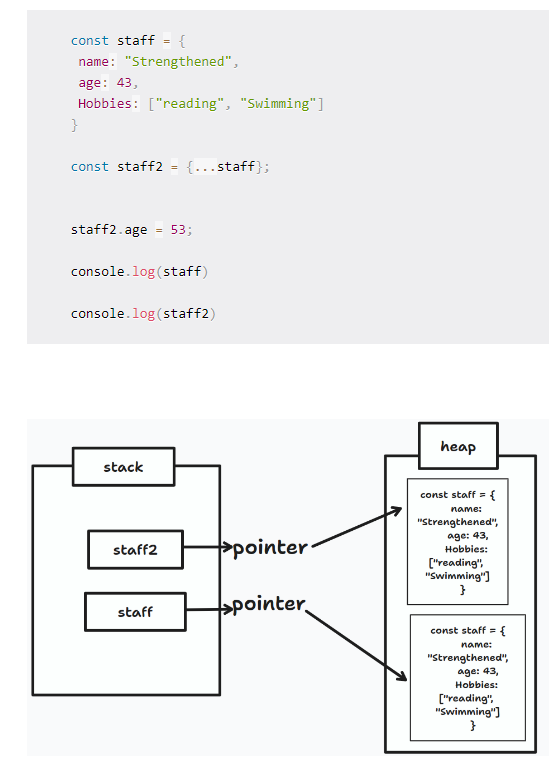
const staff2 = staff;

console.log(staff);

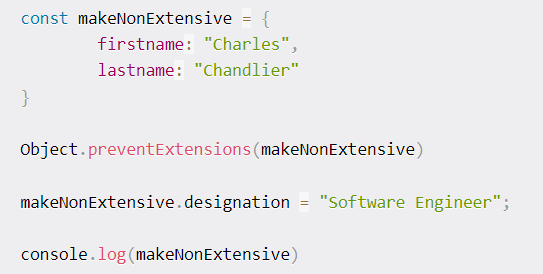
console.log(staff2);



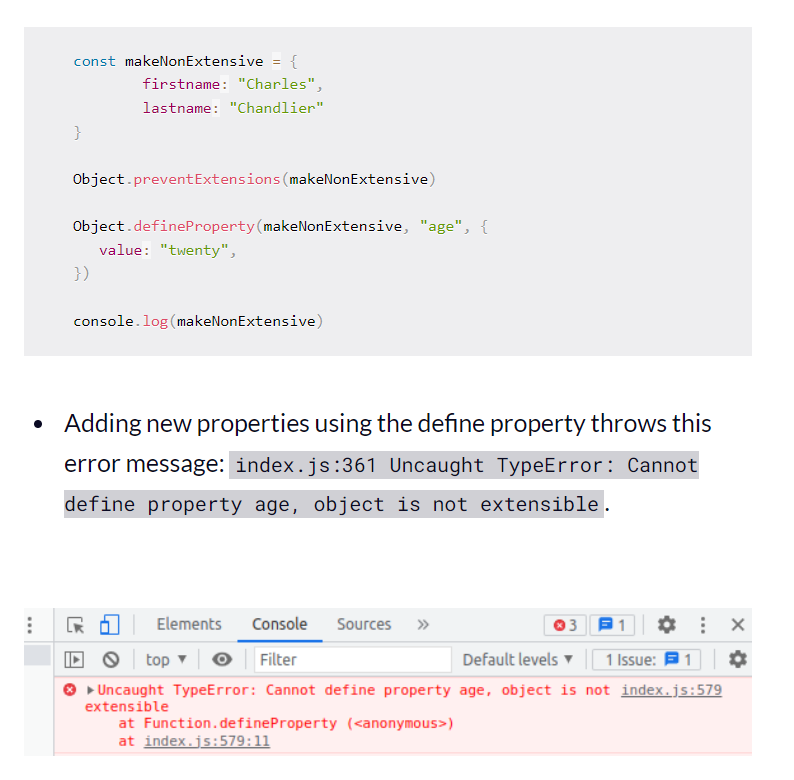
* This can also be overridden by using spread operator/ Object.assign() property.

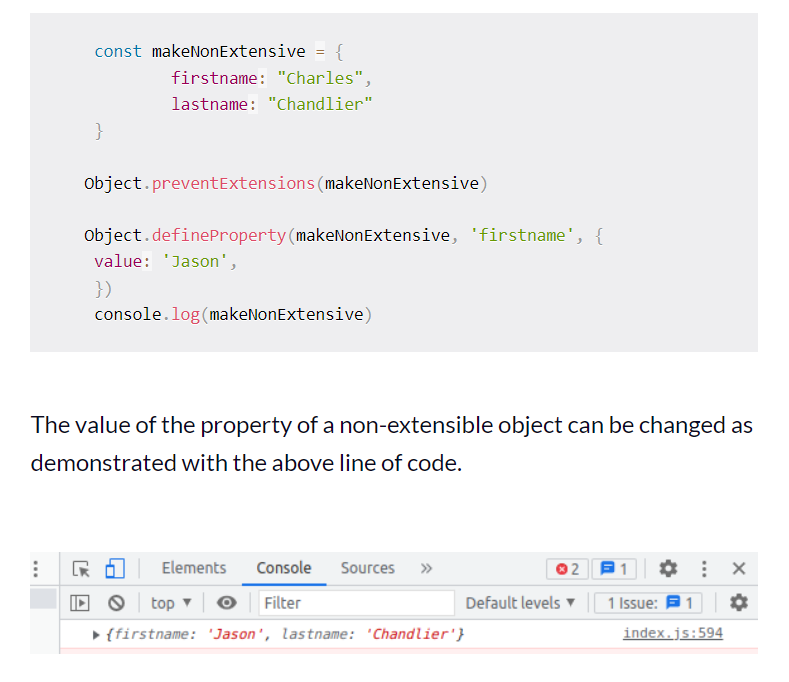
* To prevent mutability, you can use Object.preventExtensions(), Object.seal(), Object.freeze()
* .preventExtensions() **doesn’t add property**, also don’t shows error. It consoles existing Object without any changes.



* By using Object.defineProperty() we get error message displayed



* But value of existing property can be modified.



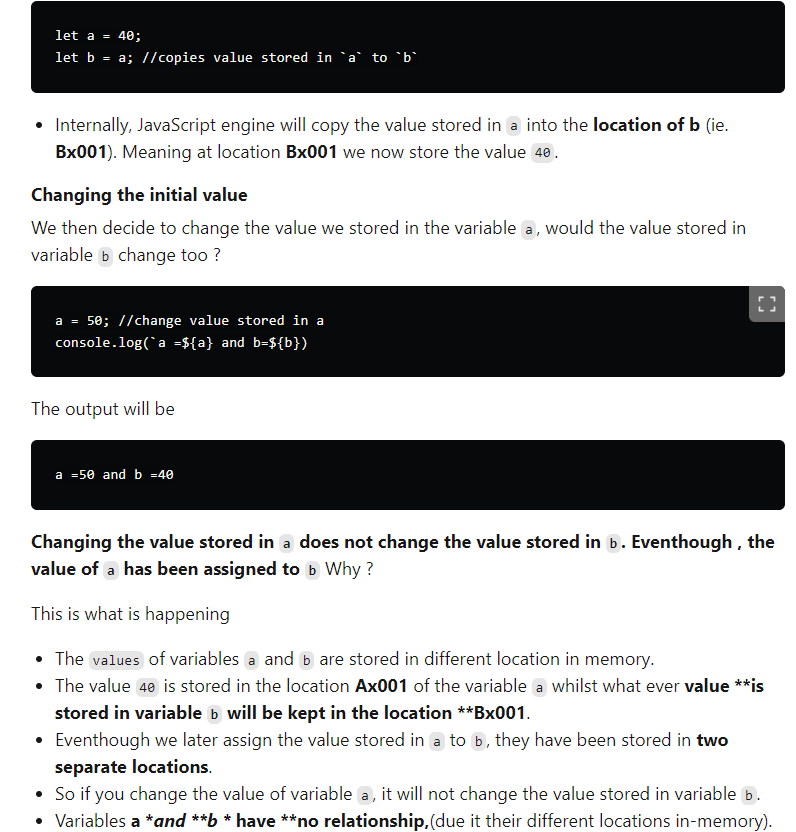
* To delete an object



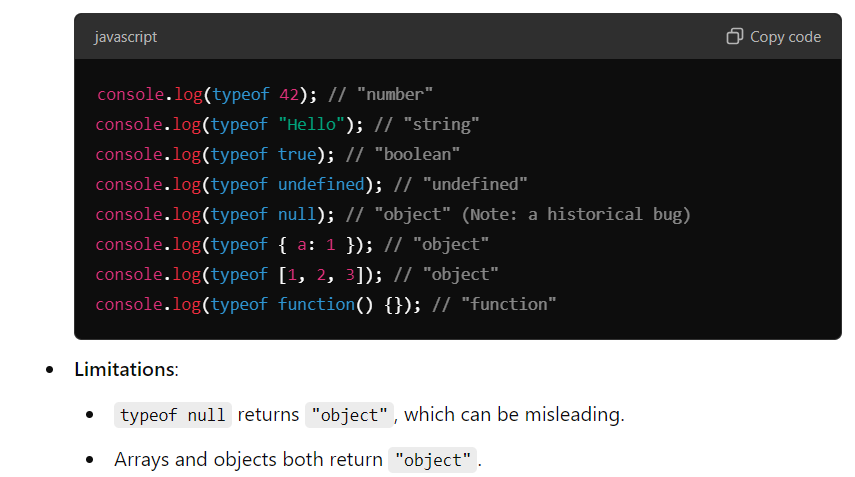
* Object.seal() 🡪 All objects in Javascript are extensible by default. Just as the name suggests, this method seals an object. You cannot add new properties to a sealed object or delete an existing property from a sealed object. But object.seal permits modifying existing properties.
* When a variable is declared with **const** that variable can’t be changed. But when we declare variable with **let** it can be redeclared.

1. Primitive and reference values

* Primitive values are data that are stored on the stack.
* Primitive value is stored directly in the location that the variable acesses.
* Reference values are objects that are stored in the heap
* Reference value stored in the variable location is a pointer to a location in memory where the object is stored.
* when in a primitive datatype if one variable’s content is copied to another variable, and changed the content of any variable, another variable remains unaffected.



* Any changes in non-primitive datatype after storing and copying it in variables, both variables will be affected. This is because variable is being stored in stack and values are store in heap.
* **Argument Passing:**
  + Pass-by-Value
  + Pass-by-reference
  + JS always passes arguments by value.
  + Pass by Value means that when you pass a variable to a function, JavaScript creates a copy of the variable’s value and uses it inside the function. This means any changes made to the variable inside the function do not affect the original variable outside the function.
  + Pass by Reference means that when you pass a variable (specifically, objects or arrays) to a function, JavaScript passes the reference or memory address of the variable, not a copy. This means any changes made to the variable inside the function will affect the original variable outside the function.
  + If the complete reference is changed inside the function it will not be reflected in original object ie., reassigning reference in function will not modify the original object.
  + Modifying original object in function changes the original object.
* Determining type:
  + Typeof operator is used to determine the type of the variable.



* + instanceOf operator is used to get the type of array/ objects mainly. It is used to get date object too.
  + Array can be checked using arr.isArray()

[shallowcopy-deepcopy](https://www.freecodecamp.org/news/copying-stuff-in-javascript-how-to-differentiate-between-deep-and-shallow-copies-b6d8c1ef09cd/)

DEEP COPY:

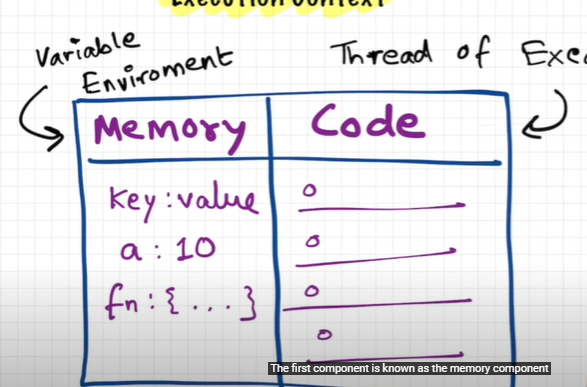
* When original copy is not modified after making changes in duplicate copy, it is known as deep copy.
* Variables under primitive datatype follows deep copy.

SHALLOW COPY:

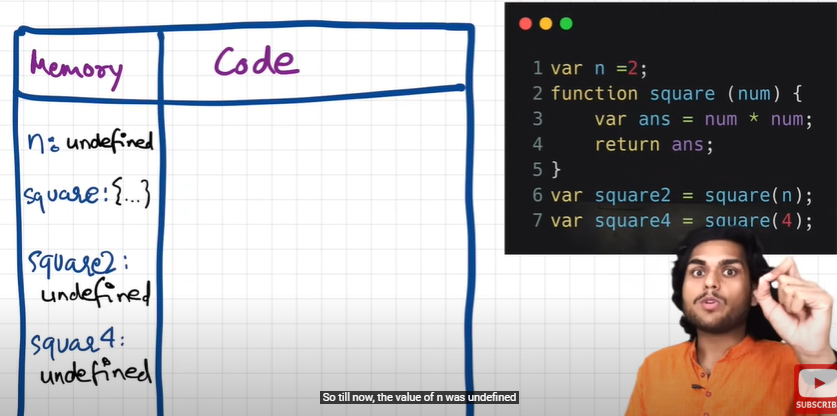
* When original copy is modified after making changes in duplicate copy, it is known as shallow copy.
* This often creates a problem.
* To overcome problems faced when accessing original copies, SPREAD OPERATOR/ OBJECT.ASSIGN() can be used.
* Objects, arrays follow shallow copy.
* By using spread operator in nested objects again shallow copy is made.
* To overcome this, objects are to be coped **let b = JSON.parse(JSON.stringify(a))**

1. Execution Context and Scope

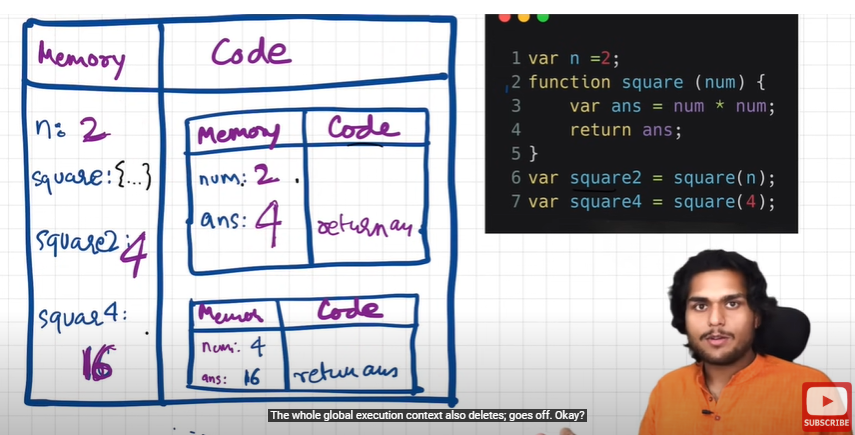
* When a JS program is made run, execution context is created.
* Everything in JS happens inside execution context.
* Execution context maybe a box/ container where all JS code is executed inside this.



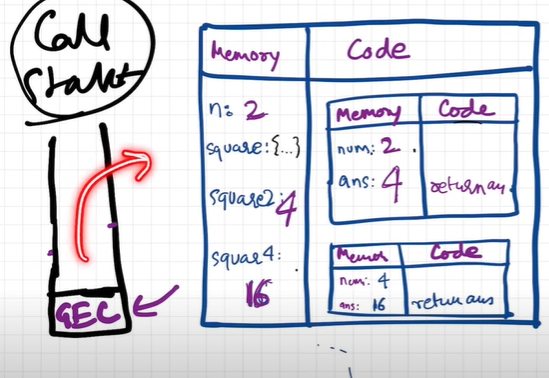
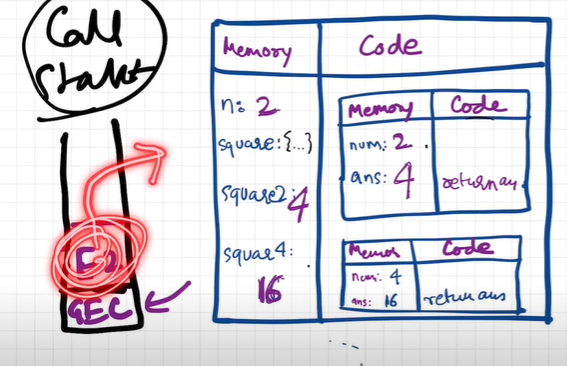
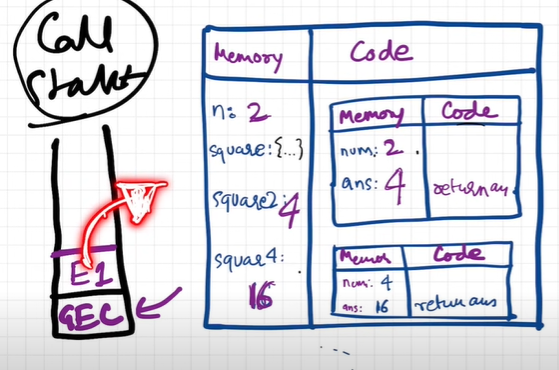
* Execution context has 2 components in it.
  + Memory component/ variable environment – all the variables and functions are stored in key-value pairs.
  + Code component/ thread of execution – all codes execute line by line.
* JS is a synchronous single-threaded language.
* 1st phase of executing code until line 7. Just assigning values. Memory execution phase



* Again JS compiler goes from the starting of the code and undergoes 2nd phase. Code execution phase



* After completing one execution function it’ trace will be deleted in global execution context.
* This is achieved by call stack. Here, global execution context🡪ex c1 🡪 ex c1 popped 🡪ex c2 🡪 ex c2 popped 🡪 global execution context popped.



* To keep the track of all the contexts, including global and functional, the JavaScript engine uses a **call stack**. A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.
* It uses the LIFO principle (Last-In-First-Out). When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.
* When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.

1. Scope:

* Scope determines the accessibility of variables, objects, and functions from different parts of the code.
* There are 3 types of scope
  + Block scope
  + Function scope
  + Global scope
* By introducing **let and const** block scope comes to play
* Variables declared with **let** inside a { } block cannot be accessed from outside the block:
* Variables declared with **var** inside a { } block can be accessed from outside the block.
* LOCAL SCOPE - Variables declared within a JavaScript function, are local to the function:
* Function scope - Variables defined inside a function are not accessible (visible) from outside the function.
* Global Scope - A variable declared outside a function, becomes GLOBAL. All scripts and functions on a web page can access it.
* If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable.
* myFunction();  
  // code here can use carName  
  function myFunction() {  
    carName = "Volvo";  
  }
* In "Strict Mode", undeclared variables are not automatically global.
* No block-level scope: Variables declared with the var keyword can NOT have block scope. Variables declared inside a { } block can be accessed from outside the block.