Js Solids - Under the Hood Part 1

callbacks,
higher-order functions,
closure

Basic JS principles

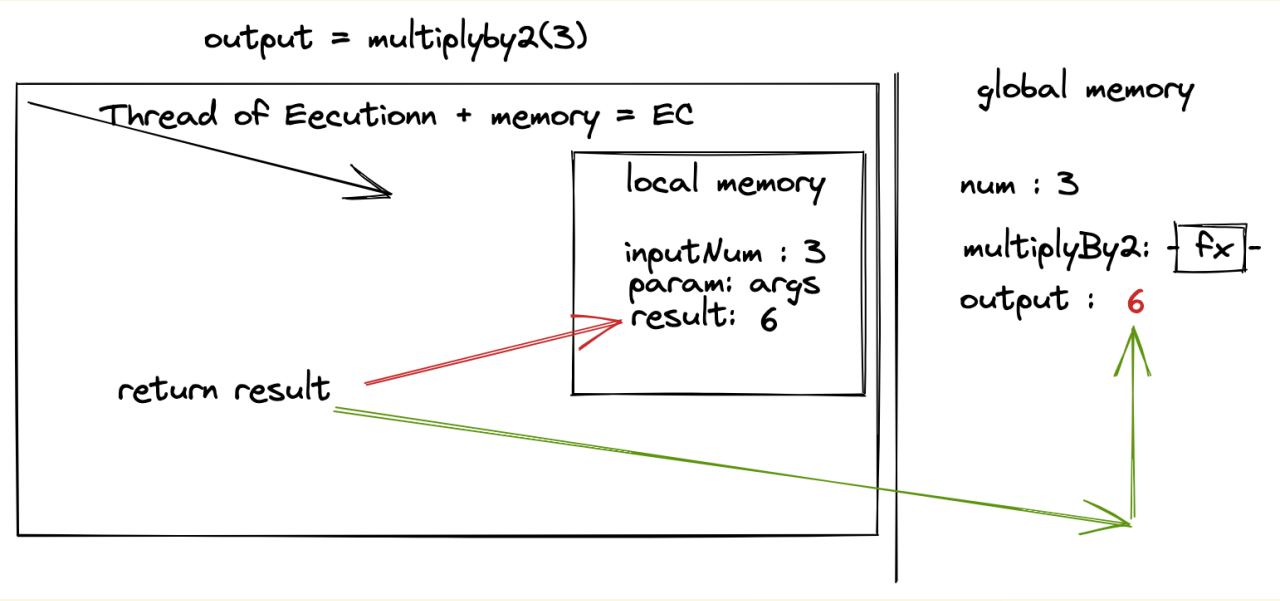
1. Js is single threaded i.e it goes through the code line by line. executing each line known as thread of execution or TOE
2. It saves data like array and objects in memory and can access them later.

Terminology 1

- TOE - Thread of execution, in js the code is executed line by line
- global memory - where data is stored
- Execution context - made up of two parts TOE and local memory

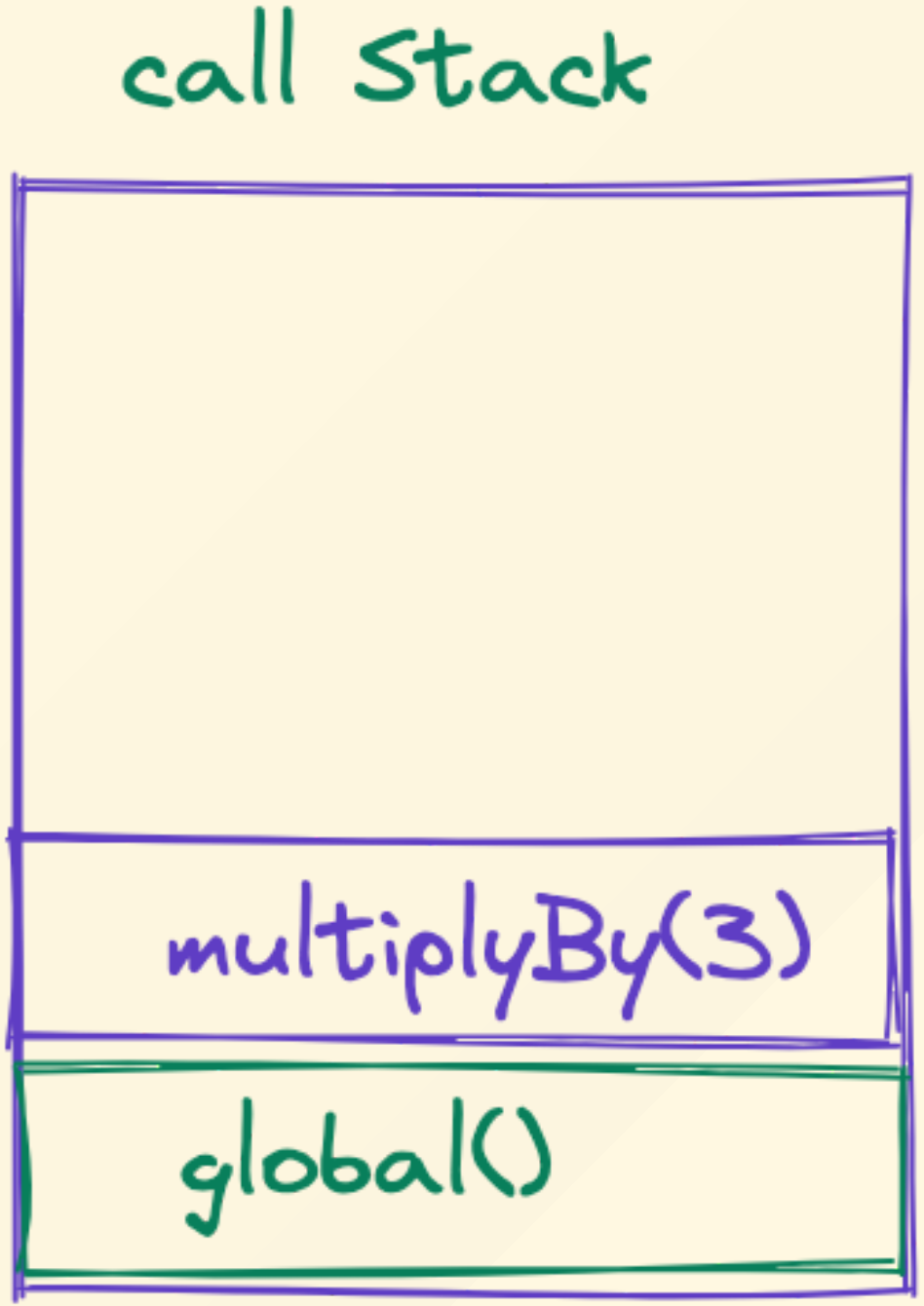
Code 1 : Thread of execution and memory

```
const num = 3;
function multiplyBy2(inputNumber){
const result = inputNumber * 2;
return result;
}
const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```



Explanation Code 1

- Js engine creates a global execution context and pushes it to the call stack.
- It then creates a memory space for the variable num and stores the value 3 in it.
- It then creates a memory space for the function multiplyBy2 and stores the function definition in it.
- it then creates a memory space for the variable output and stores the value of the function multiplyBy2(num) in it.



Call Stack

Js keeps track of what function is currently
* running a function add to calls stack
* done running a function remove from calls stack

- Call stack - where execution contexts are stored
- currently being executed functions are on top of call stack

Generalized function

- A function that can be used for multiple inputs is called a generalized function.
- for example take three functions, twoSquare, threeSquare and fourSquare.
- we can now generalize these functions into one function called square and pass the number as an argument to the function.
- this is called a generalized function, following DRY principle (Don't Repeat Yourself)

Code Snippet Generalized function

```
function twoSquare(){
return 2*2;
}
function threeSquare(){
return 3*3;
}
function fourSquare(){
return 4*4;
}
function square(number){
return number*number;
}
```

Higher Order Function

- A function that accepts another function as an argument is called a higher order function.
- A function that returns another function is also called a higher order function.
- A function that accepts another function as an argument and returns another function is also called a higher order function.

Code Snippet Higher Order Function

```
function callTwice(func){
func();
func();
}
function callTenTimes(func){
for(let i=0;i<10;i++){
rollDie();
}
}
function rollDie(){
const roll = Math.floor(Math.random()*6)+1;
console.log(roll);
}
callTwice(rollDie);
callTenTimes(rollDie);
```

Why higher order functions?

- They allow us to abstract over actions, not just values.
- take and example of three functions, copyArrayAndMultiplyBy2, copyArrayAndDivideBy2 and copyArrayAndAddBy3.

code snippet copyArrayAndMultiplyBy2

```
function copyArrayAndMultiplyBy2(array){
const output = [];
for(let i=0;i<array.length;i++){
output.push(array[i]*2);
}
return output;
}
const myArray = [1,2,3];
const result = copyArrayAndMultiplyBy2(myArray);
```

code snippet copyArrayAndDivideBy2

```
function copyArrayAndDivideBy2(array){
const output = [];
for(let i=0;i<array.length;i++){
output.push(array[i]/2);
}
return output;
}
const myArray = [1,2,3];
const result = copyArrayAndDivideBy2(myArray);
```

code snippet copyArrayAndAddBy3

```
function copyArrayAndAddBy3(array){
const output = [];
for(let i=0;i<array.length;i++){
output.push(array[i]+3);
}
return output;
}
const myArray = [1,2,3];
const result = copyArrayAndAddBy3(myArray);
```

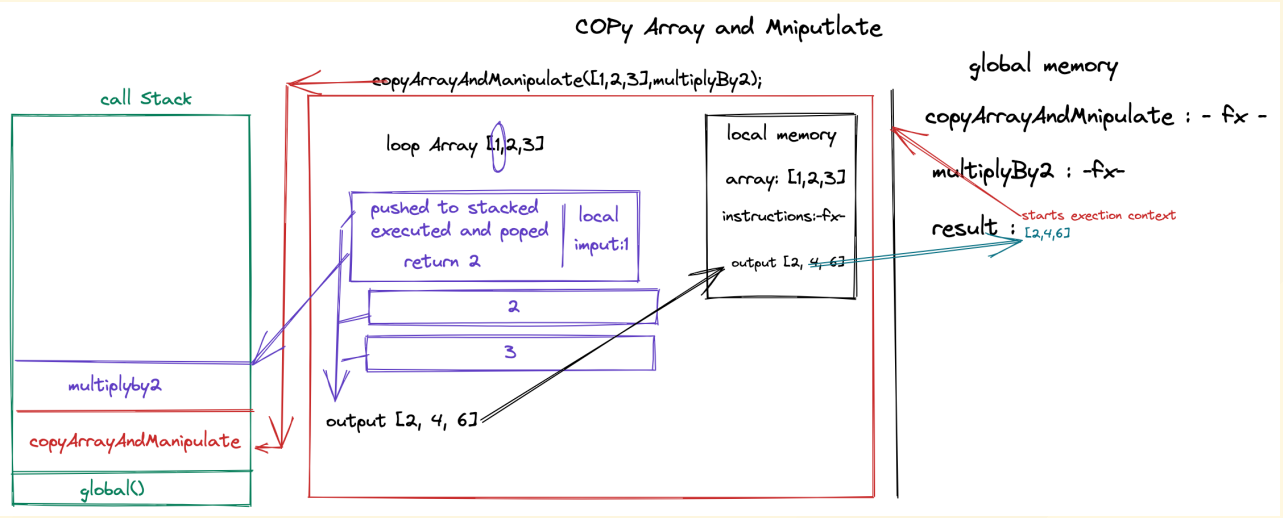
There must be a better way ..

Higer order functions and **callback functions** to the rescue!

- We can generalize the above three functions into one function called `copyArrayAndManipulate`.
- This function takes two arguments, an array and a function.
- The function argument is called a callback function.
- The callback function is called inside the `copyArrayAndManipulate` function.
- callback function is added to the call stack and executed.
- once completed it is removed from the call stack.
- This is the idea behind **Higher Order Functions**.

Code Snippet copyArrayAndManipulate

```
function copyArrayAndManipulate(array, instructions){
const output = [];
for(let i=0;i<array.length;i++){
output.push(instructions(array[i]));
}
return output;
}
function multiplyBy2(input){
return input*2;
}
const result = copyArrayAndManipulate([1,2,3],multiplyBy2);
// under the hood copyArrayAndManipulate is map
```



Explanation copyArrayAndManipulate

- The function `copyArrayAndManipulate` is stored in the global memory
- The function `multiplyBy2` is stored in the global memory
- result variable is created in the global memory and an execution context is created for the function `copyArrayAndManipulate` and pushed to the call stack.
- in the execution context of `copyArrayAndManipulate` the array argument is stored in the memory space of the variable array.
- the instruction argument is stored in the memory space of execution context
- the output label is created in the memory space of the execution context and an empty array is stored in it.
- the for loop is executed and the array is iterated over.
- for each loop the callback function `multiplyBy2` is called and the value of the array is passed as an argument to the function. multiplyBy2 is pushed, executed and removed from the call stack for each iteration.
- once the for loop is completed the output array is returned and the execution context of `multiplyBy2` is removed from the call stack.
- the result variable is assigned the value of the output array and the execution context of the `copyArrayAndManipulate` is removed from the call stack.

Arrow Functions

- Arrow functions are short hand way of writing functions.
- we could instead of writing the function `multiplyBy2`, `divideBy2` and `addBy3` pass in anonymous function using arrow functions.
- arrow function can return implicitly without the need of the return keyword when the function body is a single expression.

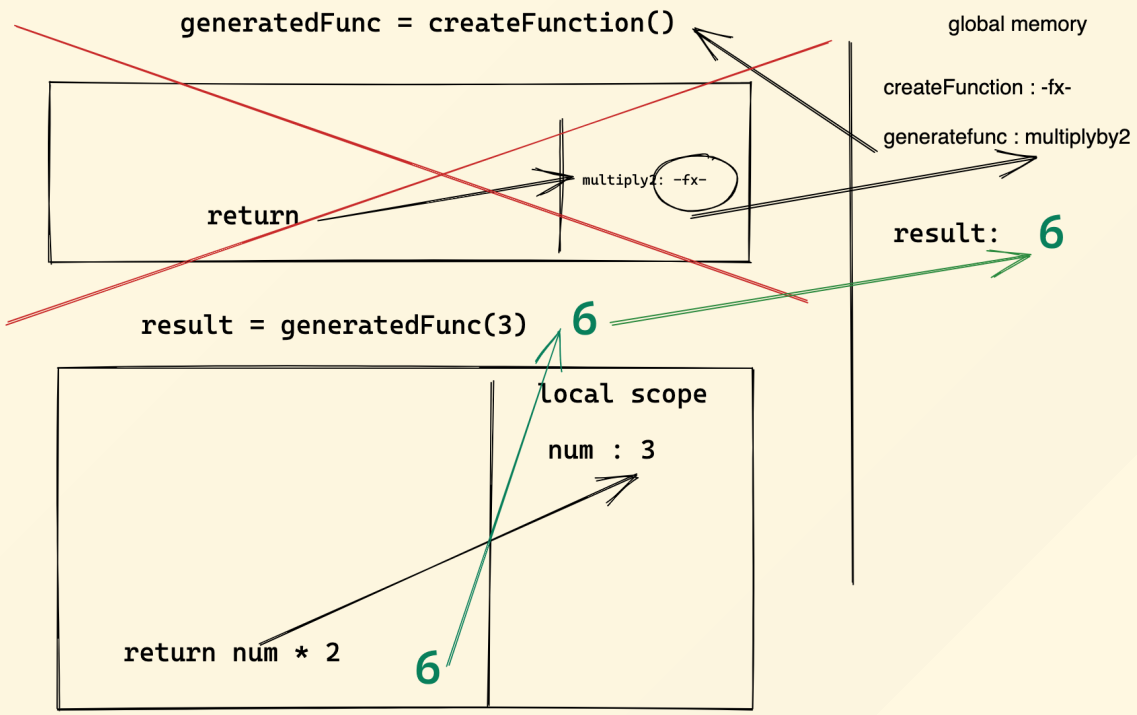
```
const result = copyArrayAndManipulate([1,2,3],(input)=> input*2);
const result2 = copyArrayAndManipulate([1,2,3],(input)=>input/2);
const result3 = copyArrayAndManipulate([1,2,3],(input)=>input+3);
```

CLOSURES!!!!

- A closure is a function that makes use of variables defined in outer functions that have previously returned.
- What if we can have a function with permanent memory of the variables that were in scope when the function was created.
- state, variables environment, local memory.

Code Snippet Closure

```
function createFunction(){
  function multiplyBy2(num){
    return num*2;
  }
  return multiplyBy2;
}
const generatedFunc = createFunction();
const result = generatedFunc(3); // 6
```



Explanation Closure

- The function `createFunction` is stored in the global memory.
- const generatedFunc is created in the global memory and an execution context is created for the function `createFunction` and pushed to the call stack.
- the returned function `multiplyBy2` is stored in the memory space of the variable `generatedFunc`.
- const result is created in the global memory and an execution context is created for the function `multiplyBy2` and pushed to the call stack.
- noticed `multiplyBy2` is being called with the argument 3, instead of createFunction this is because the function `multiplyBy2` is stored in the memory space of the variable `generatedFunc` and not the function `createFunction`.
- the return value 6 is stored in the memory space of the variable result and the execution context of `multiplyBy2` is removed from the call stack.

Why not just define the function globally?

```
function outer () {
  let counter = 0;
  function incrementCounter () {
    return counter++;
  }
  console.log(incrementCounter);
 return incrementCounter;
}
const myNewFun = outer();
myNewFun();
myNewFun();
```

link to surrounding data [[scope]]

- The function `outer` is stored in the global memory.
- outer has backpack [[scope]] which contains the variable counter.
- [[scope]] is a hidden property which is private and cannot be accessed directly.à
- this backpack called lexical environment is created when the function is created.
- the data in lexical environment is persisted, is data, is linked by a scope property : scope is the rule as in whats available to me.
- this scope rule is called lexical scope or state scope.
- not where i run the function but **where i define the function!**!! thats lexical scope!!!

Closure as professional code

- Helper functions are used to keep the code DRY, can check the backpack [[scope]] to see if was already ran.
- Memoization is a technique used to improve performance by caching the results of expensive function calls and returning the cached result when the same inputs occur again.
- expensive functions can be stored in the backpack where we check if it exits and then return the result.
- module pattern, data privacy, encapsulation, IIFE, etc.
- asynchronous programing rely on closures to maintain state.

Final Closure Code Snippet

```
// Create a function which will hold another function.
function bankAccount() {
  // Create two variables inside of the outer function.
  // We will be accessing the two variables inside of our inner function.
  const checking = 400;
  const savings = 1000;

  // Return a newly created inner function.
  return {
    displayFunds: function () {
      // We have access to our outer functions variable which we console.log.
      // This is a closure. The inner function has access to the outer functions scope.
      console.log(
        `You have $${checking} in your checking account and $${savings} in your savings account`
      );
    },
  };
}

// Create a new variable which holds the `bankAccount` function.
const myBank = bankAccount();

// With our newly created variable call the `displayFunds` method attached to it.
myBank.displayFunds();

// Check the console and expand the given object -> displayFunds -> Scope and then you should be able to visually see your closures.
console.dir(myBank);

// By console logging the outer function's variable we can see that the variables are not able to be accessed.
console.log(checking);
console.log(savings);
```