Forum Donate

Learn to code — free 3,000-hour curriculum

APRIL 29, 2016 / #JAVASCRIPT

Learn JavaScript Closures with Code Examples

Preethi Kasireddy

Closures are a fundamental JavaScript concept that every serious programmer should know inside and out.

The Internet is packed with great explanations of "what" closures are, but few deep-dive into the "why" side of things.

I find that understanding the internals ultimately gives developers a stronger grasp of their tools, so this post will be dedicated to the nuts and bolts of *how* and *why* closures work the way they do.

Hopefully you'll walk away better equipped to take advantage of closures in your day-to-day work. Let's get started!

What is a closure?

Closures are an extremely powerful property of JavaScript (and most programming languages). As defined on MDN:

Note: Free variables are variables that are neither locally declared nor passed as parameter.

Let’s look at some examples:

Example 1:

```
function numberGenerator() {
  // Local “free” variable that ends up within the closure
  var num = 1;
  function checkNumber() {
    console.log(num);
  }
  num++;
  return checkNumber;
}

var number = numberGenerator();
number(); // 2
```

In the example above, the function numberGenerator creates a local “free” variable **num** (a number) and **checkNumber** (a function which prints **num** to the console).

The function **checkNumber** doesn’t have any local variables of its own — however, it does have access to the variables within the outer function, **numberGenerator,** because of a closure.

Example 2:

In this example we'll demonstrate that a closure contains any and all local variables that were declared inside the outer enclosing function.

```
function sayHello() {
  var say = function() { console.log(hello); }
  // Local variable that ends up within the closure
  var hello = 'Hello, world!';
  return say;
}
var sayHelloClosure = sayHello();
sayHelloClosure(); // ‘Hello, world!’
```

Notice how the variable **hello** is defined *after* the anonymous function — but can still access the **hello** variable. This is because the **hello** variable has already been defined in the function "scope" at the time of creation, making it available when the anonymous function is finally executed.

(Don't worry, I'll explain what "scope" means later in the post. For now, just roll with it!)

Understanding the High Level

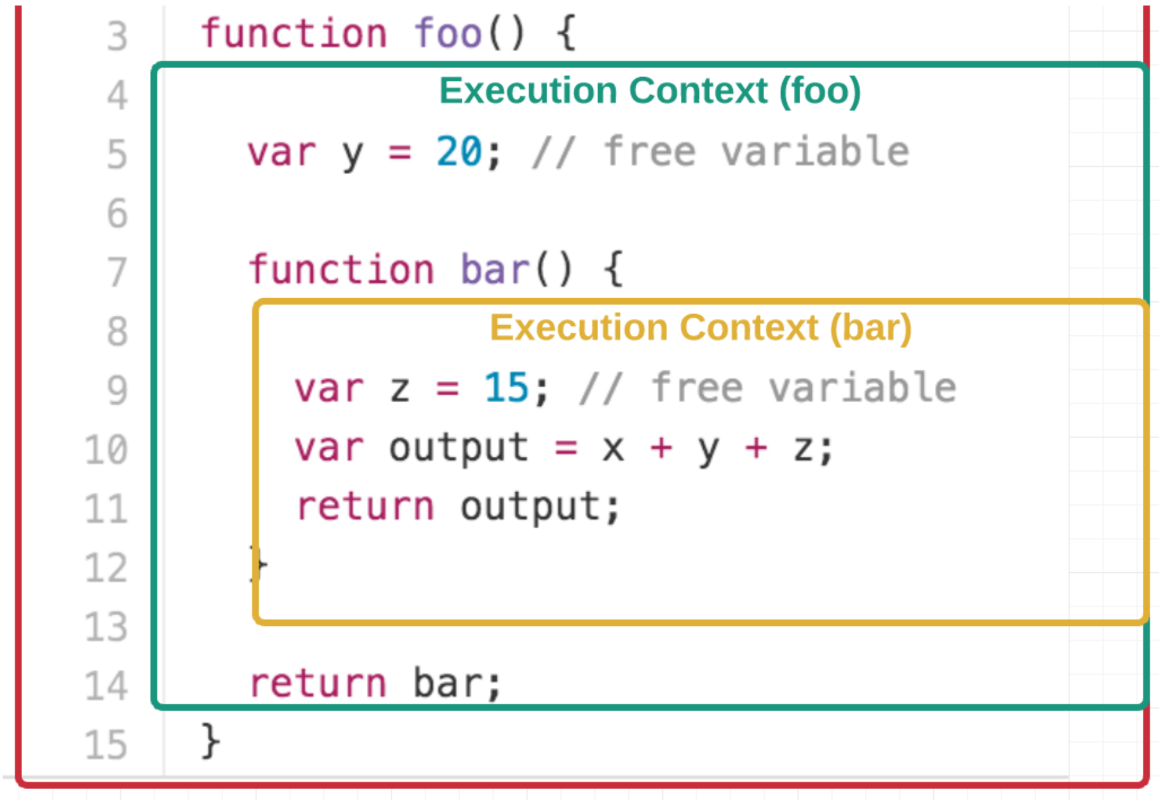
These examples illustrated "what" closures are on a high level. The general theme is this: *we have access to variables defined in enclosing*

Clearly, something is happening in the background that allows those variables to still be accessible long after the enclosing function that defined them has returned.

To understand how this is possible, we'll need to touch on a few related concepts — starting 3000 feet up and slowly climbing our way back down to the land of closures. Let's start with the overarching *context* within which a function is run, known as *"Execution context"*.

Execution Context

Execution context is an abstract concept used by the ECMAScript specification to track the runtime evaluation of code. This can be the global context in which your code is first executed or when the flow of execution enters a function body.



```
function foo() {
  var y = 20; // free variable

  function bar() {
    var z = 15; // free variable
    var output = x + y + z;
    return output;
  }

  return bar;
}
```

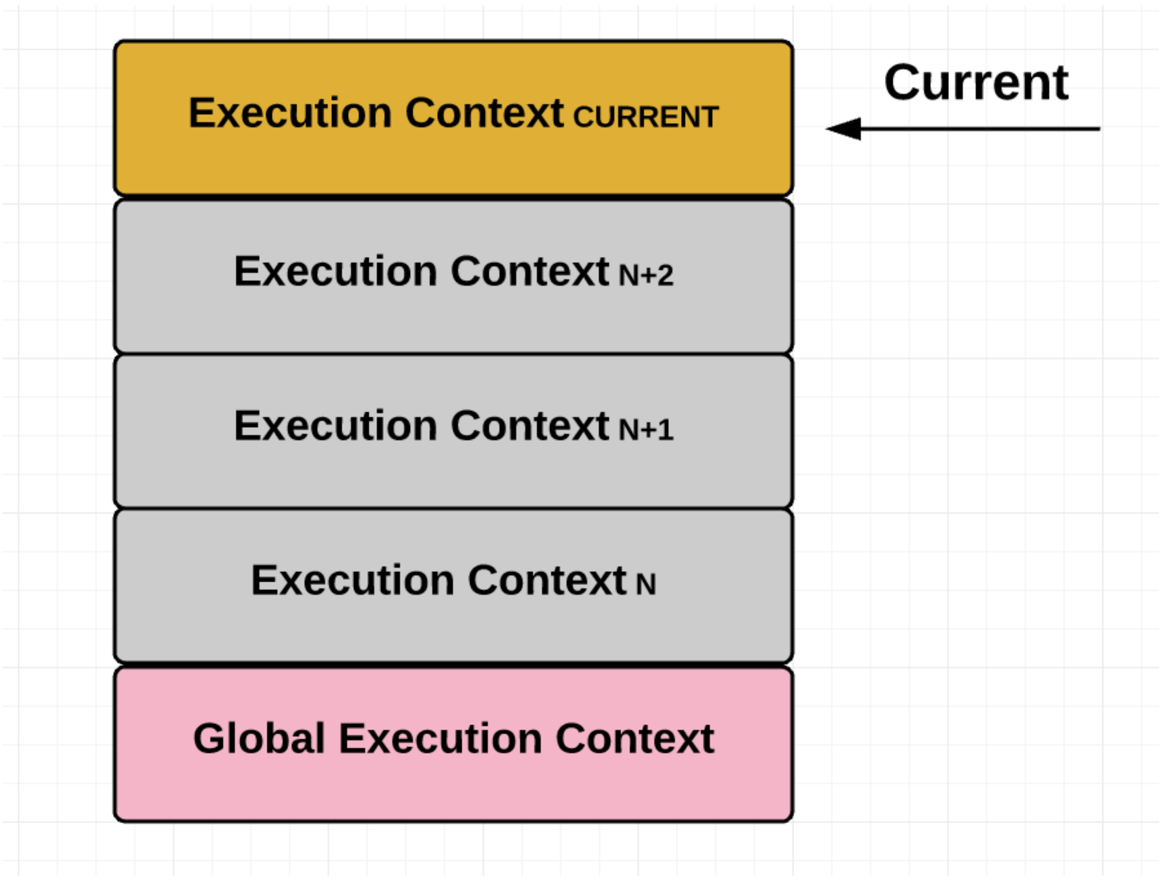
At any point in time, there can only be one execution context running. That's why JavaScript is "single threaded," meaning only one command can be processed at a time.

Typically, browsers maintain this execution context using a "stack." A stack is a Last In First Out (LIFO) data structure, meaning the last thing that you pushed onto the stack is the first thing that gets popped off it. (This is because we can only insert or delete elements at the top of the stack.)

The current or "running" execution context is always the top item in the stack. It gets popped off the top when the code in the running execution context has been completely evaluated, allowing the next top item to take over as running execution context.

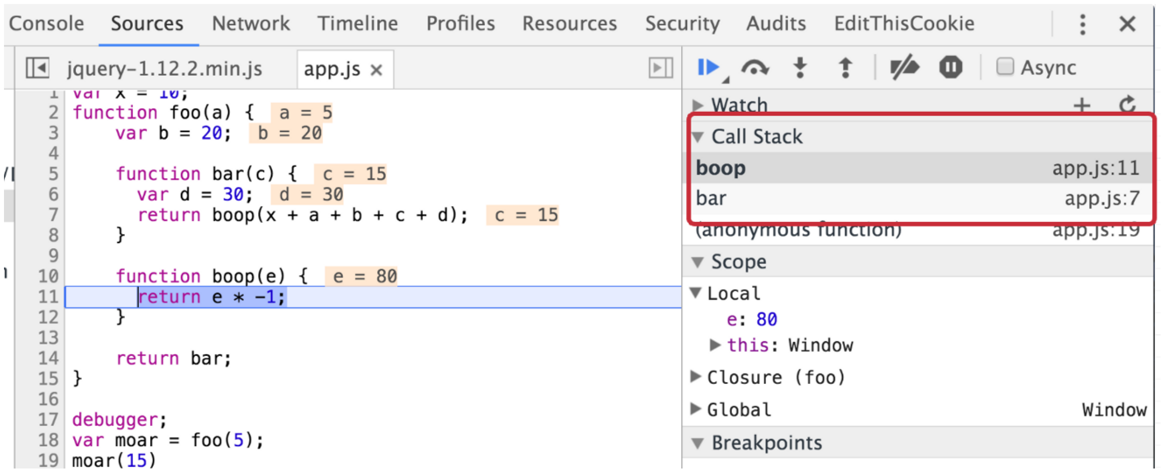
There are times when the running execution context is suspended and a different execution context becomes the running execution context. The suspended execution context might then at a later point pick back up where it left off.

Any time one execution context is replaced by another like this, a new execution context is created and pushed onto the stack, becoming the current execution context.



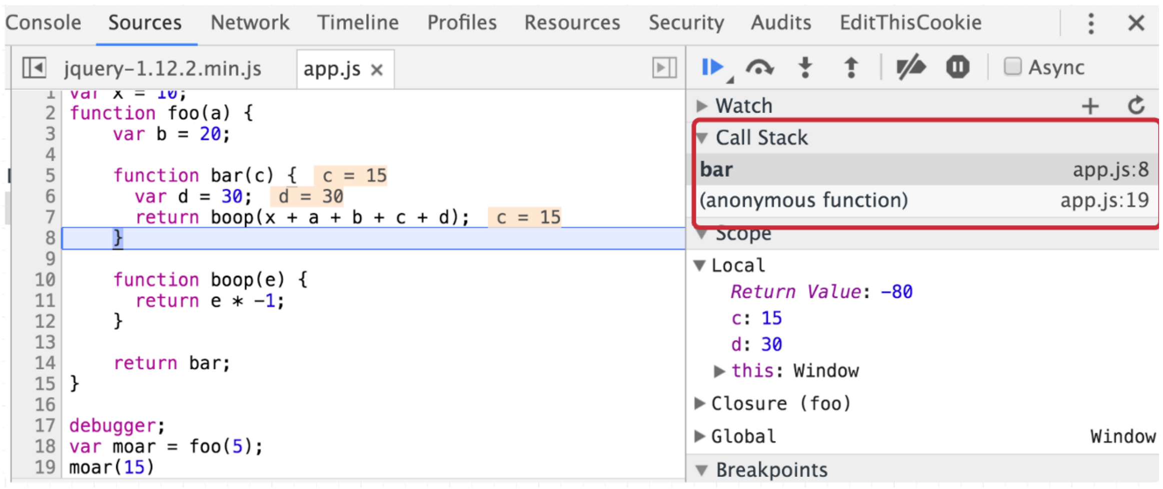
For a practical example of this concept in action in the browser, see the example below:

```
  var b = 20;

  function bar(c) {
    var d = 30;
    return boop(x + a + b + c + d);
  }

  function boop(e) {
    return e * -1;
  }

  return bar;
}

var moar = foo(5); // Closure
/*
  The function below executes the function bar which was returned
  when we executed the function foo in the line above. The function bar
  invokes boop, at which point bar gets suspended and boop gets push
  onto the top of the call stack (see the screenshot below)
*/
moar(15);
```



```
var x = 10;
function foo(a) {  a = 5
    var b = 20;  b = 20

    function bar(c) {  c = 15
      var d = 30;  d = 30
      return boop(x + a + b + c + d);  c = 15
    }

    function boop(e) {  e = 80
      return e * -1;
    }

    return bar;
}

debugger;
var moar = foo(5);
moar(15)
```

```
Call Stack
boop                    app.js:11
bar                     app.js:7
(anonymous function)    app.js:19
Scope
Local
  e: 80
  this: Window
Closure (foo)
Global                  Window
Breakpoints
```



```
var x = 10;
function foo(a) {
    var b = 20;

    function bar(c) {   c = 15
      var d = 30;   d = 30
      return boop(x + a + b + c + d);   c = 15
    }

    function boop(e) {
      return e * -1;
    }

    return bar;
}

debugger;
var moar = foo(5);
moar(15)
```

```
▶ Watch
▼ Call Stack
bar                          app.js:8
(anonymous function)         app.js:19
▼ Scope
▼ Local
    Return Value: -80
    c: 15
    d: 30
  ▶ this: Window
▶ Closure (foo)
▶ Global                     Window
▼ Breakpoints
```

When we have a bunch of execution contexts running one after another — often being paused in the middle and then later resumed — we need some way to keep track of state so we can manage the order and execution of these contexts.

And that is in fact the case. As per the ECMAScript spec, each execution context has various state components that are used to keep track of the progress the code in each context has made. These include:

- **Code evaluation state:** Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context
- **Function:** The function object which the execution context is evaluating (or null if the context being evaluated is a *script* or *module*)
- **Realm:** A set of internal objects, an ECMAScript global environment, all of the ECMAScript code that is loaded within

- **Lexical Environment:** Used to resolve identifier references made by code within this execution context.
- **Variable Environment:** Lexical Environment whose EnvironmentRecord holds bindings created by VariableStatements within this execution context.

If this sounds too confusing to you, don't worry. Of all these variables, the Lexical Environment variable is the one that's most interesting to us because it explicitly states that it resolves *"identifier references"* made by code within this execution context.

You can think of "identifiers" as variables. Since our original goal was to figure out how it's possible for us to magically access variables even after a function (or "context") has returned, Lexical Environment looks like something we should dig into!

Note*: Technically, both Variable Environment and Lexical Environment are used to implement closures. But for simplicity's sake, we'll generalize it to an "Environment". For a detailed explanation on the difference between Lexical and Variable Environment, see Dr. Alex Rauschmayer's excellent article.*

Lexical Environment

By definition:

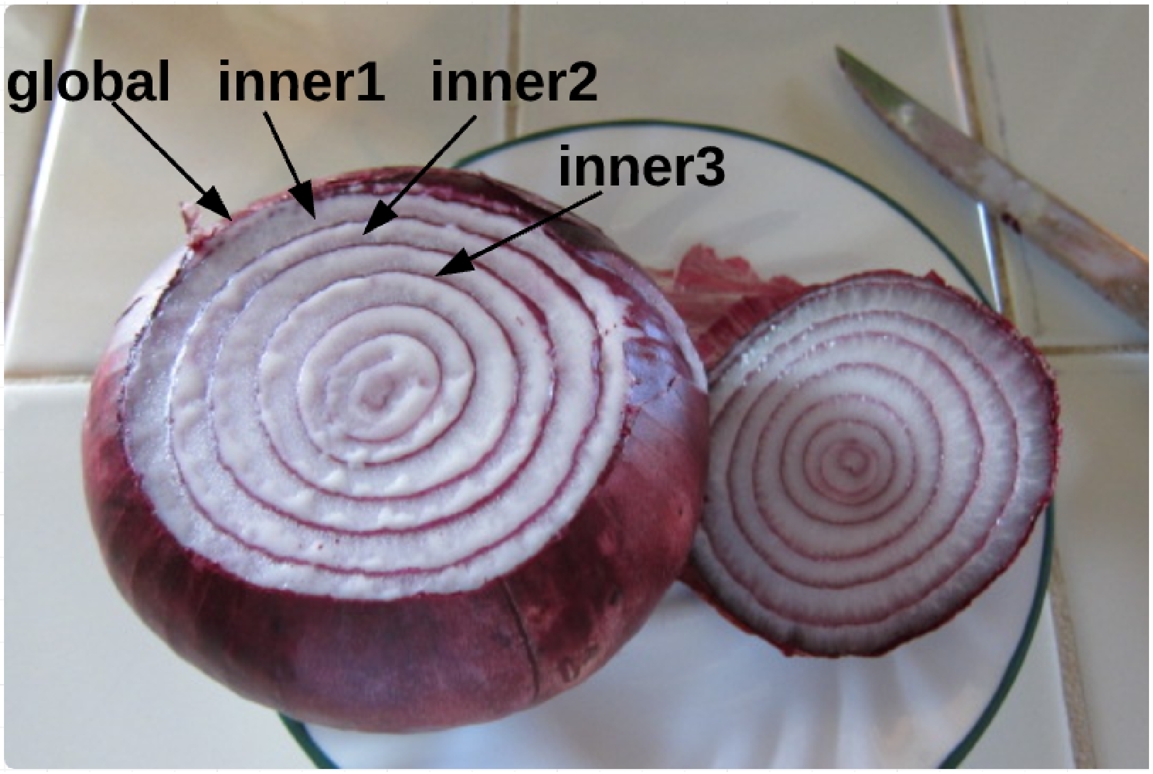
> *A Lexical Environment is a specification type used to define the association of Identifiers to specific variables and functions based*

> *Environment is associated with some specific syntactic structure of ECMAScript code such as a FunctionDeclaration, a BlockStatement, or a Catch clause of a TryStatement and a new Lexical Environment is created each time such code is evaluated. — ECMAScript-262/6.0*

Let's break this down.

- **"Used to define the association of Identifiers":** The purpose of a Lexical Environment is to manage data (i.e. identifiers) within code. In other words, it gives meaning to identifiers. For instance, if we had a line of code "*console.log(x / 10)*", it's meaningless to have a variable (or "identifier") **x** without something that provides meaning for that variable. The Lexical Environments provides this meaning (or "association") via its Environment Record (see below).
- **"Lexical Environment consists of an Environment Record":** An Environment Record is a fancy way to say that it keeps a record of all identifiers and their bindings that exist within a Lexical Environment. Every Lexical Environment has it's own Environment Record.
- **"Lexical nesting structure":** This is the interesting part, which is basically saying that an inner environment references the outer environment that surrounds it, and that this outer environment can have its own outer environment as well. As a result, an environment can serve as the outer environment for more than one inner environment. The global environment is the only Lexical environment that does not have an outer

onion; every subsequent layer below is nested within.



Abstractly, the environment looks like this in pseudocode:

```
LexicalEnvironment = {
  EnvironmentRecord: {
  // Identifier bindings go here
  },

  // Reference to the outer environment
  outer: < >
};
```

come back to this point again at the end. *(Side note: a function is not the only way to create a Lexical Environment. Others include a block statement or a catch clause. For simplicity's sake, I'll focus on environment created by functions throughout this post)*

In short, every execution context has a Lexical Environment. This Lexical environments holds variables and their associated values, and also has a reference to its outer environment.

The Lexical Environment can be the global environment, a module environment (which contains the bindings for the top level declarations of a Module), or a function environment (environment created due to the invocation of a function).

Scope Chain

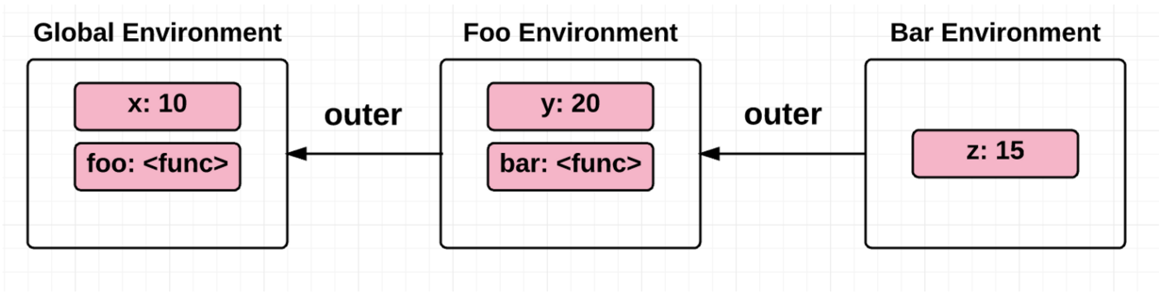
Based on the above definition, we know that an environment has access to its parent's environment, and its parent environment has access to its parent environment, and so on. This set of identifiers that each environment has access to is called *"scope."* We can nest scopes into a hierarchical chain of environments known as the *"scope chain"*.

Let's look at an example of this nesting structure:

```
var x = 10;

function foo() {
  var y = 20; // free variable
```

```
    return bar;
  }
```

As you can see, **bar** is nested within **foo**. To help you visualize the nesting, see the diagram below:



We'll revisit this example later in the post.

This scope chain, or chain of environments associated with a function, is saved to the function object at the time of its creation. In other words, it's defined statically by location within the source code. (This is also known as "lexical scoping".)

Let's take a quick detour to understand the difference between "dynamic scope" and "static scope", which will help clarify why static scope (or lexical scope) is necessary in order to have closures.

Detour: Dynamic Scope vs. Static Scope

determines what variable you are referring to.

On the other hand, static scope is when the variables referenced in a context are recorded at the *time of creation*. In other words, the structure of the program source code determines what variables you are referring to.

At this point, you might be wondering how dynamic scope and static scope are different. Here's two examples to help illustrate:

Example 1:

```
var x = 10;

function foo() {
  var y = x + 5;
  return y;
}

function bar() {
  var x = 2;
  return foo();
}

function main() {
  foo(); // Static scope: 15; Dynamic scope: 15
  bar(); // Static scope: 15; Dynamic scope: 7
  return 0;
}
```

We see above that the static scope and dynamic scope return different values when the function bar is invoked.

being 15.

Dynamic scope, on the other hand, gives us a stack of variable definitions tracked at runtime — such that which **x** we use depends on what exactly is in scope and has been defined dynamically at runtime. Running the function **bar** pushes x = 2 onto the top of the stack, making **foo** return 7.

Example 2:

```
var myVar = 100;

function foo() {
  console.log(myVar);
}

foo(); // Static scope: 100; Dynamic scope: 100

(function () {
  var myVar = 50;
  foo(); // Static scope: 100; Dynamic scope: 50
})();

// Higher-order function
(function (arg) {
  var myVar = 1500;
  arg();  // Static scope: 100; Dynamic scope: 1500
})(foo);
```

Similarly, in the dynamic scope example above the variable **myVar** is resolved using the value of **myVar** at the place where the function is

As you can see, dynamic scope often leads to some ambiguity. It's not exactly made clear which scope the free variable will be resolved from.

Closures

Some of that may strike you as off-topic, but we've actually covered everything we need to know to understand closures:

Every function has an execution context, which comprises of an environment that gives meaning to the variables within that function and a reference to its parent's environment. A reference to the parent's environment makes all variables in the parent scope available for all inner functions, regardless of whether the inner function(s) are invoked outside or inside the scope in which they were created.

So, it appears as if the function "remembers" this environment (or scope) because the function literally has a reference to the environment (and the variables defined in that environment)!

Coming back to the nested structure example:

```
var x = 10;

function foo() {
  var y = 20; // free variable
  function bar() {
    var z = 15; // free variable
    return x + y + z;
```

```
var test = foo();

test(); // 45
```

Based on our understanding of how environments work, we can say that the environment definitions for the above example look something like this (note, this is purely pseudocode):

```
GlobalEnvironment = {
  EnvironmentRecord: {
    // built-in identifiers
    Array: '<func>',
    Object: '<func>',
    // etc..

    // custom identifiers
    x: 10
  },
  outer: null
};

fooEnvironment = {
  EnvironmentRecord: {
    y: 20,
    bar: '<func>'
  }
  outer: GlobalEnvironment
};

barEnvironment = {
  EnvironmentRecord: {
    z: 15
  }
  outer: fooEnvironment
};
```

because **bar** has a reference to **y** through its outer environment, which is **foo**'s environment! **bar** also has access to the global variable **x** because **foo**'s environment has access to the global environment. This is called *"scope-chain lookup."*

Returning to our discussion of dynamic scope vs static scope: for closures to be implemented, we can't use dynamic scoping via a dynamic stack to store our variables.

The reason is because it would mean that when a function returns, the variables would be popped off the stack and no longer available — which contradicts our initial definition of a closure.

What happens instead is that the closure data of the parent context is saved in what's known as the "heap," which allows for the data to persist after the function call that made them returns (i.e. even after the execution context is popped off the execution call stack).

Make sense? Good! Now that we understand the internals on an abstract level, let's look at a couple more examples:

Example 1:

One canonical example/mistake is when there's a for-loop and we try to associate the counter variable in the for-loop with some function in the for-loop:

```
    result[i] = function () {
      console.log(i);
    };
  }

  result[0](); // 5, expected 0
  result[1](); // 5, expected 1
  result[2](); // 5, expected 2
  result[3](); // 5, expected 3
  result[4](); // 5, expected 4
```

Going back to what we just learned, it becomes super easy to spot the mistake here! Abstractly, here's what the environment looks like this by the time the for-loop exits:

```
environment: {
  EnvironmentRecord: {
    result: [...],
    i: 5
  },
  outer: null,
}
```

The incorrect assumption here was that the scope is different for all five functions within the result array. Instead, what's actually happening is that the environment (or context/scope) is the same for all five functions within the result array. Therefore, every time the variable **i** is incremented, it updates scope — which is shared by all the functions. That's why any of the 5 functions trying to access **i** returns 5 (i is equal to 5 when the for-loop exits).

```
var result = [];

for (var i = 0; i < 5; i++) {
  result[i] = (function inner(x) {
    // additional enclosing context
    return function() {
      console.log(x);
    }
  })(i);
}

result[0](); // 0, expected 0
result[1](); // 1, expected 1
result[2](); // 2, expected 2
result[3](); // 3, expected 3
result[4](); // 4, expected 4
```

Yay! That fixed it :)

Another, rather clever approach is to use **let** instead of **var**, since **let** is block-scoped and so a new identifier binding is created for each iteration in the for-loop:

```
var result = [];

for (let i = 0; i < 5; i++) {
  result[i] = function () {
    console.log(i);
  };
}

result[0](); // 0, expected 0
```

Tada! :)

Example 2:

In this example, we'll show how each *call* to a function creates a new separate closure:

```
function iCantThinkOfAName(num, obj) {
  // This array variable, along with the 2 parameters passed in,
  // are 'captured' by the nested function 'doSomething'
  var array = [1, 2, 3];
  function doSomething(i) {
    num += i;
    array.push(num);
    console.log('num: ' + num);
    console.log('array: ' + array);
    console.log('obj.value: ' + obj.value);
  }

  return doSomething;
}

var referenceObject = { value: 10 };
var foo = iCantThinkOfAName(2, referenceObject); // closure #1
var bar = iCantThinkOfAName(6, referenceObject); // closure #2

foo(2);
/*
  num: 4
  array: 1,2,3,4
  obj.value: 10
*/

bar(2);
```

```
*/

referenceObject.value++;

foo(4);
/*
  num: 8
  array: 1,2,3,4,8
  obj.value: 11
*/

bar(4);
/*
  num: 12
  array: 1,2,3,8,12
  obj.value: 11
*/
```

In this example, we can see that each call to the function **iCantThinkOfAName** creates a new closure, namely **foo** and **bar**. Subsequent invocations to either closure functions updates the closure variables within that closure itself, demonstrating that the variables in *each* closure continue to be usable by **iCantThinkOfAName**'s **doSomething** function long after **iCantThinkOfAName** returns.

Example 3:

```
function mysteriousCalculator(a, b) {
    var mysteriousVariable = 3;
    return {
        add: function() {
            var result = a + b + mysteriousVariable;
            return toFixedTwoPlaces(result);
```

```
                return toFixedTwoPlaces(result);
            }
        }
}

function toFixedTwoPlaces(value) {
    return value.toFixed(2);
}

var myCalculator = mysteriousCalculator(10.01, 2.01);
myCalculator.add() // 15.02
myCalculator.subtract() // 5.00
```

What we can observe is that **mysteriousCalculator** is in the global scope, and it returns two functions. Abstractly, the environments for the example above look like this:

```
GlobalEnvironment = {
  EnvironmentRecord: {
    // built-in identifiers
    Array: '<func>',
    Object: '<func>',
    // etc...

    // custom identifiers
    mysteriousCalculator: '<func>',
    toFixedTwoPlaces: '<func>',
  },
  outer: null,
};

mysteriousCalculatorEnvironment = {
  EnvironmentRecord: {
    a: 10.01,
    b: 2.01,
    mysteriousVariable: 3,
  }
  EnvironmentRecord: {
    result: 15.02
  }
  outer: mysteriousCalculatorEnvironment,
};

subtractEnvironment = {
  EnvironmentRecord: {
    result: 5.00
  }
  outer: mysteriousCalculatorEnvironment,
};
```

Because our **add** and **subtract** functions have a reference to the **mysteriousCalculator** function environment, they're able to make use of the variables in that environment to calculate the result.

Example 4:

One final example to demonstrate an important use of closures: to maintain a private reference to a variable in the outer scope.

```
function secretPassword() {
  var password = 'xh38sk';
  return {
    guessPassword: function(guess) {
      if (guess === password) {
        return true;
      } else {
        return false;
      }
    }
  }
}
```

This is a very powerful technique — it gives the closure function **guessPassword** exclusive access to the **password** variable, while making it impossible to access the **password** from the outside.

TL;DR

- Execution context is an abstract concept used by the ECMAScript specification to track the runtime evaluation of code. At any point in time, there can only be one execution context that is executing code.
- Every execution context has a Lexical Environment. This Lexical environments holds identifier bindings (i.e. variables and their associated values), and also has a reference to its outer environment.
- The set of identifiers that each environment has access to is called "scope." We can nest these scopes into a hierarchical chain of environments, known as the "scope chain".
- Every function has an execution context, which comprises of a Lexical Environment that gives meaning to the variables within that function and a reference to its parent's environment. And so it appears as if the function "remembers" this environment (or scope) because the function literally has a reference to this environment. This is a closure.
- A closure is created every time an enclosing outer function is called. In other words, the inner function does not need to

defined statically by its location within the source code.

- Closures have many practical use cases. One important use case is to maintain a private reference to a variable in the outer scope.

Clos(ure)ing remarks

I hope this post was helpful and gave you a mental model for how closures are implemented in JavaScript. As you can see, understanding the nuts and bolts of how they work makes it much easier to spot closures — not to mention saving a lot of headache when it's time to debug.

PS: I'm human and make mistakes — so if you find any mistakes I'd love for you to let me know!

Further Reading

For the sake of brevity I left out a few topics that might be interesting to some readers. Here are some links that I wanted to share:

- **What's the VariableEnvironment within an execution context?** Dr. Axel Rauschmayer does a phenomenol job explaining it so I'll leave you off with a link to his blog post: http://www.2ality.com/2011/04/ecmascript-5-spec-lexicalenvironment.html
- **What are the different types of Environment Records?** Read the spec here: http://www.ecma-international.org/ecma-

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

- Others? Please suggest and I'll add them!

Preethi Kasireddy

Read more posts.