Understanding Closures in JavaScript

Learn How Closures Work in JavaScript: A Hands-on guide

Sukhjinder Arora · Follow

Published in Bits and Pieces

9 min read · Sep 24, 2018

Photo by Adi Goldstein on Unsplash

Closures are a fundamental concept of JavaScript that every JavaScript developer should know and understand. Yet, it's a concept that confuses many new JavaScript developers.

Having a proper understanding of closures will help you to write better, more efficient and clean code. Which will, in turn, help you to become a better JavaScript developer.

So in this article, I will try to explain the internals of closures and how they really work in JavaScript.

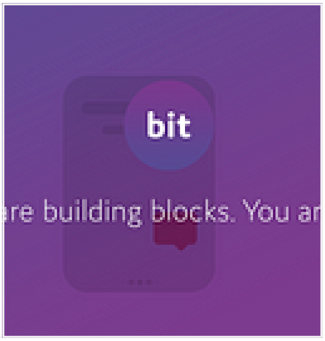
So without further ado, Let's get started :)

Tip: Use Bit to reuse components between apps. It helps your team organize and share JS components, so you can build new apps faster. Give it a try.

Component Discovery and Collaboration · Bit

Bit is where developers share components and collaborate to build amazing software together. Discover components shared...

bit.dev



React sinners with Bit: choose, play, install

. . .

What is Closure?

A closure is a function that has access to its outer function scope even after the outer function has returned. This means a closure can remember and access variables and arguments of its outer function even after the function has finished.

Before we dive into closures, let's first understand the lexical scope.

What is a Lexical Scope?

A lexical scope or static scope in JavaScript refers to the accessibility of the variables, functions, and objects based on their physical location in the source code. For example:

```
let a = 'global';

  function outer() {
    let b = 'outer';

    function inner() {
      let c = 'inner'
      console.log(c);   // prints 'inner'
      console.log(b);   // prints 'outer'
      console.log(a);   // prints 'global'
    }
    console.log(a);     // prints 'global'
    console.log(b);     // prints 'outer'
    inner();
  }
outer();
console.log(a);         // prints 'global'
```

Here the `inner` function can access the variables defined in its own scope, the `outer` function's scope, and the global scope. And the `outer` function can access the variable defined in its own scope and the global scope.

So a scope chain of the above code would be like this:

```
Global {
  outer {
    inner
  }
}
```

Notice that `inner` function is surrounded by the lexical scope of `outer` function which is, in turn, surrounded by the global scope. That's why the `inner` function can access the variables defined in `outer` function and the global scope.

Practical Examples of Closure

Let's look at some practical examples of closures before diving into how closures work.

Example 1#

```
function person() {
  let name = 'Peter';

  return function displayName() {
    console.log(name);
  };
}

let peter = person();
peter(); // prints 'Peter'
```

In this code, we are calling `person` function which returns inner function `displayName` and stores that inner function in `peter` variable. When we call `peter` function (which is actually referencing the `displayName` function), the name 'Peter' is printed to the console.

But we don't have any variable named `name` in `displayName` function, so this function can somehow access the variable of its outer function `person` even after that function has returned. So the `displayName` function is actually a closure.

Example 2#

```
function getCounter() {
  let counter = 0;
  return function() {
    return counter++;
  }
}
```

```
let count = getCounter();

console.log(count());  // 0
console.log(count());  // 1
console.log(count());  // 2
```

Again we are storing the anonymous inner function returned by `getCounter` function into the `count` variable. As `count` function is now a closure, it can access the `counter` variable of `getCounter` function even after `getCounter()` has returned.

But notice that the value of the `counter` is not resetting to `0` on each `count` function call as it usually should.

That's because, at each call of `count()`, a new scope for the function is created, but there is only single scope created for `getCounter` function, because the `counter` variable is defined in the scope of `getCounter()`, it would get incremented on each `count` function call instead of resetting to `0`.

How do Closures Work?

Up until now, we have discussed what closures are and their practical examples. Now let's understand how closures really work in JavaScript.

To really understand how closures work in JavaScript, we have to understand the two most important concepts in JavaScript, that is, 1) Execution Context and 2) Lexical Environment.

Execution Context

An execution context is an abstract environment where the JavaScript code is evaluated and executed. When the global code is executed, it's executed inside the global execution context, and the function code is executed inside the function execution context.

There can only be one currently running execution context (Because JavaScript is single threaded language), which is managed by a stack data structure known as Execution Stack or Call Stack.

An execution stack is a stack with LIFO (Last in, first out) structure in which items can only be added or removed from the top of the stack only.

The currently running execution context will be always on the top of the stack, and when the function which is currently running completes, its execution context is popped off from the stack and the control reaches to the execution context below it in the stack.

Let’s look at a code snippet to better understand execution context and stack:

Execution Context Example

When this code is executed, the JavaScript engine creates a global execution context to execute the global code, and when it encounters the call to `first()` function, it creates a new execution context for that function and pushes it to the top of the execution stack.

So the execution stack for the above code looks like this:

Execution Stack

When the `first()` function completes, its execution stack is removed from the stack, and the control reaches to execution context below it, that is, global execution context. So the remaining code in global scope will be executed.

Lexical Environment

Every time the JavaScript engine creates an execution context to execute the function or global code, it also creates a new lexical environment to store the variable defined in that function during the execution of that function.

A *lexical environment* is a data structure that holds **identifier-variable mapping.** (here **identifier** refers to the name of variables/functions, and the **variable** is the reference to actual object [including function type object] or primitive value).

A Lexical Environment has two components: (1) the **environment record** and (2) a **reference to the outer environment**.

1. The **environment record** is the actual place where the variable and function declarations are stored.
2. The **reference to the outer environment** means it has access to its outer (parent) lexical environment. This component is the most important in order to understand how closures work.

A lexical environment conceptually looks like this:

```
lexicalEnvironment = {
  environmentRecord: {
    <identifier> : <value>,
    <identifier> : <value>
  }
  outer: < Reference to the parent lexical environment>
}
```

So let’s again take a look at above code snippet:

```
let a = 'Hello World!';

function first() {
  let b = 25;
  console.log('Inside first function');
}
first();
console.log('Inside global execution context');
```

When the JavaScript engine creates a global execution context to execute global code, it also creates a new lexical environment to store the variables and functions defined in the global scope. So the lexical environment for the global scope will look like this:

```
globalLexicalEnvironment = {
  environmentRecord: {
      a     : 'Hello World!',
      first : < reference to function object >
  }
  outer: null
}
```

Here the outer lexical environment is set to `null` because there is no outer lexical environment for the global scope.

When the engine creates execution context for `first()` function, it also creates a lexical environment to store variables defined in that function during execution of the function. So the lexical environment of the function will look like this:

```
functionLexicalEnvironment = {
  environmentRecord: {
      b    : 25,
  }
  outer: <globalLexicalEnvironment>
}
```

The outer lexical environment of the function is set to the global lexical environment because the function is surrounded by the global scope in the source code.

Note — When a function completes, its execution context is removed from the stack, but its lexical environment may or may not be removed from the memory depending on if that lexical environment is referenced by any other lexical environments in their outer lexical environment property.

Detailed Closures Examples

Now that we understand the execution context and lexical environment, let's get back to the closures.

Example 1#

Take a look at this code snippet:

```
function person() {
  let name = 'Peter';

  return function displayName() {
    console.log(name);
  };
}

let peter = person();
peter(); // prints 'Peter'
```

When the `person` function is executed, the JavaScript engine creates a new execution context and lexical environment for the function. After this function finishes, it returns `displayName` function and assigns it to `peter` variable.

So its lexical environment will look like this:

```
personLexicalEnvironment = {
  environmentRecord: {
    name : 'Peter',
    displayName: < displayName function reference>
  }
  outer: <globalLexicalEnvironment>
}
```

When the `person` function finishes, its execution context is removed from the stack. But its lexical environment is still in the memory because its lexical environment is referenced by the lexical environment of its inner `displayName` function. So its variables are still available in the memory.

Please note that when the `personLexicalEnvironment` is created, the JavaScript engine attaches the `personLexicalEnvironment` to all of the function definitions inside that lexical environment. So that later on if any of the inner functions are called, the JavaScript engine can set the outer lexical environment to the lexical environment attached to that function definition.

When the `peter` function is executed (which is actually a reference to the `displayName` function), the JavaScript engine creates a new execution context and lexical environment for that function.

So its lexical environment looks like this:

```
displayNameLexicalEnvironment = {
  environmentRecord: {

  }
  outer: <personLexicalEnvironment>
}
```

As there's no variable in `displayName` function, its environment record will be empty. During the execution of this function, the JavaScript engine will try to find the variable `name` in the function's lexical environment.

As there are no variables in the lexical environment of `displayName` function, it will look into the outer lexical environment, that is, the lexical environment of the `person` function which still there in the memory. The JavaScript engine finds the variable and `name` is printed to the console.

Example 2#

```
function getCounter() {
  let counter = 0;
  return function() {
    return counter++;
  }
}

let count = getCounter();

console.log(count());  // 0
console.log(count());  // 1
console.log(count());  // 2
```

Again the lexical environment for the `getCounter` function will look like this:

```
getCounterLexicalEnvironment = {
  environmentRecord: {
    counter: 0,
    <anonymous function> : < reference to function>
  }
  outer: <globalLexicalEnvironment>
}
```

This function returns an anonymous function and assigns it to `count` variable.

When the `count` function is executed, its lexical environment will look like this:

```
countLexicalEnvironment = {
  environmentRecord: {

  }
  outer: <getCountLexicalEnvironment>
}
```

When the `count` function is called, the JavaScript engine will look into the lexical environment of this function for the `counter` variable. Again as its environment record is empty, the engine will look into the outer lexical environment of the function.

The engine finds the variable, prints it to the console and will increment the counter variable in the `getCounter` function lexical environment.

So the lexical environment for the `getCounter` function after first call `count` function will look like this:

```
getCounterLexicalEnvironment = {
  environmentRecord: {
    counter: 1,
    <anonymous function> : < reference to function>
  }
  outer: <globalLexicalEnvironment>
}
```

On each call to the `count` function, the JavaScript engine creates a new lexical environment for the `count` function, increments the `counter` variable and updates the lexical environment of `getCounter` function to reflect changes.

Conclusion

So we have learned what closures are and how they really work. Closures are fundamental concepts of JavaScript that every JavaScript developer should understand. Having a good knowledge of these concepts will help you to become a much more effective and better JavaScript developer.

That’s it and if you found this article helpful, please click the clap 👏button below, you can also follow me on Medium and Twitter, and if you have any doubt, feel free to comment! I’d be happy to help :)

. . .

Learn more

A Practical Guide to Regular Expressions (RegEx) In JavaScript

A quick guide to effectively leveraging regular expressions.

blog.bitsrc.io