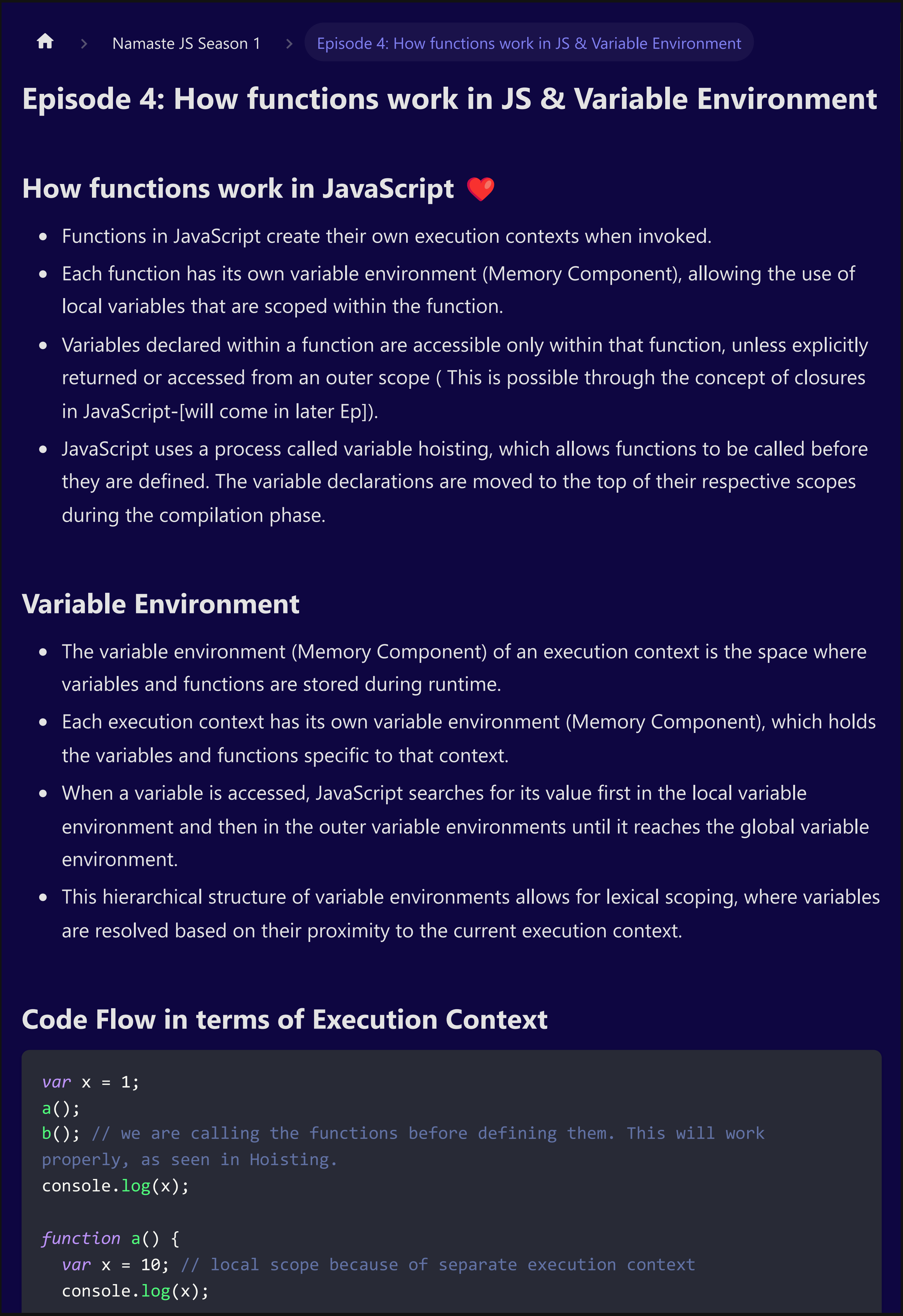


Namaste JS Season 1 › Episode 4: How functions work in JS & Variable Environment

Episode 4: How functions work in JS & Variable Environment

How functions work in JavaScript ❤️

- Functions in JavaScript create their own execution contexts when invoked.
- Each function has its own variable environment (Memory Component), allowing the use of local variables that are scoped within the function.
- Variables declared within a function are accessible only within that function, unless explicitly returned or accessed from an outer scope (This is possible through the concept of closures in JavaScript-[will come in later Ep]).
- JavaScript uses a process called variable hoisting, which allows functions to be called before they are defined. The variable declarations are moved to the top of their respective scopes during the compilation phase.

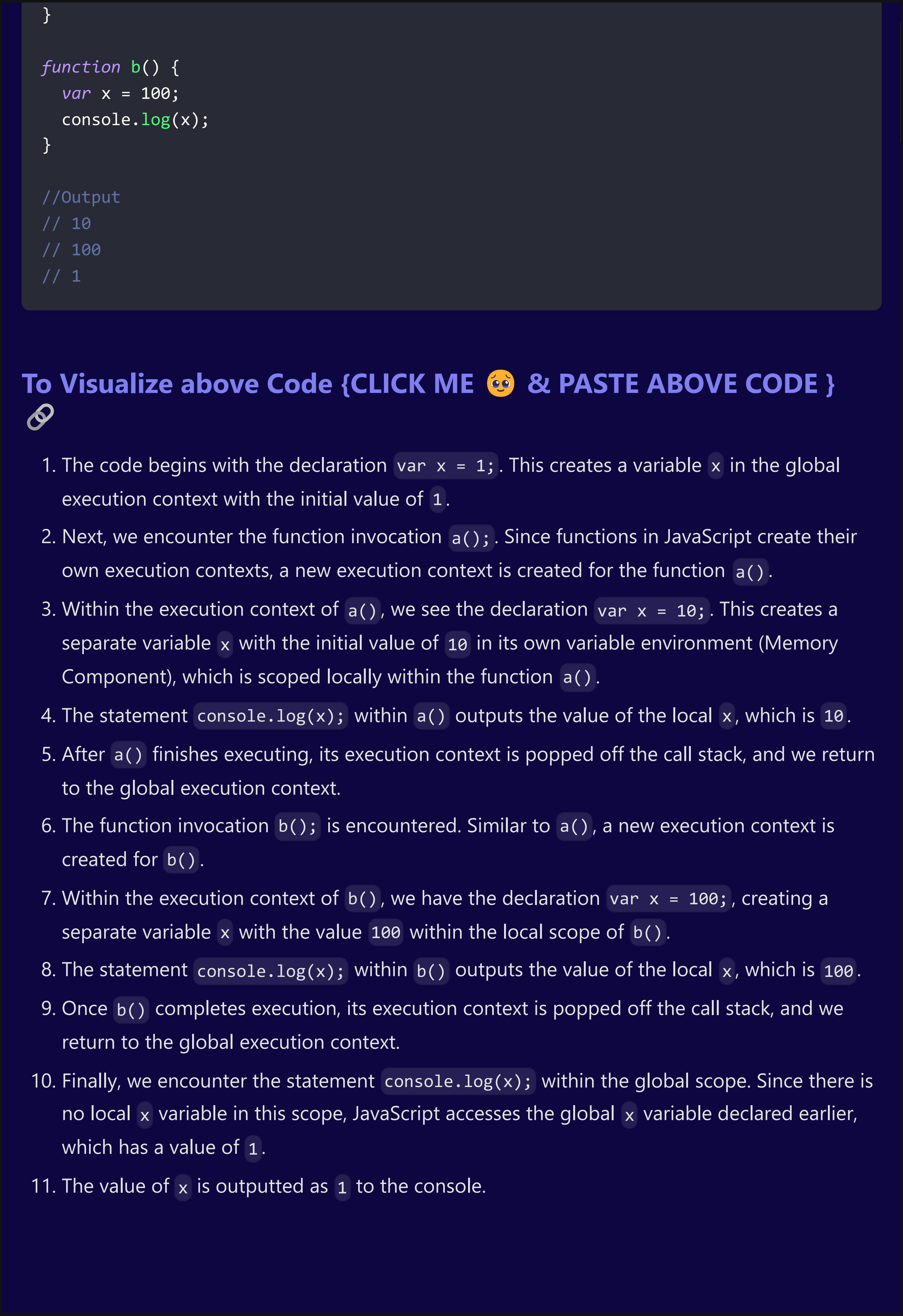
Variable Environment

- The variable environment (Memory Component) of an execution context is the space where variables and functions are stored during runtime.
- Each execution context has its own variable environment (Memory Component), which holds the variables and functions specific to that context.
- When a variable is accessed, JavaScript searches for its value first in the local variable environment and then in the outer variable environments until it reaches the global variable environment.
- This hierarchical structure of variable environments allows for lexical scoping, where variables are resolved based on their proximity to the current execution context.

Code Flow in terms of Execution Context

```
var x = 1;
a();
b(); // we are calling the functions before defining them. This will work properly, as seen in Hoisting.
console.log(x);

function a() {
  var x = 10; // local scope because of separate execution context
  console.log(x);
```



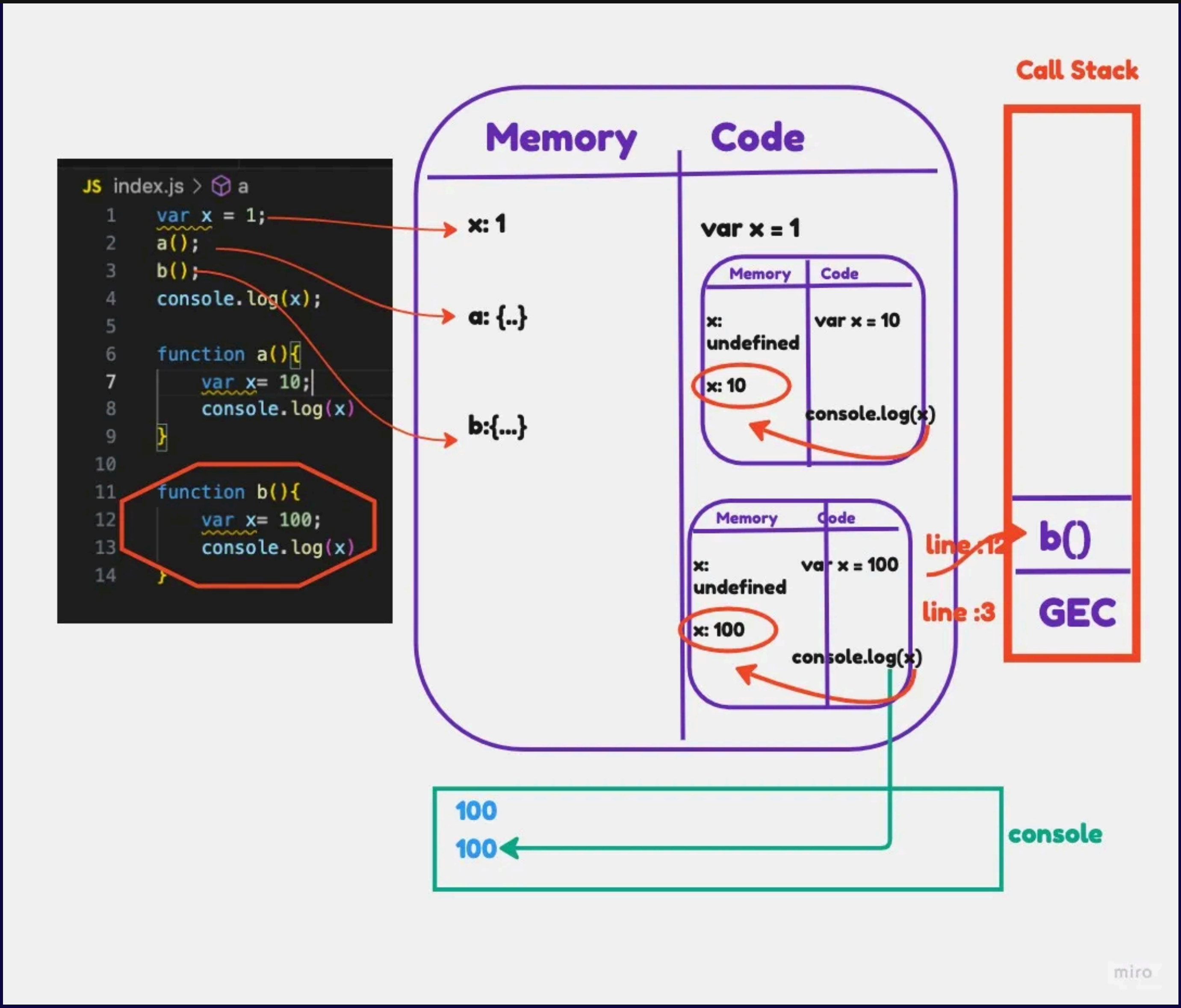
```
}

function b() {
  var x = 100;
  console.log(x);
}

//Output
// 10
// 100
// 1
```

To Visualize above Code {CLICK ME 🥺 & PASTE ABOVE CODE } 🔗

1. The code begins with the declaration `var x = 1;`. This creates a variable `x` in the global execution context with the initial value of `1`.
2. Next, we encounter the function invocation `a();`. Since functions in JavaScript create their own execution contexts, a new execution context is created for the function `a()`.
3. Within the execution context of `a()`, we see the declaration `var x = 10;`. This creates a separate variable `x` with the initial value of `10` in its own variable environment (Memory Component), which is scoped locally within the function `a()`.
4. The statement `console.log(x);` within `a()` outputs the value of the local `x`, which is `10`.
5. After `a()` finishes executing, its execution context is popped off the call stack, and we return to the global execution context.
6. The function invocation `b();` is encountered. Similar to `a()`, a new execution context is created for `b()`.
7. Within the execution context of `b()`, we have the declaration `var x = 100;`, creating a separate variable `x` with the value `100` within the local scope of `b()`.
8. The statement `console.log(x);` within `b()` outputs the value of the local `x`, which is `100`.
9. Once `b()` completes execution, its execution context is popped off the call stack, and we return to the global execution context.
10. Finally, we encounter the statement `console.log(x);` within the global scope. Since there is no local `x` variable in this scope, JavaScript accesses the global `x` variable declared earlier, which has a value of `1`.
11. The value of `x` is outputted as `1` to the console.



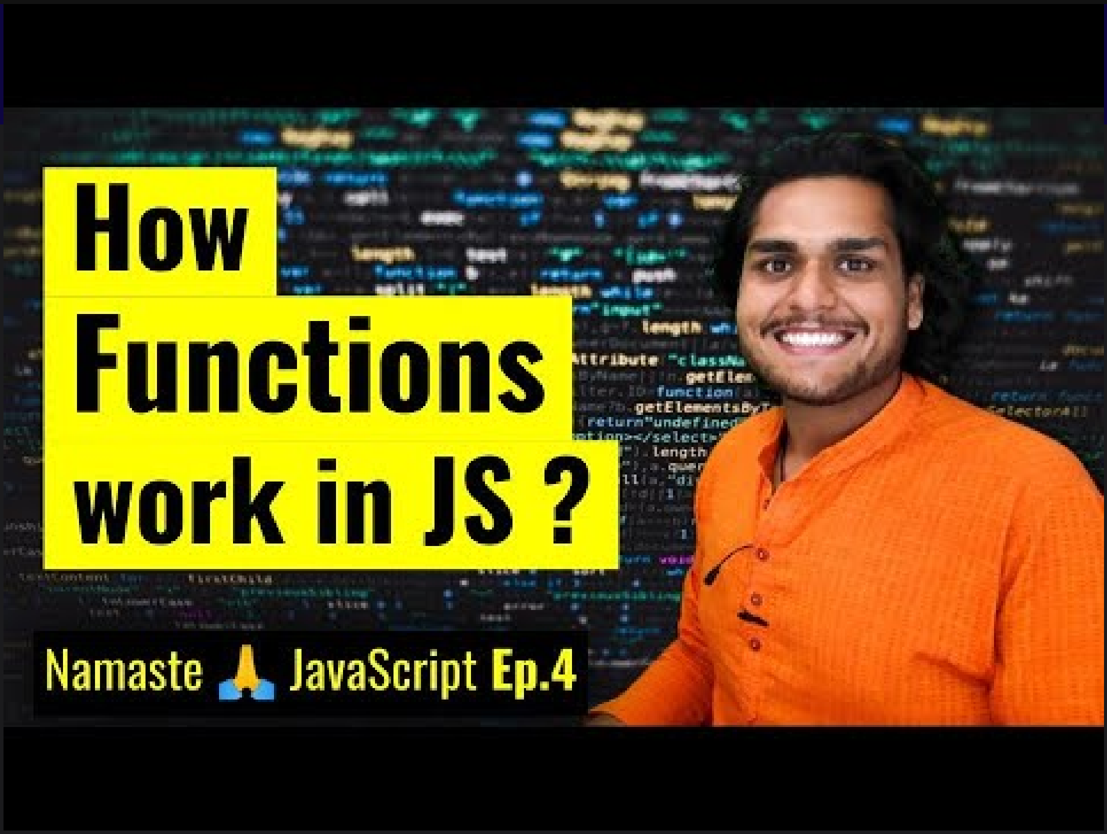
JS index.js > a

```
var x = 1;
a();
b();
console.log(x);

function a(){
    var x= 10;
    console.log(x)
}

function b(){
    var x= 100;
    console.log(x)
}
```

Watch Live On Youtube below:



✎ Edit this page