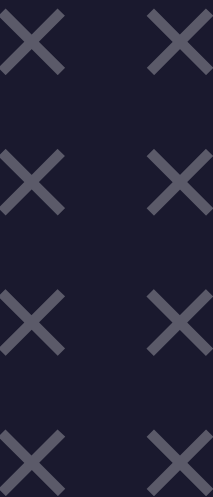




JAVASCRIPT HARD PARTS

By Huthaifa Salman



The Hard Parts of JS

Here some of the topics which considered hard for begginers

01 **Principels of JS**

Using the Execution context and the Call Stack, investigate how JavaScript runs and stores code in memory.

02 **Callback & higher order functions**

Makes our code more declarative and understandable by enabling strong pro-level functions like map, filter, and reduce (a key feature of functional programming).

03 **Closures**

Functions that has access to variables in its lexical scope, even when the function is invoked outside of that scope.

04 **Asynchronous JavaScript**

Method for running JavaScript code in a non-blocking manner. When code is run asynchronously, the interpreter may go to the next line of code before the current line of code has completed.

CONTENTS

In this session

I will just explain these topics from the hard parts, you can read more detailed in-depth explanation about them [here](#)



GitHub Repo

1

Principles of JS

Explaining how JavaScript works with memory and functions, call stack and thread of execution.

2

Closures

Defining closures and the benefits of using a closure in our code.





JavaScript Principles




JavaScript Principles

When JavaScript code is executed, it creates a *global execution context*.

 The *thread of execution* scans the code line by line and executes each line.

 A Global Variable Environment is a live *memory* of variables with data.

 A Call Stack to keep track of the function calls.



```
1  var x = 10;
2  function timesTen(a){
3      return a * 10;
4  }
5  var y = timesTen(x);
6  console.log(y);
```

Execution Context

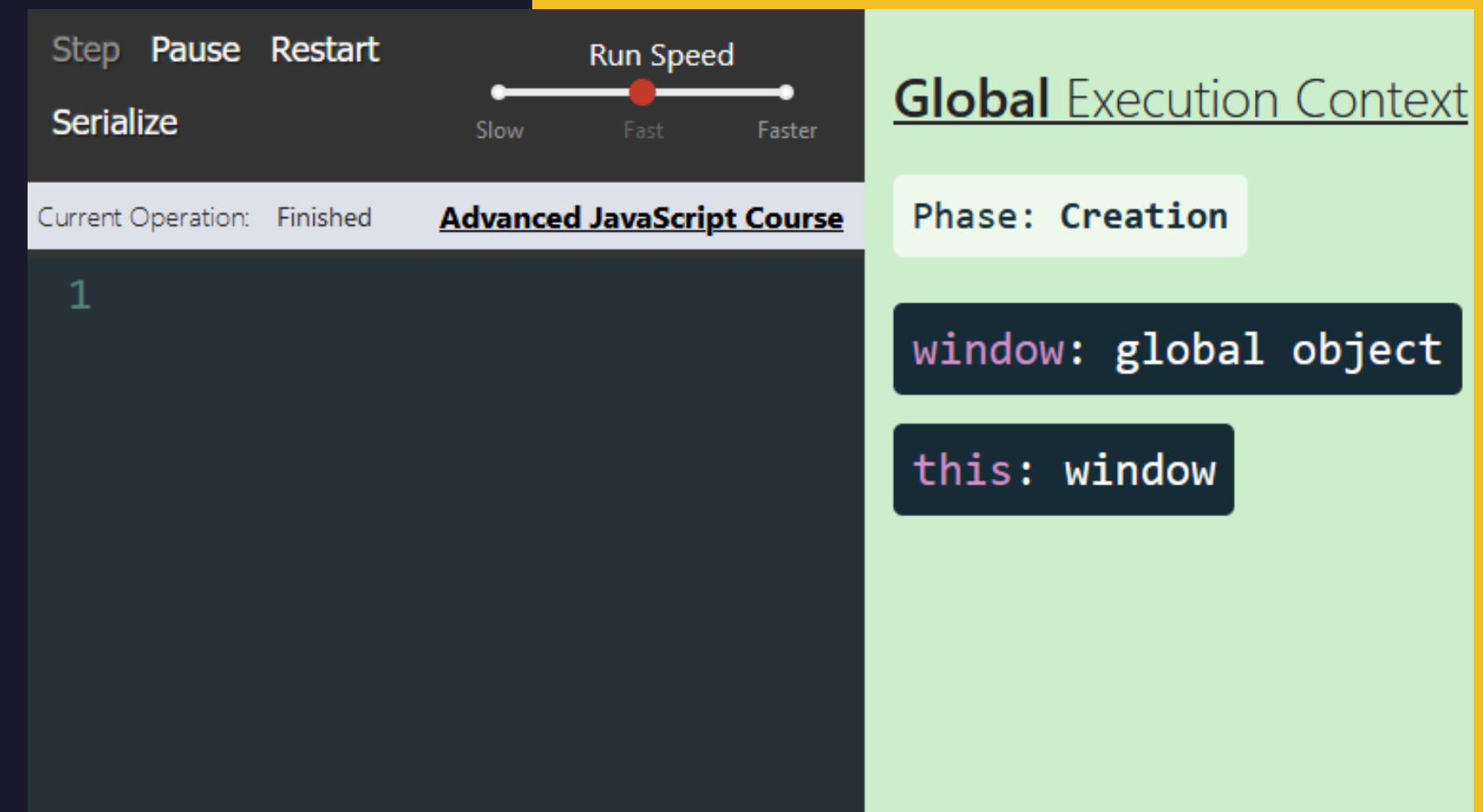
It's a conceptual environment created by JS for code evaluation and execution.

Types of EC:

- **Global Execution Context (GEC):** Highest level of abstraction for an execution context in JS.
- **Function Execution Context (FEC):** Created for every function invocation
- **eval()** Execution Context.

Each Execution Context has 2 phases:

1. **Creation Phase:** A lexical environment is created.
2. **Execution Phase:** Values are stored to variables.



Creation Phase

The global creation phase which will do the following:

1. Create a *global* object.
2. Create an object called *this*.
3. Create the lexical environment which is of two types:
 - a. **Variable Environment:** records bindings created by the var (*undefined*).
 - b. **Lexical Environment:** records both variable (let/const) bindings (*<uninitialized>*) and function declarations (*<func>*).

The screenshot displays a JavaScript execution environment with a dark theme. At the top, there are buttons for 'Step', 'Run', and 'Restart', along with a 'Run Speed' slider set to 'Fast'. Below these, the 'Current Operation' is 'Program' and the title is 'Advanced JavaScript Course'. The main area shows the following code:

```
1 var x = 10;
2 function timesTen(a){
3   return a * 10;
4 }
5 var y = 'name';
```

On the right side, the 'Global Execution Context' is shown. It indicates the 'Phase: Creation' and lists the following bindings:

- window: global object
- this: window
- x: undefined
- timesTen: fn()
- y: undefined

Execution Phase

The global execution phase is the second phase of the execution context, which will start running our code line by line and executing it.

- **x** value changed to 10.
- **y** value changed to "name".

Step Run Restart

Serialize

Run Speed

Slow Fast Faster

Current Operation: Program

Advanced JavaScript Course

```
1 var x = 10;
2 function timesTen(a){
3   return a * 10;
4 }
5 var y = 'name';
```

Global Execution Context

Phase: Execution

window: global object

this: window

x: 10

timesTen: fn()

y: "name"

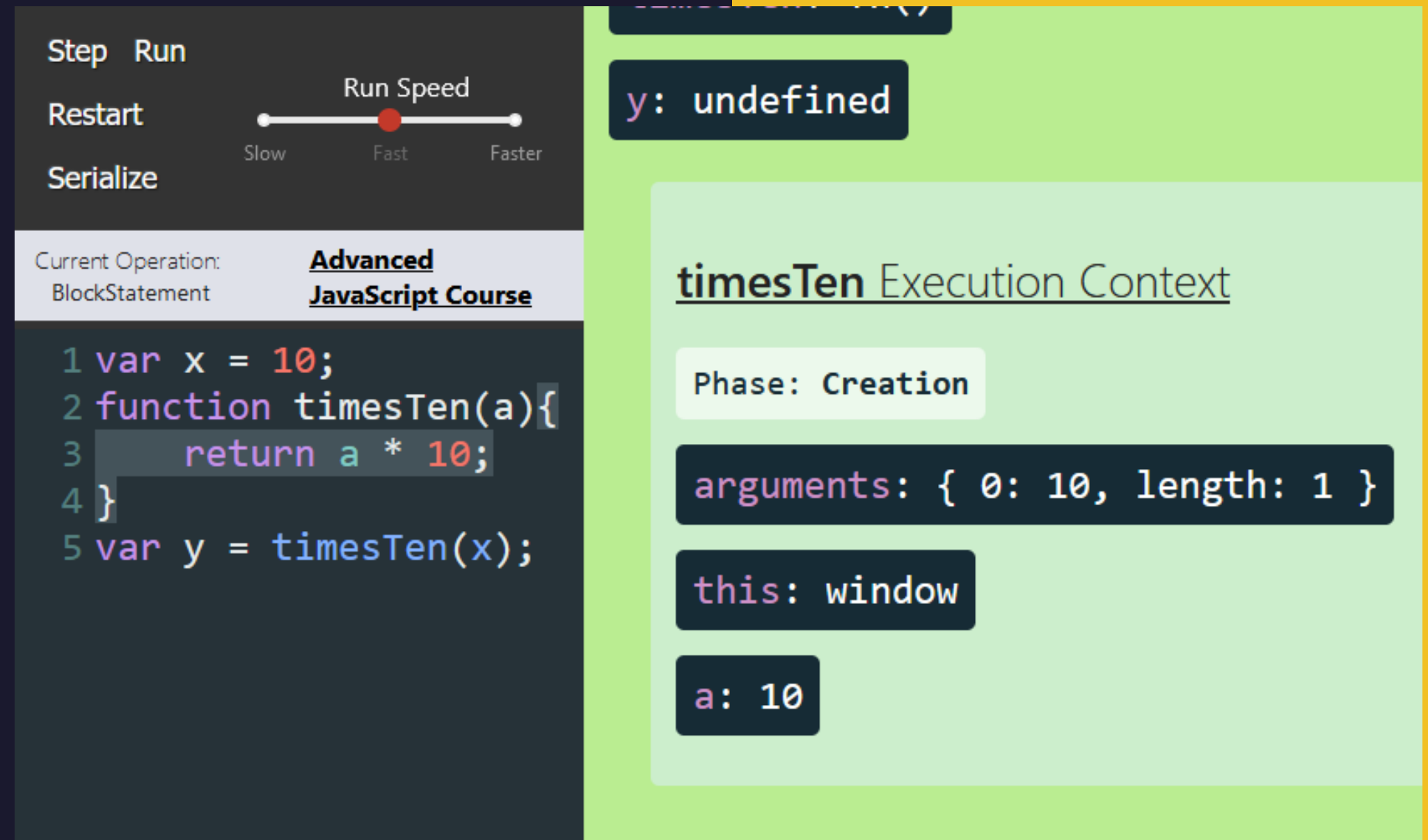
Function Execution Context

Whenever a function is invoked, a new function execution context is created and placed on top of the execution stack (Call Stack).

Like the global, this context has two phases:

1. Creation Phase: instead of creating a global object, it creates an arguments object.
2. Execution Phase: begins running and executing our code line by line.

 JavaScript engine uses Call Stack to keep track of function calls



The screenshot displays a JavaScript engine interface with a dark theme. At the top, there are buttons for 'Step', 'Run', 'Restart', and 'Serialize', along with a 'Run Speed' slider set to 'Fast'. Below this, the 'Current Operation' is 'BlockStatement'. The main code area shows five lines of JavaScript code: `1 var x = 10;`, `2 function timesTen(a){`, `3 return a * 10;`, `4 }`, and `5 var y = timesTen(x);`. The third line is highlighted. To the right, a light green panel titled 'timesTen Execution Context' shows the 'Phase: Creation'. It lists the 'arguments' as `{ 0: 10, length: 1 }`, 'this' as `window`, and a local variable `a` as `10`. Above this panel, a dark box shows `y: undefined`.

Call Stack

The call stack is used to store the return address of the caller function so that the called function knows where to return control when it is finished.

When a program is running, the call stack is constantly changing as functions are called and then returned.

1. When a function gets called, the return address of the caller will be pushed into the stack.
2. When the thread encounter the return expression, it will pop the function call from the callstack and return to the caller address.

Step Run Restart Serialize

Run Speed

Slow Fast Faster

Current Operation: VariableDeclaration

Advanced JavaScript Course

```
1 function outter(){
2   var name = "outter";
3   function inner(){
4     var name = "inner";
5     function deep(){
6       var name = "deep";
7     }
8     deep(); // Last on invoked -- First
             // one popped
9   }
10  inner(); // Second on both
11 }
12 outter(); // First one invoked -- Last
             // one popped
```

inner Execution Context

Phase: Execution

arguments: { length: 0 }

this: window

name: undefined

deep: fn()

deep Execution Context

Phase: Execution

arguments: { length: 0 }

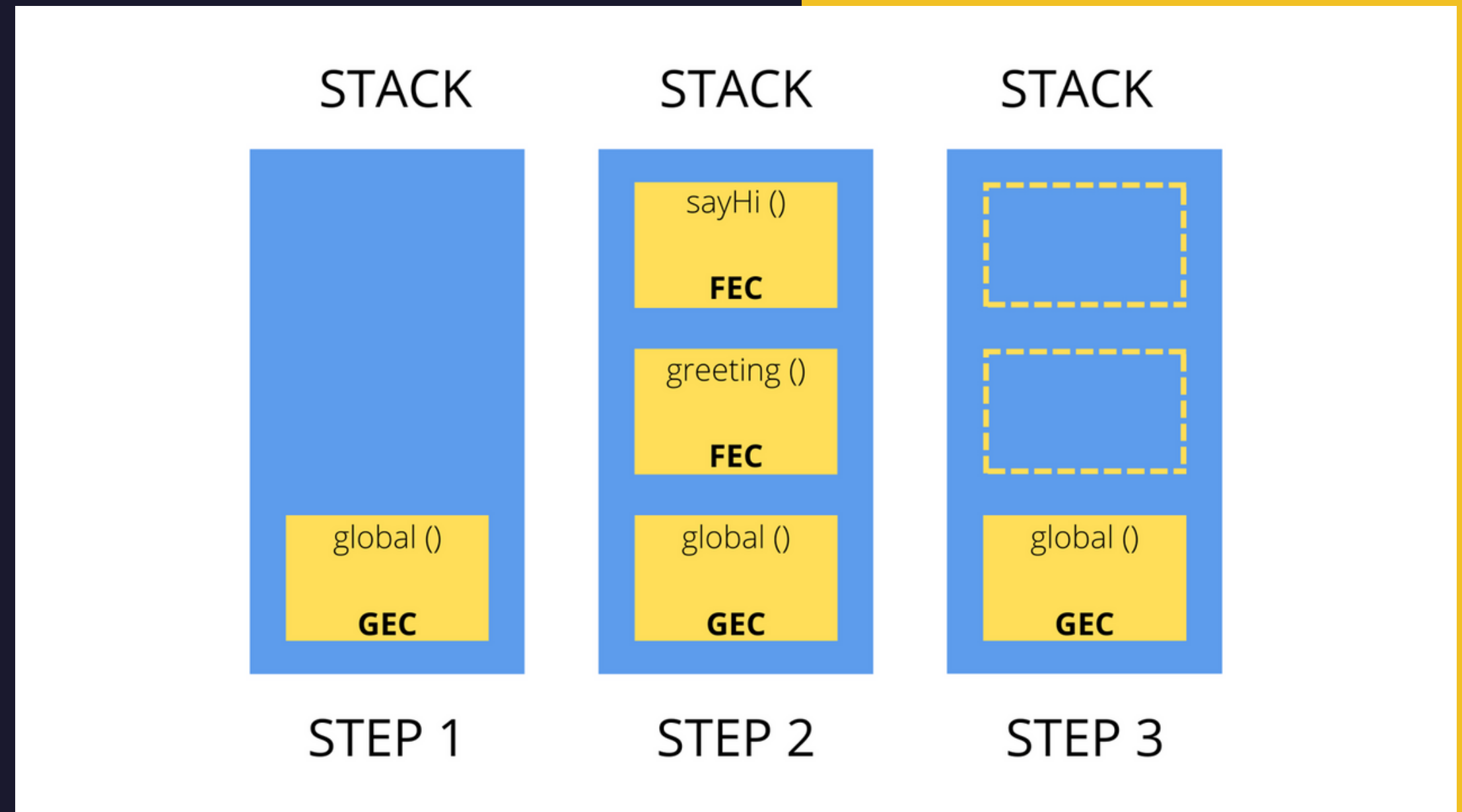
this: window

name: undefined

Call Stack

Here's an example of what happens in the call stack.

- i Note that while the JavaScript engine is running, the Call Stack is never empty, there is the `Global()` call in the bottom of the stack.





Closures



Functions Limitations

- When our functions get called, we create a **live store** of data (**local memory** / **variable environment** / **state**) for that function's execution context.
- When the function finishes executing, gets **popped** from the call stack, and its **local memory** is **deleted** (except the *returned value*).
- So, functions are limited - they forget everything each time they finish running - with no global state.
- What if we could have functions that holds **live data** between each call?
- This will provide our functions with a **permanent/persistent** memory.

Lexical Scope

The first step to understanding closures is to review the lexical scoping rules for nested functions.

The way JS looks for variables, If it can't find a variable in its **local execution context**, it will look for it in its **calling context**. And if not found there in its calling context. Repeatedly, until it is looking in the **global execution context**.

And if it does not find it there, it's **undefined**.

So, a function has access to variables that are defined in its calling context.



```
1  let scope = "global scope";  
2  function checkscope() {  
3      let scope = "local scope";  
4      return scope;  
5  }  
6  checkscope() // => "local scope"
```


Returning Functions

Since functions can return anything, we can return new functions.

.

On line 3 in the `createAdder`'s context, we declare `addNumbers` as a function **local variable** bound with `createAdder` context.

On function **return**, the function execution context is **popped** from execution stack, with local memory **deleted** including the `addNumbers` variable, but the function **declaration** is returned and saved.



```
1  let val = 7
2  function createAdder() {
3      function addNumbers(a, b) {
4          let ret = a + b
5          return ret
6      }
7      return addNumbers
8  }
9  let adder = createAdder()
10 let sum = adder(val, 8)
```

A Closure ...

At first glance, when executing **outer** in **increment** variable it creates a new **FEC** that has **counter** as a **local variable** when returning from the function that **local variable** is **deleted** alongside with the **local memory**.

When inside of the **FEC** of the returned function declaration from **outer**, it has a reference for a variable called **counter**, it looks for it until it reaches the **GEC** and then defines it as **undefined**, this considered as **NaN**, so if we log the **counts** variables the output will be the same **NaN**.



```
1  function outer() {  
2      let counter = 0;  
3      function incrementCounter() {  
4          counter = counter + 1;  
5          return counter;  
6      }  
7      return incrementCounter;  
8  }  
9  const increment = outer();  
10 let count1 = increment();  
11 let count2 = increment();  
12 let count3 = increment();
```


But is that correct?

The Confusion ...

Our earlier explanation was completely incorrect based on the results of the variables.

There must be a some sort of mechanism for a function to access a deleted variable and modify it over each call.

Whenever we **declare** a new function and assign it to a **variable**, you store the **function definition**, as well as a **closure**. The **closure** contains *all the variables that are in scope at the time of creation of the function*. It is analogous to a **backpack**.



```
1 console.log(count1, count2, count3);
2 //           1       2       3

1 function outer() {
2     let counter = 0;
3     function incrementCounter() {
4         counter = counter + 1;
5         return counter;
6     }
7     return incrementCounter;
8 }
9 const increment = outer();
10 let count1 = increment();
11 let count2 = increment();
12 let count3 = increment();
13 console.log(count1, count2, count3);
```

The Closure

After explaining why it happens, let's go through it again, when defining `incrementCounter` function, it gets a **bond** to the surrounding Local Memory The "**Variable Environment**" in which it has been **defined**.

When calling `outer`, we maintain the **bond** to the `outer`'s **live local memory** which is **returned** with the new function definition.

So, when `increment` is called, first it looks for `counter` variable in its **local memory**, then in its **variable environment** "**Backpack**".

The screenshot shows a JavaScript IDE with a code editor and a console. The code defines an `outer` function that creates a `counter` variable and an `incrementCounter` function. `incrementCounter` increments `counter` and returns it. `outer` returns `incrementCounter`. Below the code, the console shows the state of the closure scope.

```
function outer() {  
  var counter = 0;  
  function incrementCounter()  
  {  
    counter = counter + 1;  
    return counter;  
  }  
  return incrementCounter;  
}  
var increment = outer();  
var count1 = increment();  
var count2 = increment();  
var count3 = increment();
```

Run Speed: Slow Fast Faster
Current Operation: VariableDeclaration **Advanced JavaScript Course**

count2: undefined
count3: undefined

Closure Scope

- arguments: { length: 0 }
- this: window
- counter: 2
- incrementCounter: fn()

Individual Closures

The bond is created when defining a function, based on it, everytime we call **outer** function, there is a new **counter** variable is created and assigned to **0** and we are defining a new **incrementCounter** function.

So technically each **counter** function in the global has access to an **individual set of data** inside it's **lexical environment**.



```
1  function outer() {  
2      let counter = 0;  
3      function incrementCounter() {  
4          counter = counter + 1;  
5          return counter;  
6      }  
7      return incrementCounter;  
8  }  
9  const counter1 = outer(); // counter = 0  
10 let count1 = counter1(); // counter = 1  
11 const counter2 = outer(); // counter = 0  
12 let count2 = counter2(); // counter = 1
```

Once Function

This helper function converts any function to run **only once**, any further calls will return the **first returned value**.



This is similar to the **singleton pattern**.



There is also other helper function called **memorize**, has the same principle but it stores each run with it returned value "in **a hashmap**", so it won't run if you give it a **duplicate value**.



```
1  function once(func) {
2      let called;
3      return function (...args) {
4          if (!called) {
5              called = func(...args)
6          }
7          return called;
8      }
9  }
10 function temp(value) {
11     //some code here...
12 }
13 const runMeOnce = once(temp);
14 runMeOnce('some value'); // executed
15 runMeOnce('some value'); // returned same value!
```

Private Data

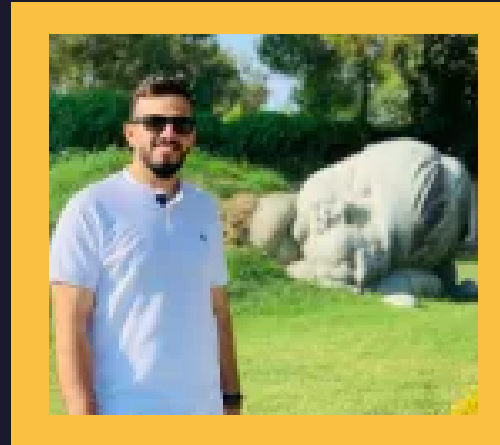
This function provides a simple way to hide variables, and to control the changes made on it, this will prevent unwanted changes for that variable.



This is used to **encapsulate data** just like in **OOP**.

```
1  function makeCounter() {
2      let privateCounter = 0;
3      function changeBy(val) {
4          privateCounter += val;
5      }
6      return {
7          increment: function () {
8              changeBy(1);
9          },
10         decrement: function () {
11             changeBy(-1);
12         },
13         value: function () {
14             return privateCounter;
15         }
16     }
17 }
18 const counter1 = makeCounter();
19 counter1.increment();
20 counter1.decrement();
```

Session Organizers



Tamer Naana, Lead

Trainer, Supervisor



Huthaifa Salman, Intern

Author



THANKS
FOR WATCHING