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.

```
var x = 10;
function timesTen(a){
    return a * 10;
}
var y = timesTen(x);
console.log(y);
```

Execution Context

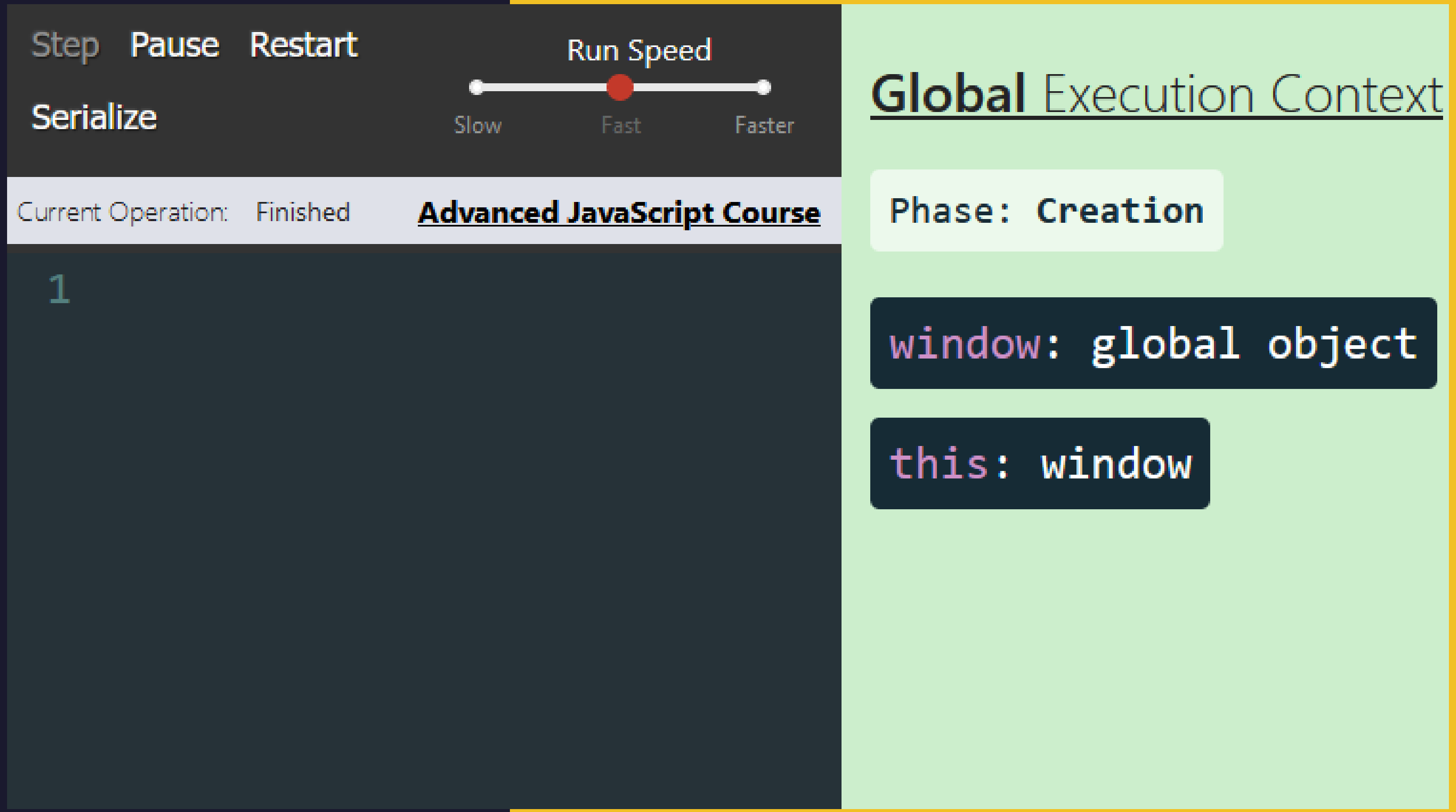
It's a conceptual environemnt created by JS for code evaluation and execution.

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:

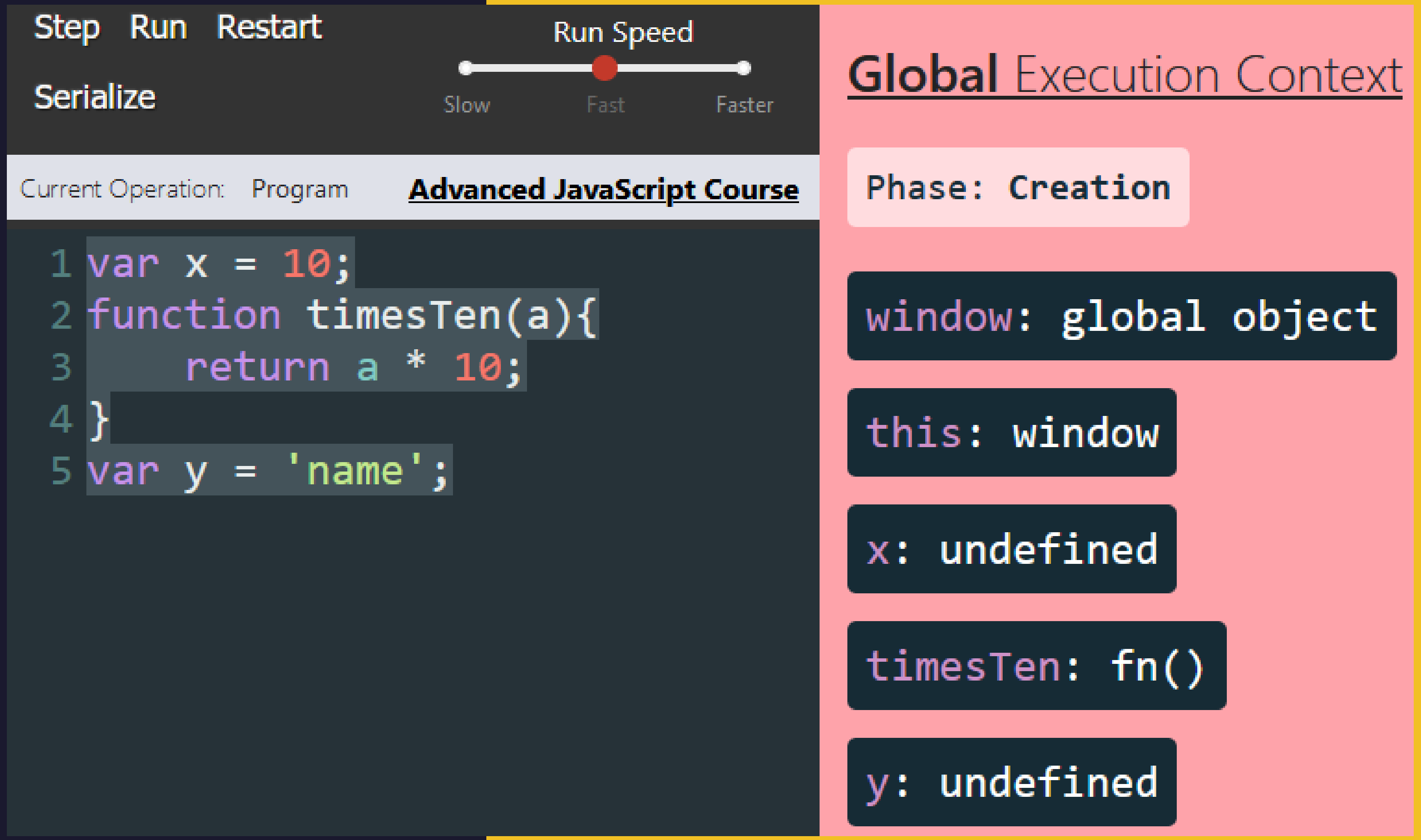
1. **Creation Phase:** A lexical environment is created.
2. Execution Phase: Values are stored to variables.



Creation Phase

The global creation phase which will do the following:

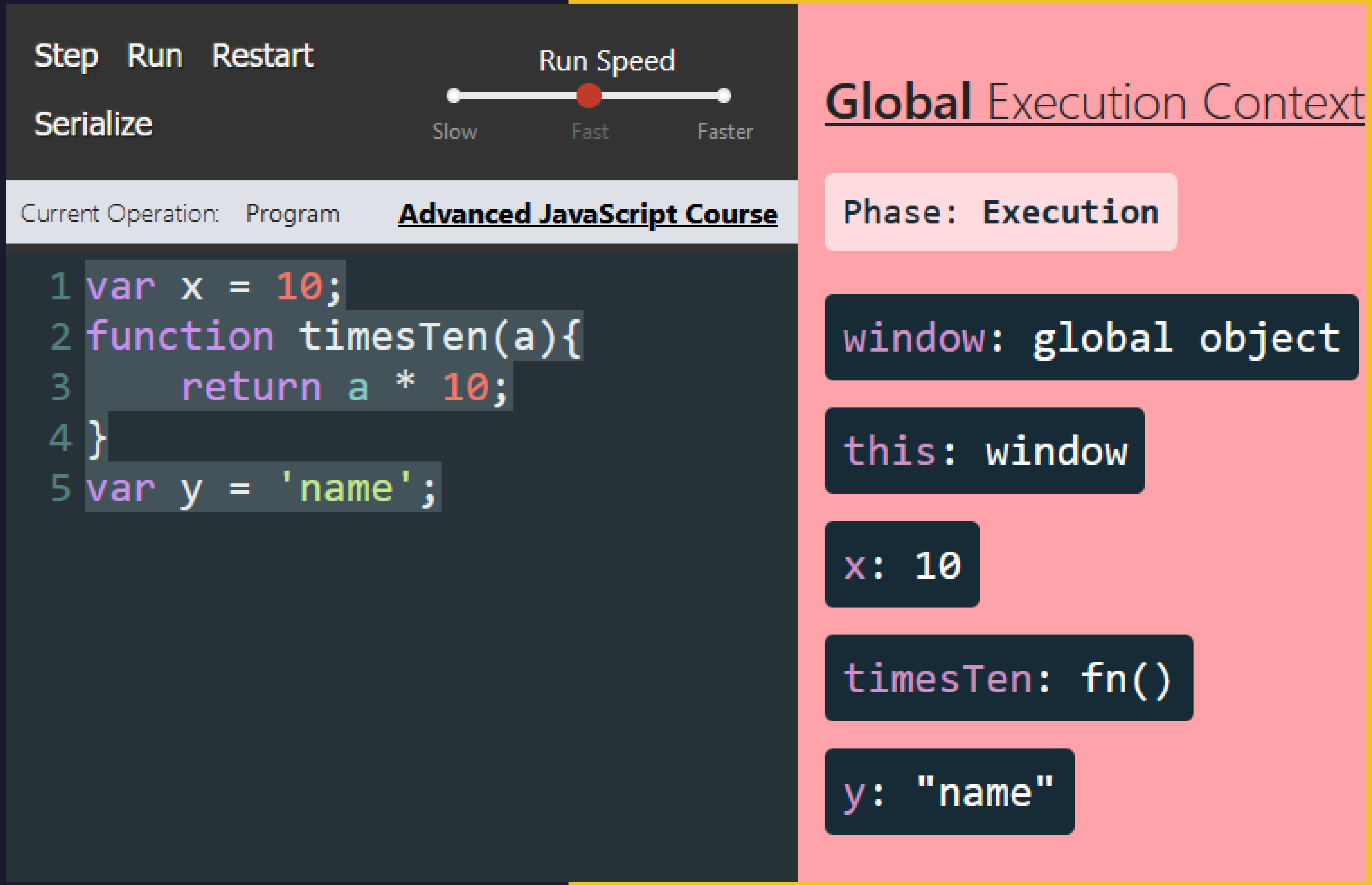
1. Create a *global* object.
2. Create an object called `` `this` ``.
3. Create the lexical environment which is of two types:
    a. Variable Environment: records bindings created by the var (*undefined*).
    b. Lexical Environment: records both variable (let/const) bindings (*<uninitialized>*) and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



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".

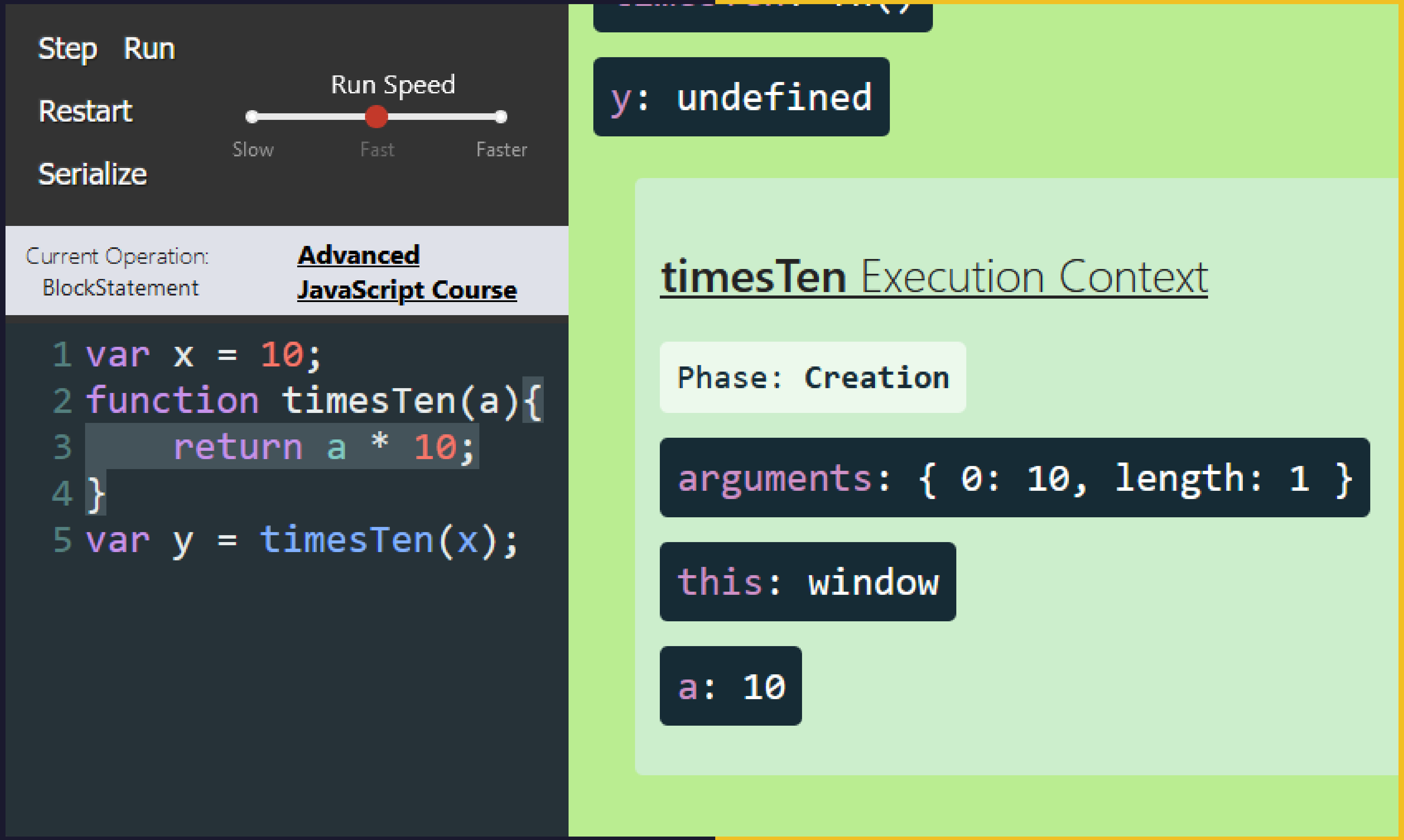


Function Execution Context

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.



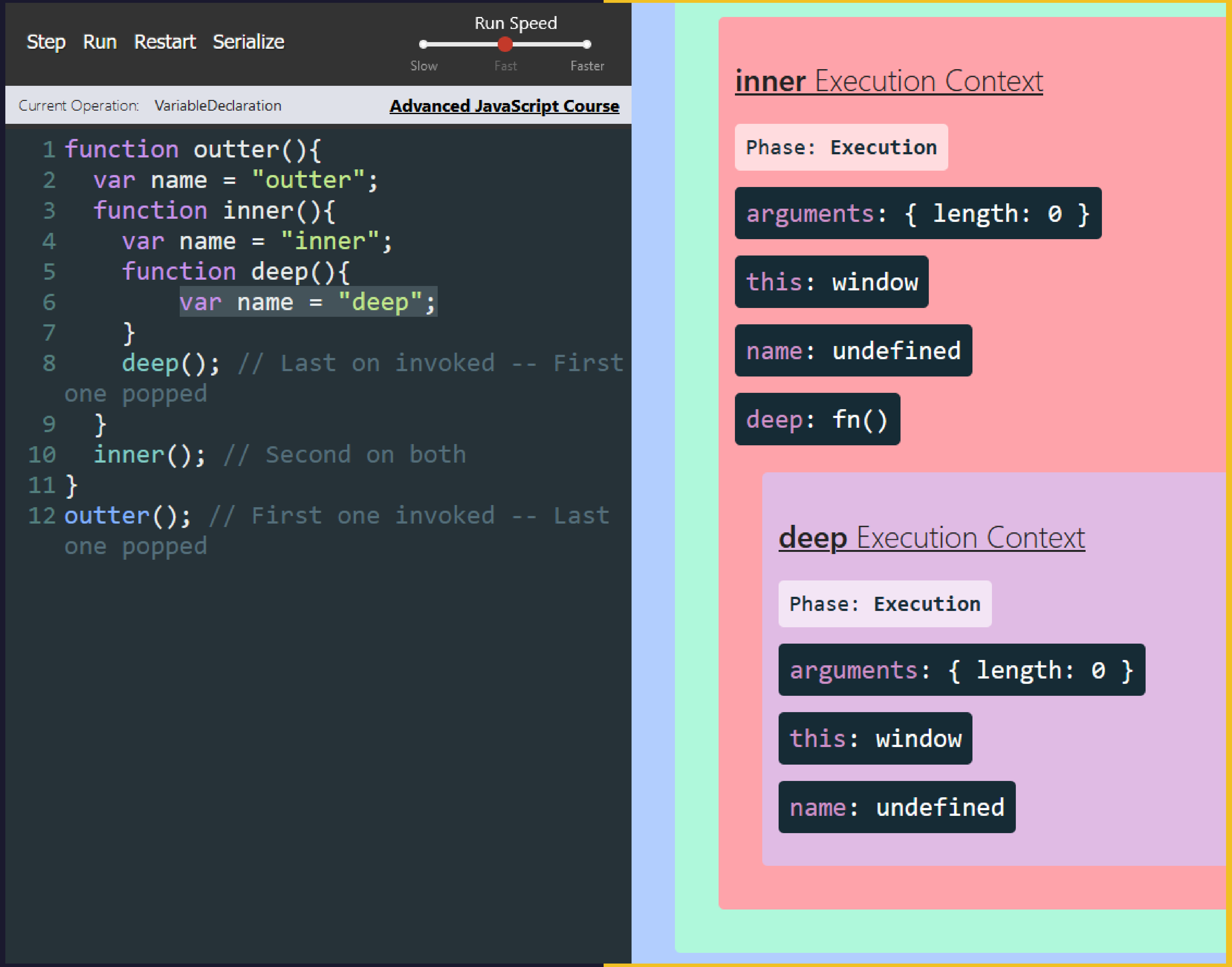
JavaScript engine uses Call Stack to keep track of function calls

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.

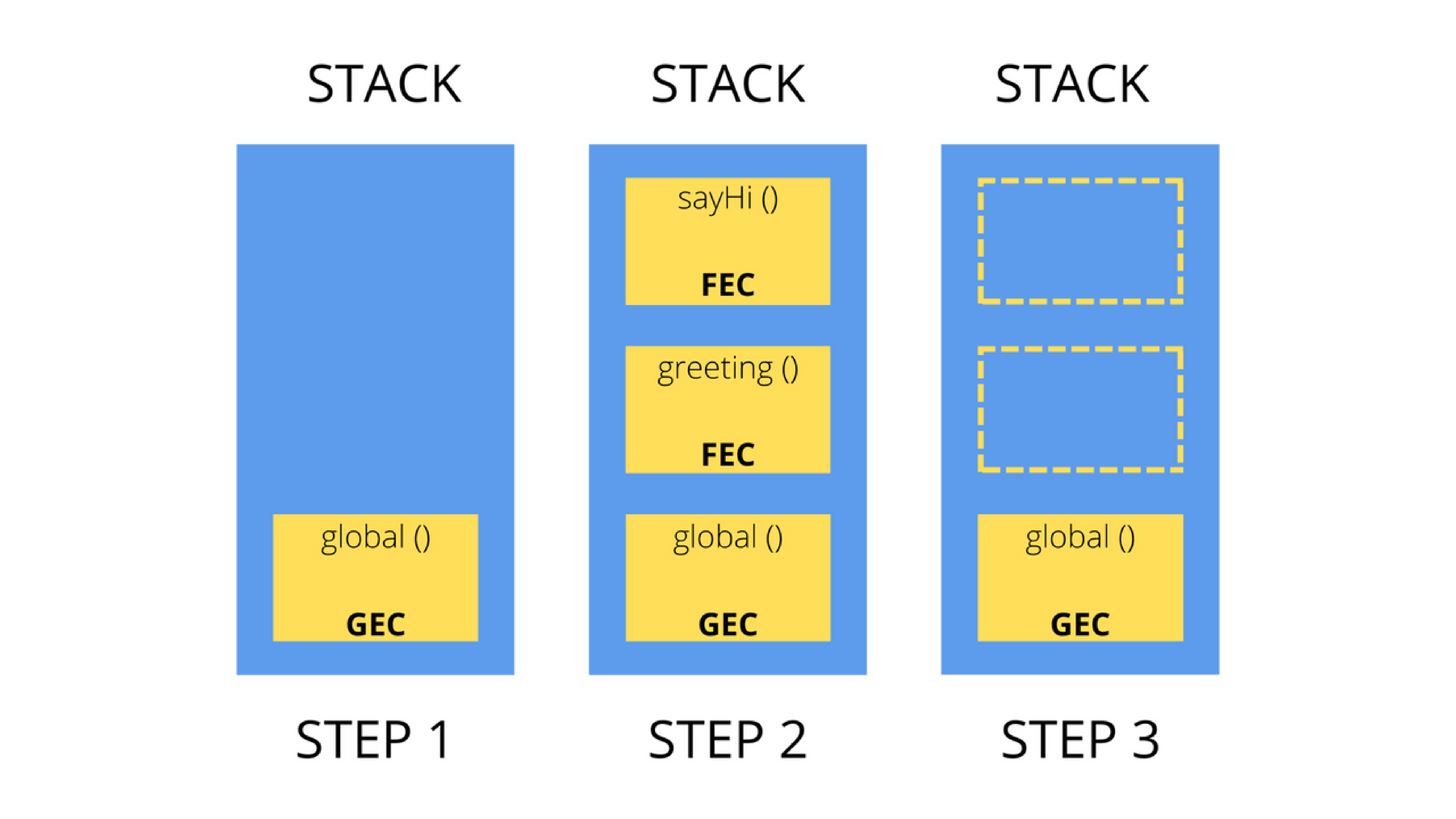
When a program is running, the call stack is constantly changing as functions are called and then returned.

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.



Call Stack

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



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.

```
let scope = "global scope";
function checkscope() {
    let scope = "local scope";
    return scope;
}
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.

```
function outer() {
    let counter = 0;
    function incrementCounter() {
        counter = counter + 1;
        return counter;
    }
    return incrementCounter;
}
const increment = outer();
let count1 = increment();
let count2 = increment();
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.

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

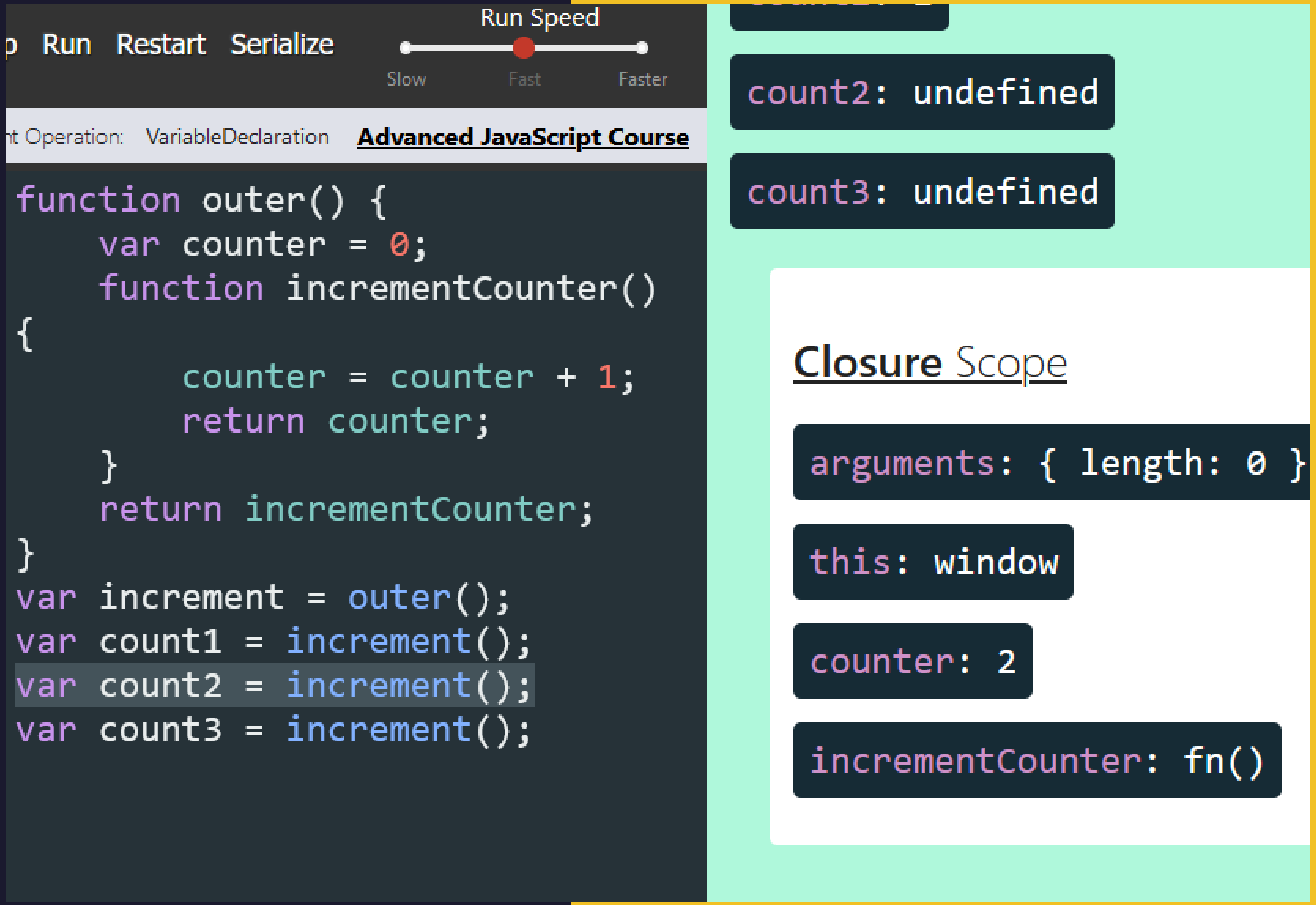
```
function outer() {
    let counter = 0;
    function incrementCounter() {
        counter = counter + 1;
        return counter;
    }
    return incrementCounter;
}
const increment = outer();
let count1 = increment();
let count2 = increment();
let count3 = increment();
console.log(count1, count2, 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 it's local memory, then in its variable environment "Backpack".



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.

```
function outer() {
    let counter = 0;
    function incrementCounter() {
        counter = counter + 1;
        return counter;
    }
    return incrementCounter;
}
const counter1 = outer();// counter = 0
let count1 = counter1(); // counter = 1
const counter2 = outer();// counter = 0
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.

```
function once(func) {
    let called;
    return function (...args) {
        if (!called) {
            called = func(...args)
        }
        return called;
    }
}
function temp(value) {
    //some code here...
}
const runMeOnce = once(temp);
runMeOnce('some value'); // executed
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.

```javascript
function makeCounter() {
    let privateCounter = 0;
    function changeBy(val) {
        privateCounter += val;
    }
    return {
        increment: function () {
            changeBy(1);
        },
        decrement: function () {
            changeBy(-1);
        },
        value: function () {
            return privateCounter;
        }
    }
}
const counter1 = makeCounter();
counter1.increment();
counter1.decrement();
```

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

Session
Organizers

Tamer Naana, Lead

Trainer, Supervisor

Huthaifa Salman, Intern

Author

THANKS
FOR WATCHING