CLOSURES

Actions Supported by All the Laws of Nature

Wholeness: A fundamental aspect of function-oriented programming in JavaScript is the use of closures to store state information associated with a function when the function is passed to other objects. Science of Consciousness: Closures provide a protective wrapper for state information associated with a function. An analogy in consciousness is the supportive wrapper that transcendental consciousness provides to our own consciousness. At this level of consciousness we are connected to the home of all the laws of nature.

Main Points

1. Closure
2. Global and Function objects

Main Point Preview: Closures

Closures are created whenever an inner function with free variables is returned or assigned as a callback. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Closures provide a supportive wrapper for actions that will occur in another context. Transcendental consciousness provides a supportive wrapper for our actions that will occur outside of meditation.

Closure

- JavaScript is function-oriented language.
  - A function can be created dynamically,
  - copied to another variable or
  - passed as an argument to another function and called from a totally different place later.
- a function can access variables outside of it, this feature is used quite often.
  - what happens when an outer variable changes?
    - Does a function get the most recent value or the one that existed when the function was created?
- what happens when a function travels to another place in the code and is called from there
  - does it get access to the outer variables of the new place?

Callbacks with reference to outer variable?

- Common use case: function is scheduled to execute later
  - E.g., after a user action or a network request

```
let name = "John";
function sayHi() {
  alert("Hi, " + name);
}
name = "Pete";
sayHi(); // what will it show: "John" or "Pete"?

function makeWorker() {
  let name = “John";
  return function() {
    alert(name);
  };
}
let name = “Pete";
// create a function
let work = makeWorker();
// call it
work(); // what will it show? “John" (name where created) or “Pete" (name where called)?
```

exercise

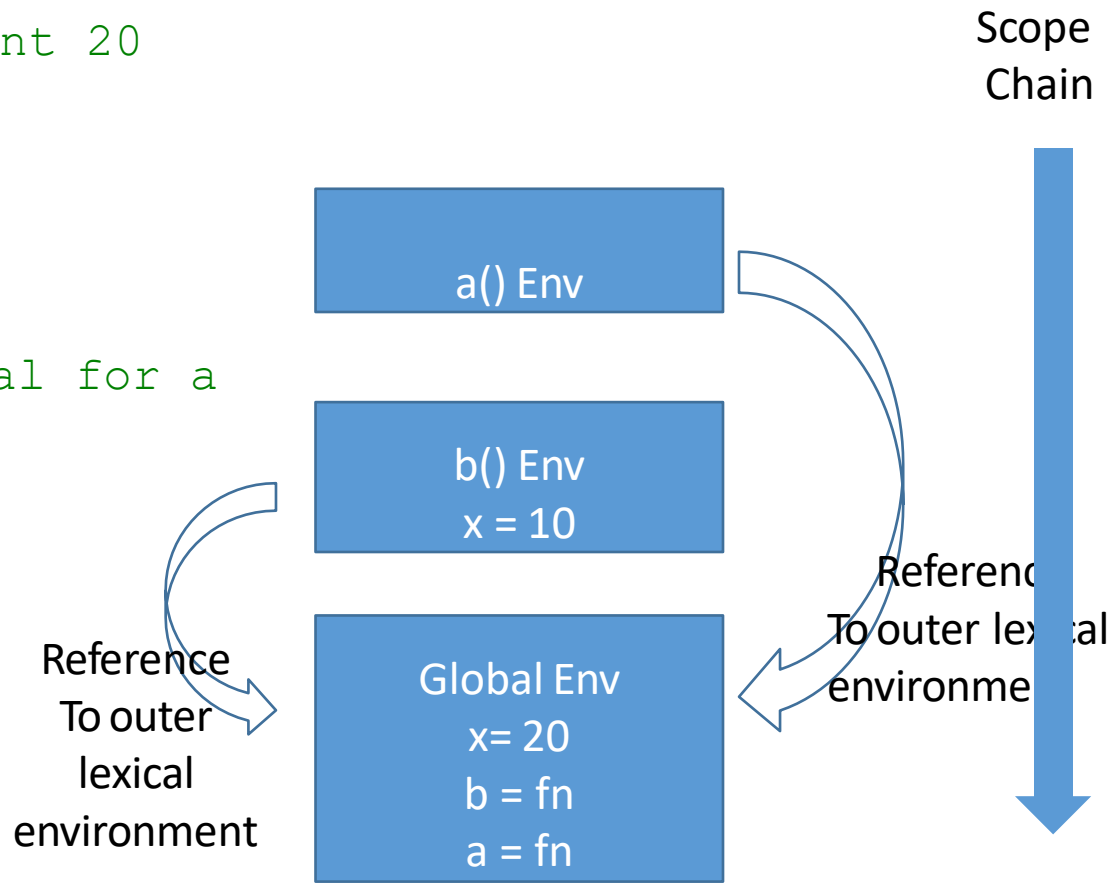
Write a function, makeCounter() that declares a local variable, count, and another local variable, innerFunc, which is an inner function; innerFunc will increment the count variable and return the incremented value. makeCounter should return innerFunc.

Call it twice to make different counters, counter1 and counter2.

Do they keep independent counts?

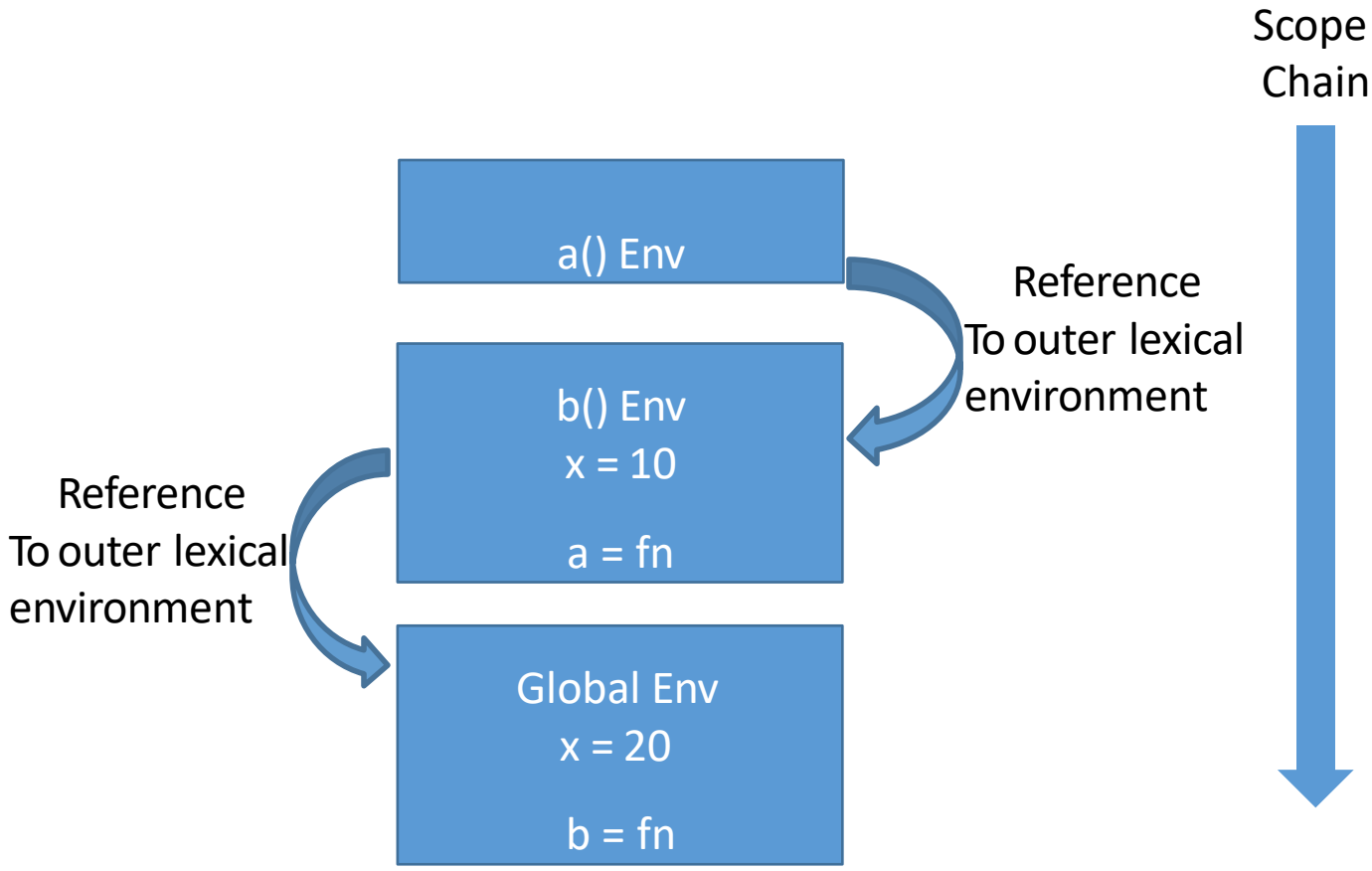
Scope Example1 (review)

```
function a() {
 console.log(x); // consult
 Global for x and print 20
 from Global
}
function b() {
 const x = 10;
 a(); // consult Global for a
}
const x = 20;
b();
```



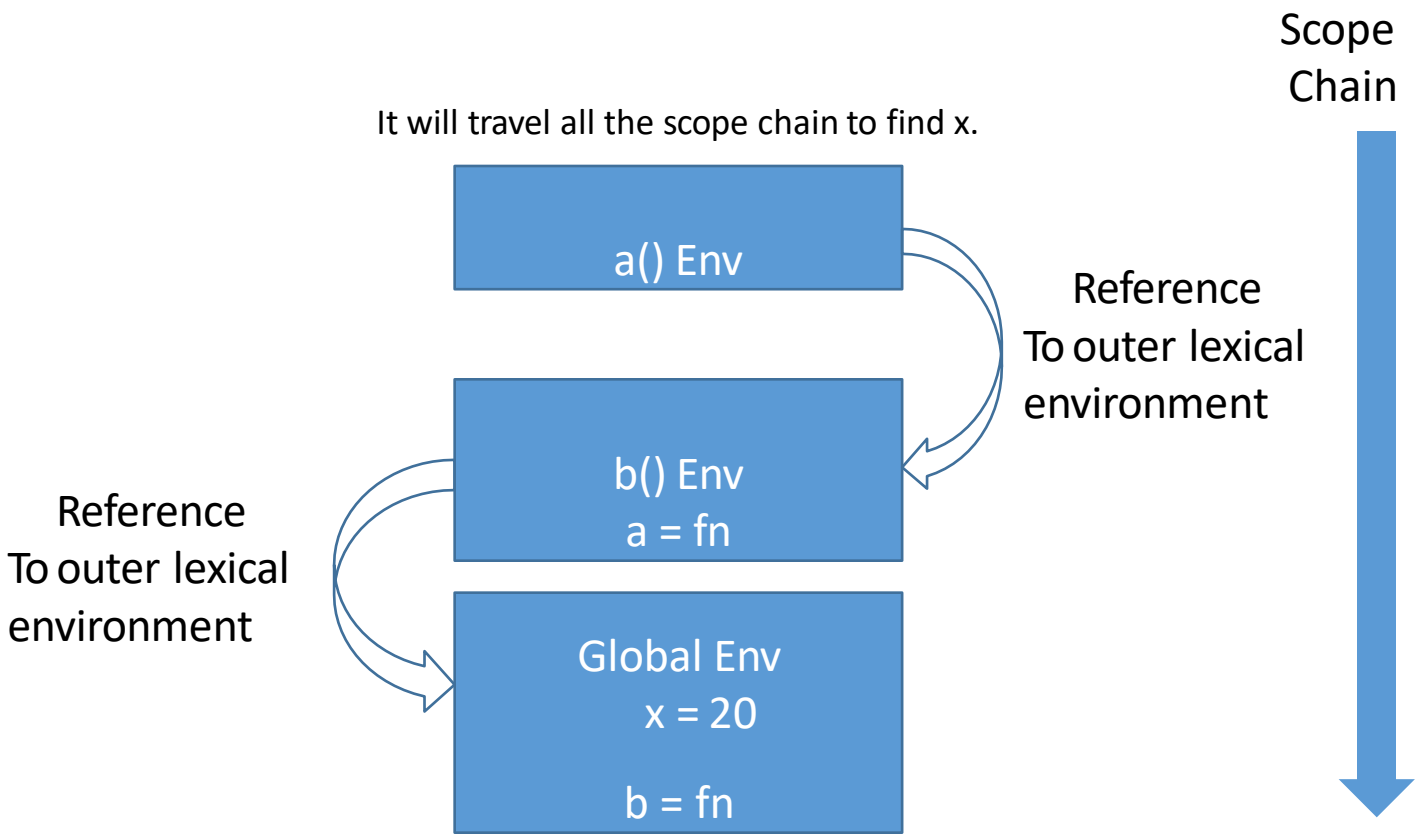
Scope Example 2: Inner function (review)

```
function b() {
 function a() {
   console.log(x);
 }
 const x = 10;
 a();
}
const x = 20;
b();
```



Scope Example 3 (review)

```
function b() {
 function a() {
    console.log(x);
 }
 a();
}
const x = 20;
b();
```



Exercise

```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}
let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Draw the execution context diagram for this code

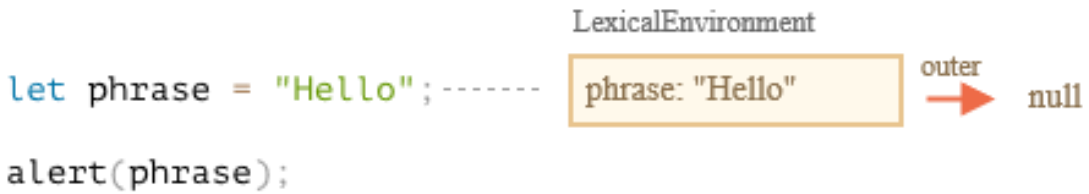
Important: execution context cannot be discarded if it contains an external variable for a returned function

Lexical environment

- How variables work in the compiler

- every running function, code block {...}, and script have internal object
  - Lexical Environment., which has two parts
    - Environment Record –stores all local variables as its properties (and information like value of this).
    - reference to the outer lexical environment

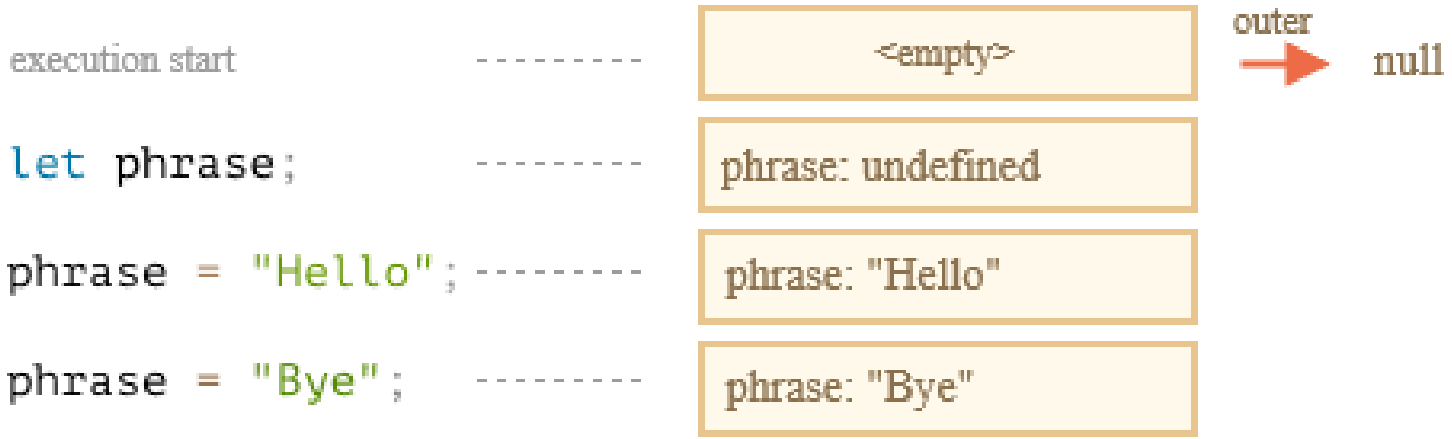
- A “variable” is a property of Environment Record
  - To get or change a variable” means “to get or change a property of that object”.



- rectangle shows Environment Record (variable store)
  - arrow means the outer reference.
  - global Lexical Environment has no outer reference, so it points to null

Lexical environment 2

- Rectangles demonstrate how global Lexical Environment changes :
  - When script starts, the Lexical Environment is empty
  - The let phrase definition appears. It has been assigned no value, so undefined is stored.
  - phrase is assigned a value.
  - phrase changes value.

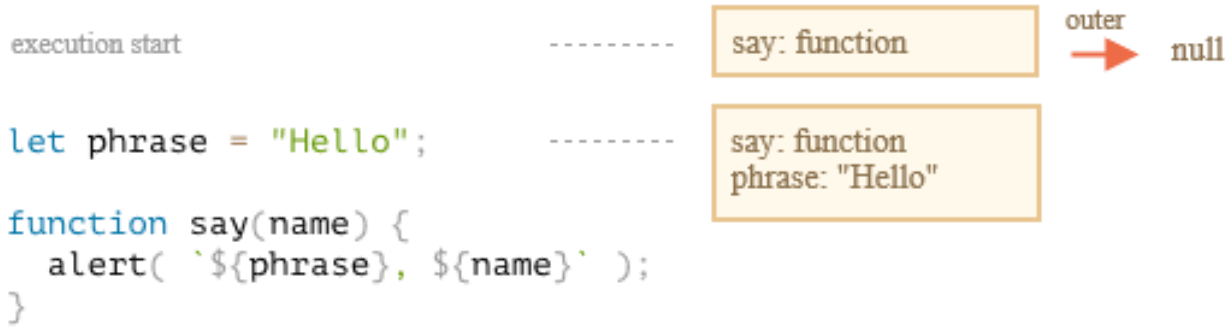


Global lexical environment

- To summarize:
  - A variable is a property of a special internal object,
    - associated with the currently executing block/function/script. (execution context stack)
  - Working with variables is working with the properties of that object

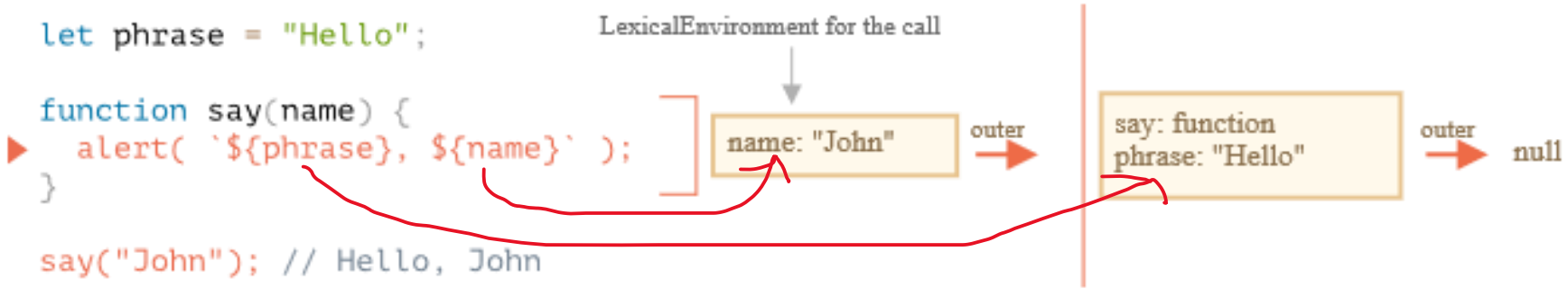
- Function Declaration
  - fully initialized when a Lexical Environment is created.
  - For top-level functions, it is the moment when the script is started.
  - why we can call a function declaration before it is defined.

  - Lexical Environment is non-empty from the beginning.
    - It has say function and phrase, because they are declarations
    - later it gets value assigned to phrase
    - “2 pass compiler”



Inner and outer Lexical Environment

- what happens when access outer variable.
  - During the call, say() uses the outer variable phrase
  - new Lexical Environment is created automatically to store local variables and parameters of the call.



- outer Lexical Environment is the global Lexical Environment.
- inner Lexical Environment has a reference to the outer one.
  - When the code wants to access a variable
    - the inner Lexical Environment is searched first,
    - then the outer one, then the more outer one and so on until the global one.

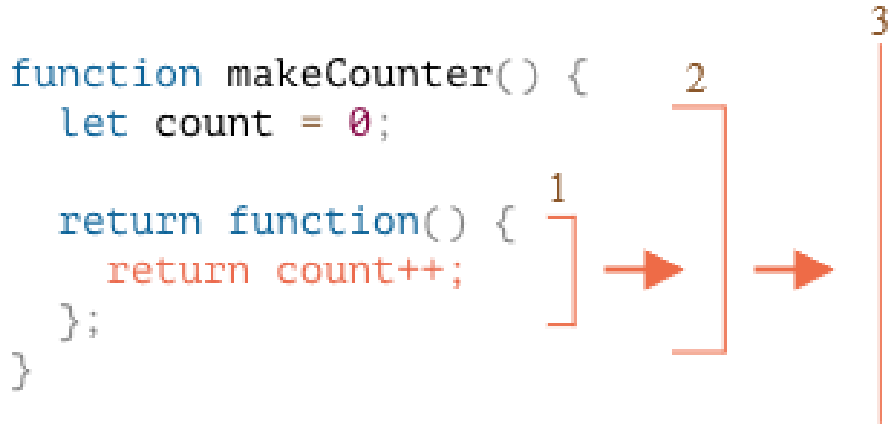
- A function gets outer variables as they are now, it uses the most recent values.
  - Old variable values are not saved anywhere

Nested functions

- A function is called “nested” (inner) when it is created inside another function
- Nested functions are quite common in JavaScript.
- What’s much more interesting, a nested function can be returned:
  - as a property of a new object
    - if the outer function creates an object with methods
  - as a result by itself.
    - can then be used somewhere else.
    - No matter where, it still has access to the same outer variables

Nested functions 2

- When inner function runs,
  - variable in count++ is searched from inside out.
    1. The locals of the nested function…
    2. The variables of the outer function…
    3. And so on until it reaches global variables.

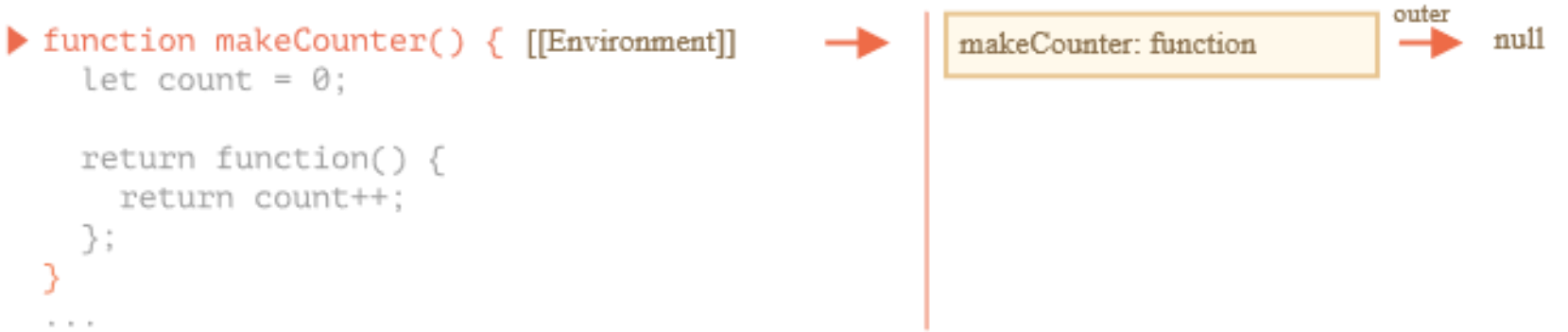


- two questions:
  - Can we somehow reset the counter count from the code that doesn't belong to makeCounter? E.g. after console.log calls
  - If we call makeCounter() multiple times
    - returns multiple counter functions.
    - independent or share the same count?

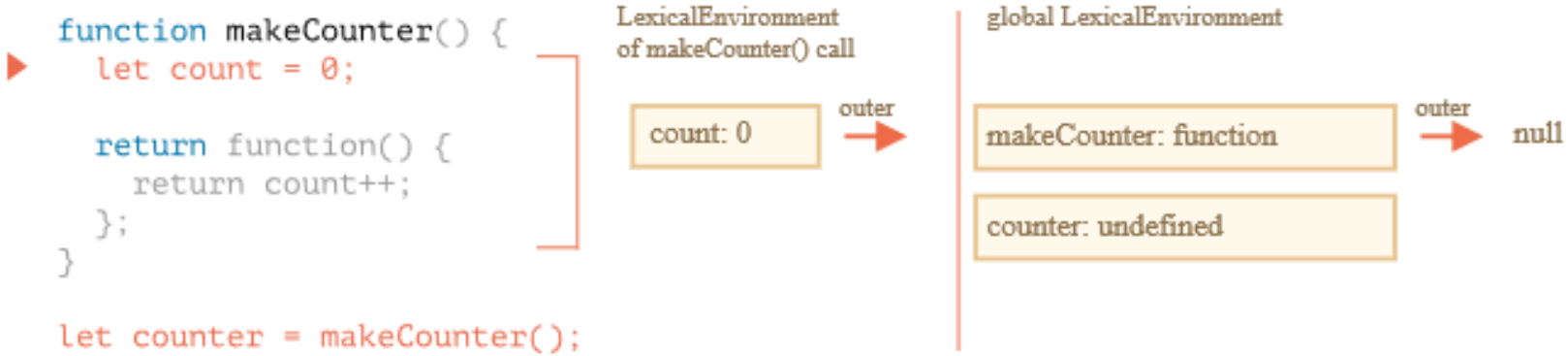
```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}
let counter = makeCounter();
console.log( counter() ); // 0
console.log( counter() ); // 1
console.log( counter() ); // 2
```

Environments in detail

- All functions receive a hidden property [[Environment]]
  - reference to Lexical Environment of their creation -- "outer pointer"
  - how function knows where it was made.
- Here, makeCounter is created in global Lexical Environment
  - makeCounter.[[Environment]] is hidden function property that has reference to global Lexical Environment

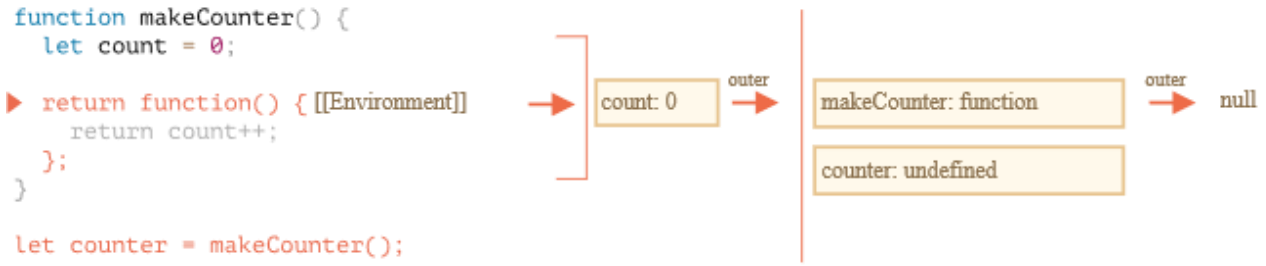


- snapshot of the moment when execution is on first line inside makeCounter():
  - counter.[[Environment]] has reference to {counter: 0 } Lexical Environment of makeCounter

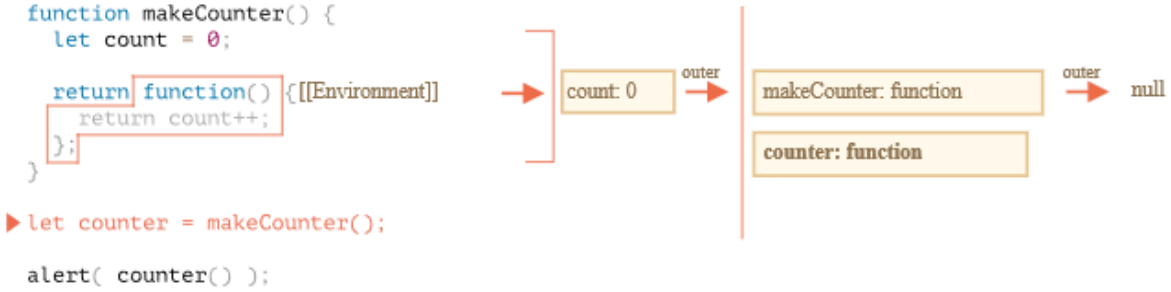


Environments in detail 2

- During the execution of makeCounter(), a tiny nested function is created.
  - doesn't matter whether the function is created using Function Declaration or Function Expression.
  - All functions get the [[Environment]] property that references the Lexical Environment in which they were made.
  - For our new nested function the value of [[Environment]] is the current Lexical Environment of makeCounter() (where it was born):
  - on step the inner function was created, it is not yet called.

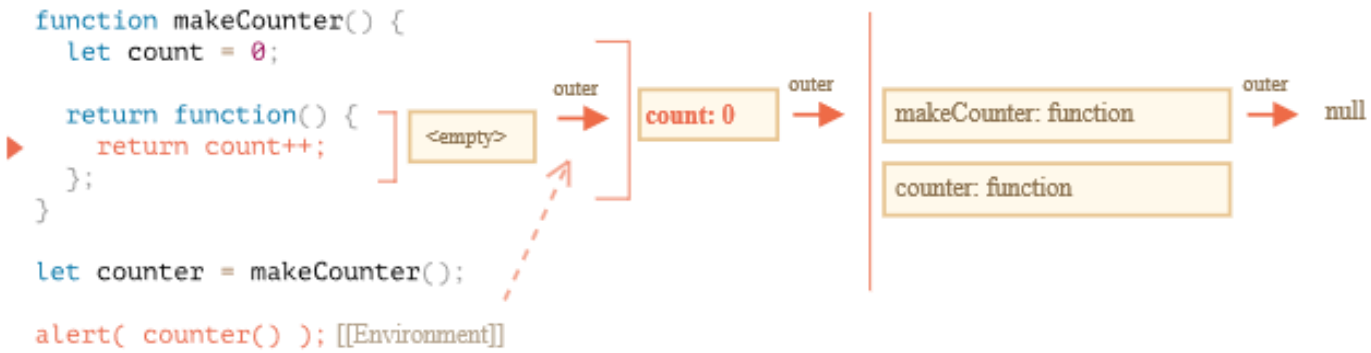


- As the execution goes on, the call to makeCounter() finishes,
  - result (the tiny nested function) is assigned to the global variable counter:
  - That function has only one line: return count++, that will be executed when we run it.



Environments in detail 3

- When counter() is called, a new Lexical Environment is created for the call.
  - empty, as counter has no local variables by itself.
  - [[Environment]] of counter is used as the outer reference for it,
    - provides access to the variables of the former makeCounter() call where it was created
- when the call looks for count variable,
  - first searches its own Lexical Environment (empty),
  - then the Lexical Environment of the outer makeCounter() call, where finds it.

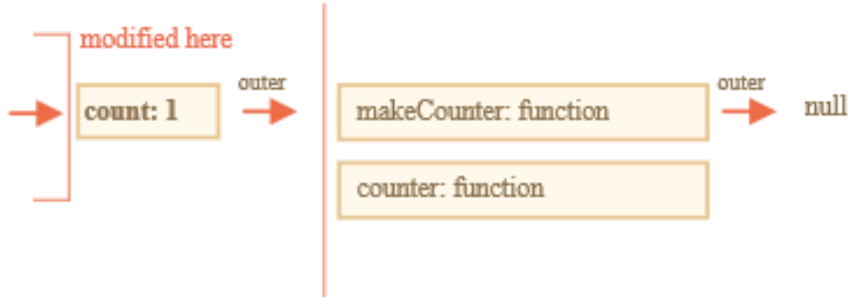


- memory management
  - makeCounter() call finished some time ago,
  - its Lexical Environment was retained in memory,
    - because there's a nested function with [[Environment]] referencing it.
  - a Lexical Environment object lives as long as there is a function which may use it.
  - only when there are none remaining, it is cleared

Environments in detail 4

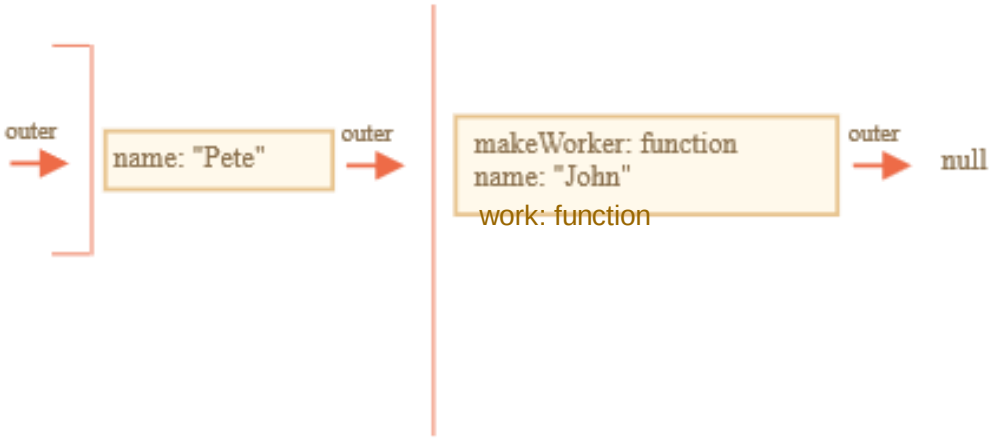
- The call to counter() returns the value of count, and then increases it (postfix)
  - count is modified exactly in the environment where it was found.
  - The next counter() invocation does the same.

```
function makeCounter() {
  let count = 0;

  return function() { [[Environment]]
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() );
```



- answer to second question from the beginning of the chapter??

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";
let work = makeWorker();

work(); // Pete
```



Exercise

```
let name = "John";
function sayHi() {
  alert("Hi, " + name);
}
name = "Pete";
sayHi(); // what will it show: "John" or "Pete"?
```

Draw the execution context diagram for this code

Exercise

```
function makeWorker() {
  let name = “John";
  return function() {
    alert(name);
  };
}
let name = “Pete";
// create a function
let work = makeWorker();
// call it
work(); // what will it show? “John" (name where created) or “Pete" (name where called)?
```

Draw the execution context diagram for this code

What is a Closure?

- (def) A **closure** is the combination of
  - function bundled together (enclosed) with
  - references to its surrounding state (the **lexical environment**).

- a closure gives you access to an outer function's scope from an inner function
  - closures are created every time a function is created

- Inner functions are created (declared) when the outer function is called
  - Whenever a function is called a new execution context is created and added to the call stack
  - Every execution context has a lexical environment associated with it that tracks the variable bindings and values

Closures

- general programming term “closure”, that developers should know

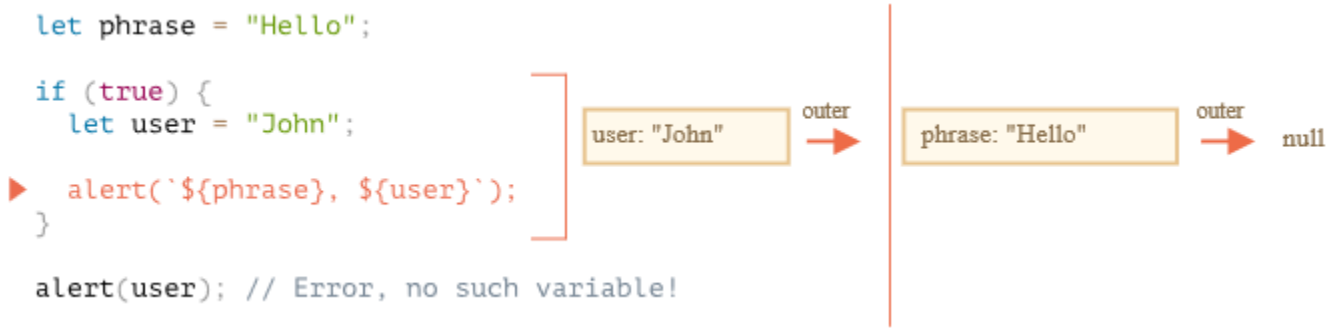
- Closure: function that remembers its outer variables and can access them
  - Not possible in all languages
  - in JavaScript, all functions are naturally closures
    - automatically remember where they were created
    - using hidden [[Environment]] property

- Common front end job interview question “what’s a closure?”,
  - a valid answer would be the definition
  - explanation that all functions in JavaScript are closures,
  - few more words about : the [[Environment]] property and how Lexical Environments work
  - Some define closures to be (only) when there is an inner function with free outer variable
    - “free” variables (not defined in the local function)
    - Implies an inner function

- To use a closure, define a function inside another function and expose it.
  - To expose a function, return it or pass it to another function.
  - inner function will have access to variables in the outer function scope,
    - even after outer function has returned.

Code blocks and scope

- a Lexical Environment exists for any code block {...}
  - created when a code block runs and contains block-local variables.



- When execution gets to the if block,
  - new “if-only” Lexical Environment is created for it
  - has the reference to the outer one, so phrase can be found.
  - all variables and Function Expressions declared inside it reside in that Lexical Environment
    - can’t be seen from the outside
    - after if finishes, the alert below won’t see the user, hence the error.

For, while

- For a loop, every iteration has a separate Lexical Environment.
  - If a variable is declared in for(let ...), then it's also in there:

```
for (let i = 0; i < 10; i++) {
  // Each loop has its own Lexical Environment
  // {i: value}
}
alert(i); // Error, no such variable
```

- for let i is visually outside of {...}.
  - The for and while constructs are special
  - each iteration of the loop has its own Lexical Environment with the current i in it.
- like if, after the loop i is not visible.

Global object

- window in browsers
  - global in Node
- Window contains all the global functions
  - alert
  - prompt
  - setInterval
  - setTimer
  - console.log
  - Array
  - String
  - screenX, screenY, …
  - And hundreds of other global properties and methods
- Can view in the browser console
- Every global variable declaration or function declaration gets added to the global object
  - Bad practice to do this
  - Node fixes with module system—every file is a module

Bare code blocks

- can use “bare” code block {…} to isolate variables into a “local scope”.
  - in web browser all scripts share the same global area.
  - create a global variable in one script, it becomes available to others.
  - source of conflicts if two scripts use the same variable name
    - Last one loaded overwrites and becomes source of values for early script too
    - if the variable name is a widespread word, e.g., lat, long

- to avoid that, can use a code block to isolate the whole script or a part of it:

```
{
  // do some job with local variables that should not be seen outside
  let message = "Hello";
  alert(message); // Hello
}
alert(message); // Error: message is not defined
```

- code outside block doesn’t see variables inside the block
  - Every block has own Lexical Environment.

IIFE

- Before ES6 no block-level lexical environment in JavaScript.
  - Only function scope
- “immediately-invoked function expressions” (abbreviated as IIFE).
  - Special syntax to wrap code and protect global namespace
  - declare a Function Expression and run it immediately
  - nowadays there’s no reason to write such code

```
(function() {
  let message = "Hello";
  alert(message); // Hello
})();
```

Homework I

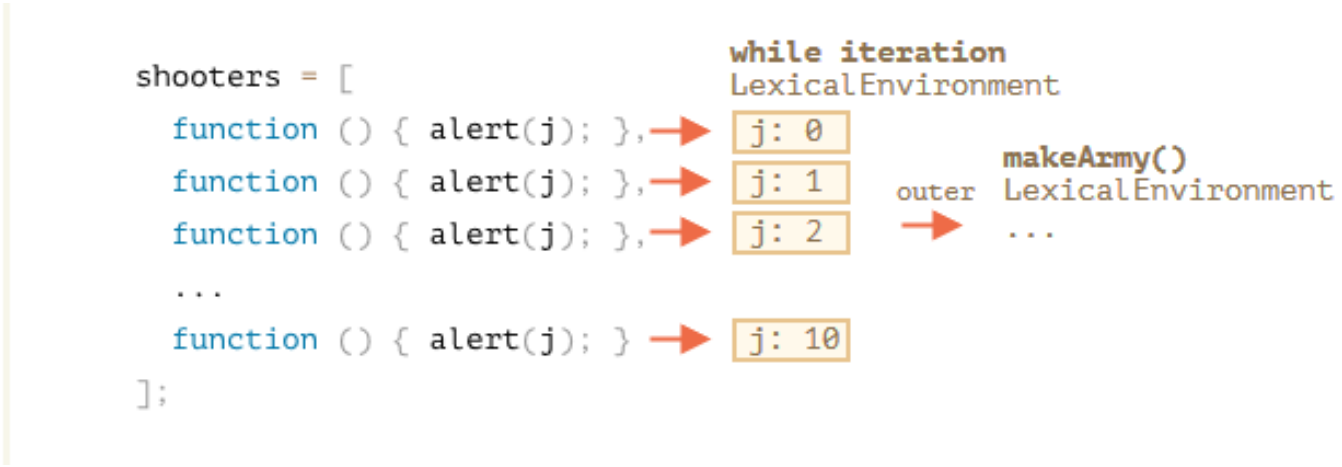
- Does a function pickup latest changes?
- Which variables are available?
- Are counters independent?
- Counter object
- Function in if
- Sum with closures
- Is variable visible?

Homework II

- Filter through function
  - hint: filter's argument must be a function that returns t/f on elements
  - where is the closure?

- Sort by field
  - hint: sort's argument must be a function that takes 2 arguments and returns 1 if arg1 > arg2 or -1
  - where is the closure?

- Army of functions
  - hint below: note that they all share the same j



Homework exercise

- Draw a lexical environment diagram for the following code.
- Follow model of previous page, show:
  - global lexical environment (LE)
  - LE for makeArmy()
  - LE for LE of the while loop
  - LE for army[0]
  - What will army[0] alert?
  - Can you fix the code?
  - How will the diagram change?

```
function makeArmy() {
  let shooters = [];
  let i = 0;
  while (i < 10) {
        let shooter = function() {
          alert( i );
        };
        shooters.push(shooter);
        i++;
  }
  return shooters;
}
let army = makeArmy();
army[0]();
```

The old “var”

- In the very first chapter about variables, we mentioned three ways of variable declaration:
  1. let
  2. const
  3. var
- let and const behave exactly the same way in terms of Lexical Environments.
  - var is a very different beast,
  - originates from very old times.
  - generally not used in modern scripts, but still lurks in the old ones.
- If you don’t plan on meeting such scripts you may even skip this or postpone it
  - But is used in millions of programs – anything prior to ES6

- function scope
  - Ignores blocks
  - Are always ‘hoisted’, which means they are visible throughout the function
    - Even if defined at the end! (like a function declaration)
    - Only the declaration is hoisted, not the assignment
    - Undefined until assignment reached

Main Point: Closures

Closures are created whenever an inner function with free variables is returned or assigned as a callback. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Closures provide a supportive wrapper for actions that will occur in another context. Transcendental consciousness provides a supportive wrapper for our actions that will occur outside of meditation.

Main Point Preview: Function Objects

Functions are objects and can have their own properties. Function properties can store state information associated with a function like a closure, however, function properties are accessible to external objects unlike closure variables.

Function object

- In JavaScript, functions are objects.
- consider functions as callable “action objects”.
  - can not only call them, but also treat them as objects:
  - add/remove properties, pass by reference etc

- “name” property

```
function sayHi() {
  alert("Hi");
}
alert(sayHi.name); // sayHi
```

- “length” that returns the number of function parameters, for instance:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}
alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2  -- rest parameters are not counted
```

Function object 2

- We can also add properties of our own.
- Here we add the counter property to track the total calls count:

```
function sayHi() {
  alert("Hi");
  // let's count how many times we run
  sayHi.counter++;   //property created here
}
sayHi.counter = 0; // initial value
sayHi(); // Hi
sayHi(); // Hi
alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

- A property is not a variable
  - A property assigned to a function like sayHi.counter = 0 does not define a local variable counter inside it.
  - a property counter and a variable let counter are two unrelated things.
  - We can treat a function as an object, store properties in it, but that has no effect on its execution.
    - Variables are not function properties and vice versa
    - parallel worlds.

Function properties can replace closures sometimes

- rewrite the counter function example to use a function property:

```
    function makeCounter() {
  // instead of:
  // let count = 0
  function counter() {
          counter.count += 1;
    return counter.count;  //instead of return count
  };
  counter.count = 0;
  return counter;
}
let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

- better or worse than using a closure?
- Function properties and closures both save state information with a function
  - Closure hides the state info from external functions, local to the closure
  - Function properties publicly accessible
    - E.g., counter.count = 500000;
  - Choice depends on application requirements

Main Point: Function Objects

Functions are objects and can have their own properties. Function properties can store state information associated with a function like a closure, however, function properties are accessible to external objects unlike closure variables.

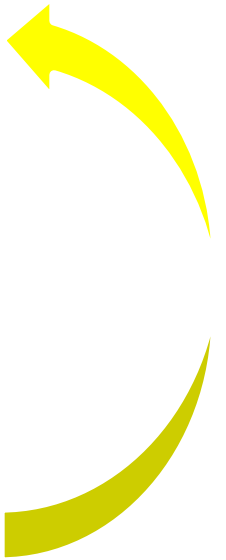
CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Actions Supported by All the Laws of Nature

1. Closures are a feature of functional programming languages that allow state information to be encapsulated with functions when they are passed among objects.
2. The scope of variables in modern JavaScript is defined by the lexical environment.

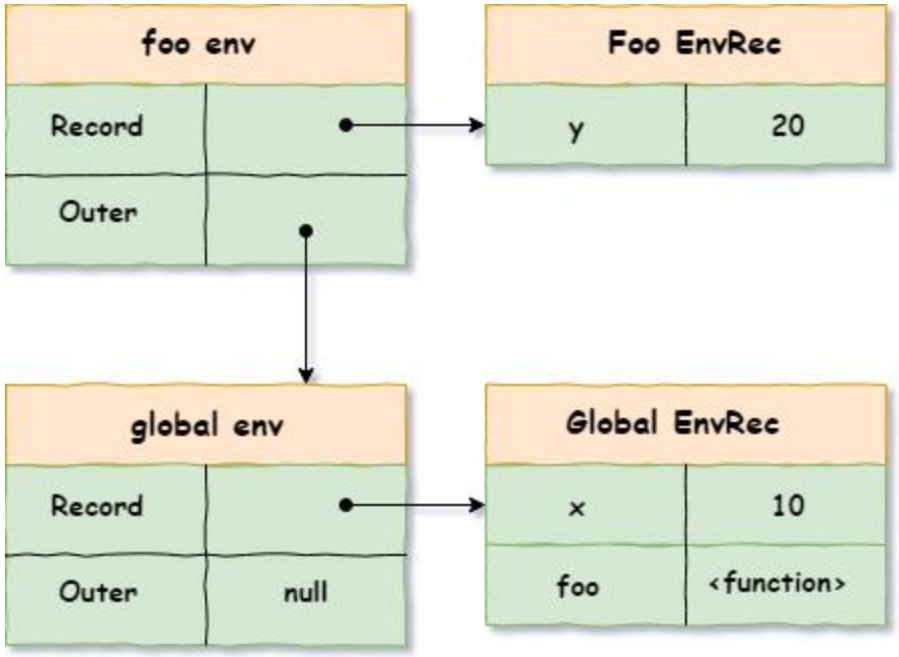
---

3. **Transcendental consciousness**. Is the home of all the laws of nature.
4. **Impulses within the transcendental field:** Thoughts encapsulated by this deep level of consciousness will result in actions in accord with all the laws of nature.
5. **Wholeness moving within itself:** In unity consciousness all thoughts and perceptions are enhanced by this supportive experience.



Lexical environment two main components: environmentRecord and reference to outer environment

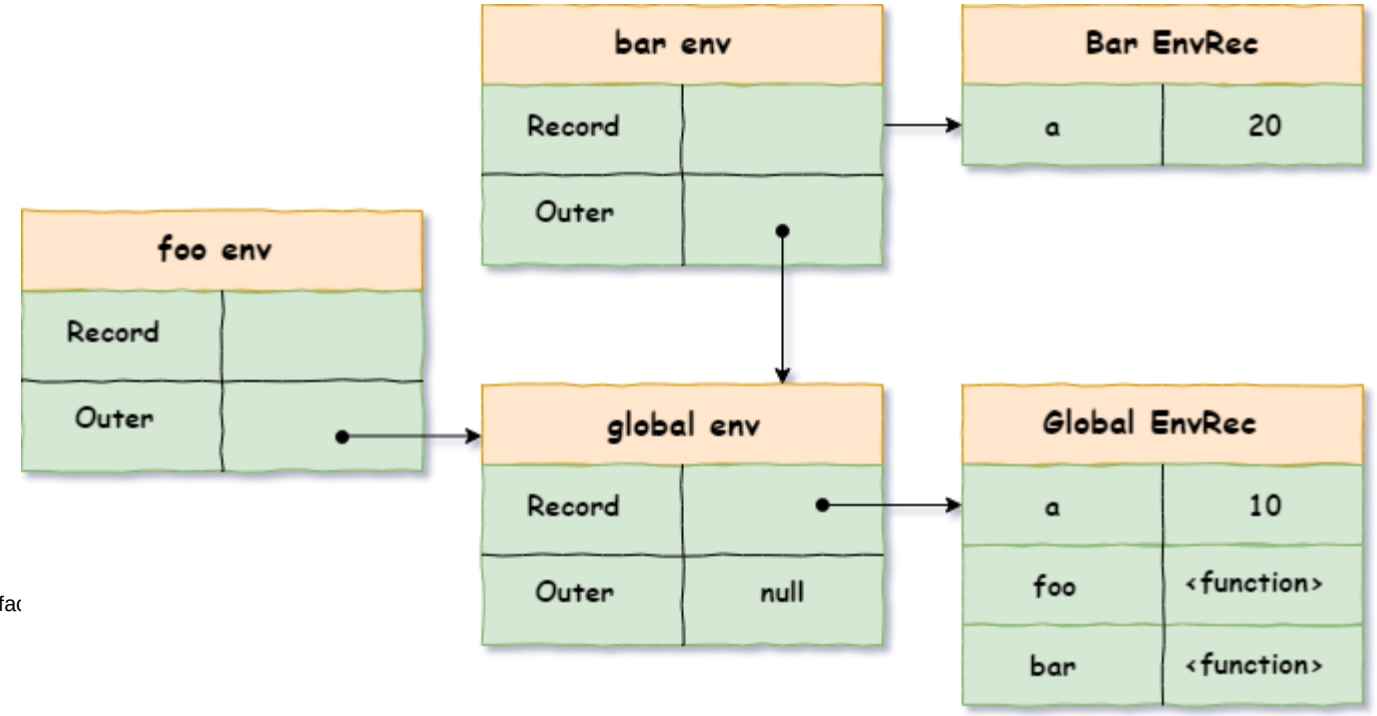
```
const x = 10;
function foo(){
  const y = 20;
 console.log(x+y); // 30
}

globalEnvironment = {// Environment of the global context
  environmentRecord: {
    // built-ins
    // our bindings:
    x: 10  },
  outer: null // no parent environment
};
fooEnvironment = {// Environment of the "foo" function
  environmentRecord: {
    y: 20
  },
  outer: globalEnvironment
};      // https://amnsingh.medium.com/lexical-environment-the-hidden-part-to-understand-closures-71d60efac0e0
```



statically capture outer binding

```
const a = 10;
function foo(){
  console.log(a);
};
function bar(){
 const a = 20;
  foo();
};
bar(); // will print "10“
```

// https://amnsingh.medium.com/lexical-environment-the-hidden-part-to-understand-closures-71d60efa



Lexical environment for closure example

```
function outer() {
 let id = 1;
return function inner(){
  console.log(id);
    }
};
const innerFunc = outer();
innerFunc(); // prints 1;
```

// https://amnsingh.medium.com/lexical-environment-the-hidden-part-to-understand-closures-71d60efac0e0