How do JavaScript closures work?

Asked 16 years, 3 months ago Modified 1 year, 5 months ago

Viewed 1.6m times

7613

This question's answers are a community effort. Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

How would you explain JavaScript closures to someone with a knowledge of the concepts they consist of (for example functions, variables and the like), but does not understand closures themselves?

I have seen the Scheme example given on Wikipedia, but unfortunately it did not help.

javascript function variables scope closures

Share Follow

edited Apr 9, 2017 at 13:55

community wiki
28 revs, 21 users 17%
Zaheer Ahmed

Comments disabled on deleted / locked posts / reviews

86 Answers

Sorted by: Highest score (default)

1 2 3 Next

8347

+100

A closure is a pairing of:

1. A function and
2. A reference to that function's outer scope (lexical environment)

A lexical environment is part of every execution context (stack frame) and is a map between identifiers (i.e. local variable names) and values.

Every function in JavaScript maintains a reference to its outer lexical environment. This reference is used to configure the execution context created when a function is invoked. This reference enables code inside the function to "see" variables declared outside the function, regardless of when and where the function is called.

If a function was called by a function, which in turn was called by another function, then a chain of references to outer lexical environments is created. This chain is called the scope chain.

In the following code, `inner` forms a closure with the lexical environment of the execution context created when `foo` is invoked, *closing over* variable `secret`:

```
function foo() {
  const secret = Math.trunc(Math.random() * 100)
  return function inner() {
    console.log(`The secret number is ${secret}.`)
  }
}
const f = foo() // `secret` is not directly accessib
f() // The only way to retrieve `secret` is to invok
```

Run code snippet   Expand snippet

In other words: in JavaScript, functions carry a reference to a private "box of state", to which only they (and any other functions declared within the same lexical environment) have access. This box of the state is invisible to the caller of the function, delivering an excellent mechanism for data-hiding and encapsulation.

And remember: functions in JavaScript can be passed around like variables (first-class functions), meaning these pairings of functionality and state can be passed around your program, similar to how you might pass an instance of a class around in C++.

If JavaScript did not have closures, then more states would have to be passed between functions *explicitly*, making parameter lists longer and code noisier.

So, if you want a function to always have access to a private piece of state, you can use a closure.

...and frequently we *do* want to associate the state with a function. For example, in Java or C++, when you add a private instance variable and a method to a class, you are associating the state with functionality.

In C and most other common languages, after a function returns, all the local variables are no longer accessible because the stack-frame is destroyed. In JavaScript, if you declare a function within another function, then the local variables of the outer function can remain accessible after returning from it. In this way, in the code above, `secret` remains available to the function object `inner`, *after* it has been returned from `foo`.

Uses of Closures

Closures are useful whenever you need a private state associated with a function. This is a very common scenario - and remember: JavaScript did not have a class syntax until 2015, and it still does not have a private field syntax. Closures meet this need.

Private Instance Variables

In the following code, the function `toString` closes over the details of the car.

```
function Car(manufacturer, model, year, color) {
  return {
    toString() {
      return `${manufacturer} ${model} (${year}, ${c
      }
    }
  }

  const car = new Car('Aston Martin', 'V8 Vantage', '2
  console.log(car.toString())
```

Run code snippet   Expand snippet

Functional Programming

In the following code, the function `inner` closes over both `fn` and `args`.

```
function curry(fn) {
  const args = []
  return function inner(arg) {
    if(args.length === fn.length) return fn(...args)
    args.push(arg)
    return inner
  }
}

function add(a, b) {
  return a + b
}

const curriedAdd = curry(add)
console.log(curriedAdd(2)(3)()) // 5
```

Run code snippet   Expand snippet

Event-Oriented Programming

In the following code, function `onClick` closes over variable `BACKGROUND_COLOR`.

```
const $ = document.querySelector.bind(document)
const BACKGROUND_COLOR = 'rgba(200, 200, 242, 1)'

function onClick() {
  $('body').style.background = BACKGROUND_COLOR
}

$('button').addEventListener('click', onClick)
```

```
<button>Set background color</button>
```

Run code snippet   Expand snippet

Modularization

In the following example, all the implementation details are hidden inside an immediately executed function expression. The functions `tick` and `toString` close over the private state and functions they need to complete their work. Closures have enabled us to modularize and encapsulate our code.

```
let namespace = {};

(function foo(n) {
  let numbers = []
  function format(n) {
    return Math.trunc(n)
  }

  function tick() {
    numbers.push(Math.random() * 100)
  }

  function toString() {
    return numbers.map(format)
  }

  n.counter = {
    tick,
    toString
  }
}(namespace))

const counter = namespace.counter
counter.tick()
counter.tick()
console.log(counter.toString())
```

Run code snippet   Expand snippet

Examples

Example 1

This example shows that the local variables are not copied in the closure: the closure maintains a reference to the original variables *themselves*. It is as though the stack-frame stays alive in memory even after the outer function exits.

```
function foo() {
  let x = 42
  let inner = () => console.log(x)
  x = x + 1
  return inner
}

foo()() // logs 43
```

Run code snippet   Expand snippet

Example 2

In the following code, three methods `log`, `increment`, and `update` all close over the same lexical environment.

And every time `createObject` is called, a new execution context (stack frame) is created and a completely new variable `x`, and a new set of functions (`log` etc.) are created, that close over this new variable.

```
function createObject() {
  let x = 42;
  return {
    log() { console.log(x) },
    increment() { x++ },
    update(value) { x = value }
  }
}

const o = createObject()
o.increment()
o.log() // 43
o.update(5)
```

```
o.log() // 5
const p = createObject()
p.log() // 42
```

Run code snippet   Expand snippet

Example 3

If you are using variables declared using `var`, be careful you understand which variable you are closing over. Variables declared using `var` are hoisted. This is much less of a problem in modern JavaScript due to the introduction of `let` and `const`.

In the following code, each time around the loop, a new function `inner` is created, which closes over `i`. But because `var i` is hoisted outside the loop, all of these inner functions close over the same variable, meaning that the final value of `i` (3) is printed, three times.

```
function foo() {
  var result = []
  for (var i = 0; i < 3; i++) {
    result.push(function inner() { console.log(i) }
  }

  return result
}

const result = foo()
// The following will print `3`, three times...
for (var i = 0; i < 3; i++) {
  result[i]()
}
```

Run code snippet   Expand snippet

Final points:

- Whenever a function is declared in JavaScript closure is created.
- Returning a `function` from inside another function is the classic example of closure, because the state inside the outer function is implicitly available to the returned inner function, even after the outer function has completed execution.
- Whenever you use `eval()` inside a function, a closure is used. The text you `eval` can reference local variables of the function, and in the non-strict mode, you can even create new local variables by using `eval('var foo = …')`.
- When you use `new Function(…)` (the Function constructor) inside a function, it does not close over its lexical environment: it closes over the global context instead. The new function cannot reference the local variables of the outer function.
- A closure in JavaScript is like keeping a reference (**NOT** a copy) to the scope at the point of function declaration, which in turn keeps a reference to its

outer scope, and so on, all the way to the global object at the top of the scope chain.

- A closure is created when a function is declared; this closure is used to configure the execution context when the function is invoked.
- A new set of local variables is created every time a function is called.

Links

- Douglas Crockford's simulated private attributes and private methods for an object, using closures.
- A great explanation of how closures can cause memory leaks in IE if you are not careful.
- MDN documentation on JavaScript Closures.
- The Beginner's Guide to JavaScript Closures.

Share Improve this answer Follow

edited Jul 16, 2023 at 17:04

community wiki
69 revs, 58 users 30%
Ben Aston

11 I've only been a frontend dev for six years, so I'm curious how common examples like `curriedAdd(2)(3)()` in your Functional Programming examples are besides when explaining closures or in coding interviews. I've done a lot of code reviews and have never come across it, but I've also

never worked with computer science MVPs like I assume FANG companies employ. – jsonp Mar 14, 2022 at 14:26

Undoubtedly what you've written is true, though I haven't verified it personally. However, I find it deeply disturbing. Could you do an edit to show how you would print 0, 1, and 2. Also, in both loops i is less than 3, so how could it possibly print 3? Does it go through and do it after the loop ended. NOTE: I started with a "C" then C++ and eventually C# background before I came to javascirpt. – JosephDoggie Apr 20, 2023 at 14:53

This would require an update as JS does provide a private field/method syntax with # prefix now. – Vic Stalkeur Feb 8 at 13:14

4145

Every function in JavaScript maintains a link to its outer lexical environment. A lexical environment is a map of all the names (eg. variables, parameters) within a scope, with their values.

So, whenever you see the `function` keyword, code inside that function has access to variables declared outside the function.

```
function foo(x) {
  var tmp = 3;

  function bar(y) {
    console.log(x + y + (++tmp)); // will log 16
  }

  bar(10);
}
```

```
foo(2);
```

Run code snippet   Expand snippet

This will log `16` because function `bar` closes over the parameter `x` and the variable `tmp`, both of which exist in the lexical environment of outer function `foo`.

Function `bar`, together with its link with the lexical environment of function `foo` is a closure.

A function doesn't have to *return* in order to create a closure. Simply by virtue of its declaration, every function closes over its enclosing lexical environment, forming a closure.

```
function foo(x) {
  var tmp = 3;

  return function (y) {
    console.log(x + y + (++tmp)); // will also log 1
  }
}

var bar = foo(2);
bar(10); // 16
bar(10); // 17
```

Run code snippet   Expand snippet

The above function will also log 16, because the code inside `bar` can still refer to argument `x` and variable `tmp`, even though they are no longer directly in scope.

However, since `tmp` is still hanging around inside `bar`'s closure, it is available to be incremented. It will be incremented each time you call `bar`.

The simplest example of a closure is this:

```
var a = 10;

function test() {
  console.log(a); // will output 10
  console.log(b); // will output 6
}
var b = 6;
test();
```

Run code snippet   Expand snippet

When a JavaScript function is invoked, a new execution context `ec` is created. Together with the function arguments and the target object, this execution context also receives a link to the lexical environment of the calling execution context, meaning the variables declared in the outer lexical environment (in the above example, both `a` and `b`) are available from `ec`.

Every function creates a closure because every function has a link to its outer lexical environment.

Note that variables *themselves* are visible from within a closure, *not* copies.

Share Improve this answer Follow

edited Mar 28, 2020 at 5:44

community wiki
41 revs, 31 users 31%
Ali

2616

FOREWORD: this answer was written when the question was:

> Like the old Albert said : "If you can't explain it to a six-year old, you really don't understand it yourself.”. Well I tried to explain JS closures to a 27 years old friend and completely failed.
>
> Can anybody consider that I am 6 and strangely interested in that subject ?

I'm pretty sure I was one of the only people that attempted to take the initial question literally. Since then, the question has mutated several times, so my answer may now seem incredibly silly & out of place. Hopefully the general idea of the story remains fun for some.

I'm a big fan of analogy and metaphor when explaining difficult concepts, so let me try my hand with a story.

Once upon a time:

There was a princess...

```
function princess() {
```

She lived in a wonderful world full of adventures. She met her Prince Charming, rode around her world on a unicorn, battled dragons, encountered talking animals, and many other fantastical things.

```
    var adventures = [];

    function princeCharming() { /* ... */ }

    var unicorn = { /* ... */ },
        dragons = [ /* ... */ ],
        squirrel = "Hello!";

    /* ... */
```

But she would always have to return back to her dull world of chores and grown-ups.

```
    return {
```

And she would often tell them of her latest amazing adventure as a princess.

```
        story: function() {
            return adventures[adventures.length - 1];
        }
```

```
    };
}
```

But all they would see is a little girl...

```
var littleGirl = princess();
```

...telling stories about magic and fantasy.

```
littleGirl.story();
```

And even though the grown-ups knew of real princesses, they would never believe in the unicorns or dragons because they could never see them. The grown-ups said that they only existed inside the little girl's imagination.

But we know the real truth; that the little girl with the princess inside...

...is really a princess with a little girl inside.

Share Improve this answer Follow

edited Nov 1, 2017 at 11:40

community wiki
11 revs, 5 users 58%
Jacob Swartwood

390 I love this explanation, truly. For those who read it and don't follow, the analogy is this: the princess() function is a complex scope containing private data. Outside the function, the private data can't be seen or accessed. The

princess keeps the unicorns, dragons, adventures etc. in her imagination (private data) and the grown-ups can't see them for themselves. BUT the princess's imagination is captured in the closure for the `story()` function, which is the only interface the `littleGirl` instance exposes into the world of magic. – Patrick M Feb 28, 2013 at 7:49

4 Having undefined values makes it more difficult to understand. Here is the true story jsfiddle.net/rjdx34k0/3 – Hugolpz Sep 2, 2020 at 19:13

3 And Prince Charming can add to her adventures, can kill all the dragons to save her from dangers like below: `function princeCharming { adventures.push('Honeymoon Trip', 'Skydiving', 'Visiting Somalia'); const pickADragonToKill = dragons.pop(); }` – Shiv Jan 13, 2021 at 5:15

1 A key point to my understanding was adding `console.log(littleGirl)` to @Hugolpz answer. When drilling into the littleGirl object with devtools I could not find princeCharming, unicorn, dragons, or squirrel anywhere. – Bradleo Dec 19, 2022 at 9:42

815

Taking the question seriously, we should find out what a typical 6-year-old is capable of cognitively, though admittedly, one who is interested in JavaScript is not so typical.

On Childhood Development: 5 to 7 Years it says:

> Your child will be able to follow two-step directions. For example, if you say to your child,

> "Go to the kitchen and get me a trash bag" they will be able to remember that direction.

We can use this example to explain closures, as follows:

> The kitchen is a closure that has a local variable, called `trashBags`. There is a function inside the kitchen called `getTrashBag` that gets one trash bag and returns it.

We can code this in JavaScript like this:

```
function makeKitchen() {
  var trashBags = ['A', 'B', 'C']; // only 3 at firs

  return {
    getTrashBag: function() {
      return trashBags.pop();
    }
  };
}

var kitchen = makeKitchen();

console.log(kitchen.getTrashBag()); // returns trash
console.log(kitchen.getTrashBag()); // returns trash
console.log(kitchen.getTrashBag()); // returns trash
```

Run code snippet   Expand snippet

Further points that explain why closures are interesting:

- Each time `makeKitchen()` is called, a new closure is created with its own separate `trashBags`.
- The `trashBags` variable is local to the inside of each kitchen and is not accessible outside, but the inner function on the `getTrashBag` property does have access to it.
- Every function call creates a closure, but there would be no need to keep the closure around unless an inner function, which has access to the inside of the closure, can be called from outside the closure. Returning the object with the `getTrashBag` function does that here.

Share Improve this answer Follow

edited Oct 10, 2018 at 17:50

community wiki
18 revs, 5 users 84%
dlaliberte

Is the closure literally created when `makeKitchen()` is called? I would say that the closure is created by the `return` statement that acquires reference to local variable `trashBags` while creating the function object to be returned. I think the closure is *owned* by the anonymous function referenced by property `getTrashBag` of the returned anonymous object. (I've been learning Rust lately and I think the ownership is a consept that helps to straighten things in other languages, too.) – Mikko Rantalainen Sep 30, 2022 at 7:49

@MikkoRantalainen, you are correct that the closure around the inner function is not necessarily created when the containing function is called, but it must have been created by the time the function returns, or any time the inner function is passed to some other context, which doesn't happen in this example. – dlaliberte Oct 4, 2022 at 14:59

Yes, the closure is created at the moment the anonymous *function is created* while defining the property `getTrashBag` of the anonymous object to be returned. – Mikko Rantalainen Oct 6, 2022 at 10:51

629

The Straw Man

I need to know how many times a button has been clicked and do something on every third click...

Fairly Obvious Solution

```
// Declare counter outside event handler's scope
var counter = 0;
var element = document.getElementById('button');

element.addEventListener("click", function() {
  // Increment outside counter
  counter++;

  if (counter === 3) {
    // Do something every third time
    console.log("Third time's the charm!");

    // Reset counter
    counter = 0;
  }
});
```

```
<button id="button">Click Me!</button>
```

Run code snippet   Expand snippet

Now this will work, but it does encroach into the outer scope by adding a variable, whose sole purpose is to keep track of the count. In some situations, this would be preferable as your outer application might need access to this information. But in this case, we are only changing every third click's behavior, so it is preferable to **enclose this functionality inside the event handler**.

Consider this option

```
var element = document.getElementById('button');

element.addEventListener("click", (function() {
  // init the count to 0
  var count = 0;

  return function(e) { // <- This function becomes t
    count++; //    and will retain access to the abo

    if (count === 3) {
      // Do something every third time
      console.log("Third time's the charm!");

      //Reset counter
      count = 0;
    }
  };
})());
```

```html
<button id="button">Click Me!</button>
```

Run code snippet   Expand snippet

Notice a few things here.

In the above example, I am using the closure behavior of JavaScript. **This behavior allows any function to have access to the scope in which it was created, indefinitely.** To practically apply this, I immediately invoke a function that returns another function, and because the function I'm returning has access to the internal count variable (because of the closure behavior explained above) this results in a private scope for usage by the resulting function... Not so simple? Let's dilute it down...

A simple one-line closure

```javascript
//          _______________________Immediately invoked
//         |
//         |        Scope retained for use      ___Ret
//         |       only by returned function   |    va
//         |             |            |         |
//         v             v            v         v
var func = (function() { var a = 'val'; return functio
```

All variables outside the returned function are available to the returned function, but they are not directly available to the returned function object...

```
func();  // Alerts "val"
func.a;  // Undefined
```

Get it? So in our primary example, the count variable is contained within the closure and always available to the event handler, so it retains its state from click to click.

Also, this private variable state is **fully** accessible, for both readings and assigning to its private scoped variables.

There you go; you're now fully encapsulating this behavior.

Full Blog Post (including jQuery considerations)

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
17 revs, 8 users 64%
jondavidjohn

545

Closures are hard to explain because they are used to make some behaviour work that everybody intuitively expects to work anyway. I find the best way to explain them (and the way that *I* learned what they do) is to imagine the situation without them:

```
const makePlus = function(x) {
    return function(y) { return x + y; };
}

const plus5 = makePlus(5);
console.log(plus5(3));
```

Run code snippet   Expand snippet

What would happen here if JavaScript *didn't* know closures? Just replace the call in the last line by its method body (which is basically what function calls do) and you get:

```
console.log(x + 3);
```

Now, where's the definition of `x`? We didn't define it in the current scope. The only solution is to let `plus5` *carry* its scope (or rather, its parent's scope) around. This way, `x` is well-defined and it is bound to the value 5.

Share Improve this answer Follow

edited Jun 30, 2020 at 8:16

community wiki
4 revs, 3 users 70%
Konrad Rudolph

Closure is just saving the outer lexical environment. if a function was created at some lexical environment that's mean its part of the memory of that lexical environment. when i will call the function a new execution context will be created and a new lexical environment will be created and it's outer reference will be point the the lexical environment which the function was created. – Nadav Shlush Sep 23, 2021 at 10:39

422

TLDR

A closure is a link between a function and its outer lexical (ie. as-written) environment, such that the identifiers (variables, parameters, function declarations etc) defined within that environment are visible from within the function, regardless of when or from where the function is invoked.

Details

In the terminology of the ECMAScript specification, a closure can be said to be implemented by the `[[Environment]]` reference of every function-object, which points to the lexical environment within which the function is defined.

When a function is invoked via the internal `[[Call]]` method, the `[[Environment]]` reference on the function-object is copied into the *outer environment reference* of the environment record of the newly-created execution context (stack frame).

In the following example, function `f` closes over the lexical environment of the global execution context:

```
function f() {}
```

In the following example, function `h` closes over the lexical environment of function `g`, which, in turn, closes over the lexical environment of the global execution context.

```
function g() {
    function h() {}
}
```

If an inner function is returned by an outer, then the outer lexical environment will persist after the outer function has returned. This is because the outer lexical environment needs to be available if the inner function is eventually invoked.

In the following example, function `j` closes over the lexical environment of function `i`, meaning that variable `x` is visible from inside function `j`, long after function `i` has completed execution:

```
function i() {
    var x = 'mochacchino'
    return function j() {
        console.log('Printing the value of x, from w
    }
}

const k = i()
setTimeout(k, 500) // invoke k (which is j) after 50
```

Run code snippet Expand snippet

In a closure, the variables in the outer lexical environment *themselves* are available, *not* copies.

```
function l() {
  var y = 'vanilla';

  return {
    setY: function(value) {
      y = value;
    },
    logY: function(value) {
      console.log('The value of y is: ', y);
    }
  }
}

const o = l()
o.logY() // The value of y is: vanilla
o.setY('chocolate')
o.logY() // The value of y is: chocolate
```

Run code snippet Expand snippet

The chain of lexical environments, linked between execution contexts via outer environment references, forms a *scope chain* and defines the identifiers visible from any given function.

Please note that in an attempt to improve clarity and accuracy, this answer has been substantially changed from the original.

Share Improve this answer Follow

edited Feb 17, 2020 at 19:56

community wiki
17 revs, 6 users 49%
Ben

see this thread for similar emphasis on ECMA spec – shea
Aug 11 at 19:49

408

OK, 6-year-old closures fan. Do you want to hear the simplest example of closure?

Let's imagine the next situation: a driver is sitting in a car. That car is inside a plane. Plane is in the airport. The ability of driver to access things outside his car, but inside the plane, even if that plane leaves an airport, is a closure. That's it. When you turn 27, look at the more detailed explanation or at the example below.

Here is how I can convert my plane story into the code.

```
var plane = function(defaultAirport) {

  var lastAirportLeft = defaultAirport;

  var car = {
    driver: {
      startAccessPlaneInfo: function() {
        setInterval(function() {
          console.log("Last airport was " + lastAirp
        }, 2000);
      }
    }
  };
  car.driver.startAccessPlaneInfo();

  return {
    leaveTheAirport: function(airPortName) {
      lastAirportLeft = airPortName;
    }
  }
  }("Boryspil International Airport");

  plane.leaveTheAirport("John F. Kennedy");
```

Run code snippet   Expand snippet

Share Improve this answer Follow

edited Oct 10, 2018 at 18:38

community wiki
8 revs, 6 users 46%
Max Tkachenko

385

This is an attempt to clear up several (possible) misunderstandings about closures that appear in some of the other answers.

- **A closure is not only created when you return an inner function.** In fact, the enclosing function *does not need to return at all* in order for its closure to be created. You might instead assign your inner function to a variable in an outer scope, or pass it as an argument to another function where it could be called immediately or any time later. Therefore, the closure of the enclosing function is probably created *as soon as the enclosing function is called* since any inner function has access to that closure whenever the inner function is called, before or after the enclosing function returns.

- **A closure does not reference a copy of the *old values* of variables in its scope.** The variables themselves are part of the closure, and so the value seen when accessing one of those variables is the latest value at the time it is accessed. This is why inner functions created inside of loops can be tricky, since each one has access to the same outer variables rather than grabbing a copy of the variables at the time the function is created or called.
- **The "variables" in a closure include any named functions** declared within the function. They also include arguments of the function. A closure also has access to its containing closure's variables, all the way up to the global scope.
- **Closures use memory, but they don't cause memory leaks** since JavaScript by itself cleans up its own circular structures that are not referenced. Internet Explorer memory leaks involving closures are created when it fails to disconnect DOM attribute values that reference closures, thus maintaining references to possibly circular structures.

Share Improve this answer edited May 5, 2016 at 15:00

Follow

community wiki
4 revs, 2 users 76%
dlaliberte

255

I wrote a blog post a while back explaining closures. Here's what I said about closures in terms of **why** you'd want one.

> Closures are a way to let a function have **persistent, private variables** - that is, variables that only one function knows about, where it can keep track of info from previous times that it was run.

In that sense, they let a function act a bit like an object with private attributes.

Example

```
var iplus1= (function ()
{
    var plusCount = 0;
    return function ()
    {
        return ++plusCount;
    }
})();
```

Here the outer self-invoking anonymous function run only once and sets `plusCount` variable to 0, and returns the inner function expression.

Whereas inner function has access to `plusCount` variable. Now every time we invoke the function `iplus1()`, the inner function increments the `plusCount` variable.

The important point to keep in mind is that **no other script on the page has access to `plusCount` variable and the only way to change the `plusCount` variable is through `iplus1` function.**

Read More for reference: So what are these closure thingys?

Share Improve this answer Follow

edited Apr 10, 2023 at 20:27

community wiki
4 revs, 3 users 54%
Nathan Long

240

The original question had a quote:

> If you can't explain it to a six-year old, you really don't understand it yourself.

This is how I'd try to explain it to an actual six-year-old:

You know how grown-ups can own a house, and they call it home? When a mom has a child, the child doesn't really own anything, right? But its parents own a house, so whenever someone asks "Where's your home?", the child can answer "that house!", and point to the house of its parents.

A "Closure" is the ability of the child to always (even if abroad) be able to refer to its home, even though it's really the parent's who own the house.

Share Improve this answer Follow

edited Sep 1, 2022 at 11:01

community wiki
5 revs, 3 users 75%
Magne

226

Closures are simple:

The following simple example covers all the main points of JavaScript closures.*

Here is a factory that produces calculators that can add and multiply:

```
function make_calculator() {
  var n = 0; // this calculator stores a single number
  return {
    add: function(a) {
      n += a;
      return n;
    },
    multiply: function(a) {
      n *= a;
      return n;
    }
  };
}

first_calculator = make_calculator();
second_calculator = make_calculator();

first_calculator.add(3); // returns 3
second_calculator.add(400); // returns 400

first_calculator.multiply(11); // returns 33
second_calculator.multiply(10); // returns 4000
```

The key point: Each call to `make_calculator` creates a new local variable `n`, which continues to be usable by that calculator's `add` and `multiply` functions long after `make_calculator` returns.

If you are familiar with stack frames, these calculators seem strange: How can they keep accessing `n` after `make_calculator` returns? The answer is to imagine that JavaScript doesn't use "stack frames", but instead uses "heap frames", which can persist after the function call that made them returns.

Inner functions like `add` and `multiply`, which access variables declared in an outer function**, are called *closures*.

That is pretty much all there is to closures.

* For example, it covers all the points in the "Closures for Dummies" article given in another answer, except example 6, which simply shows that variables can be used before they are declared, a nice fact to know but completely unrelated to closures. It also covers all the points in the accepted answer, except for the points (1) that functions copy their arguments into local variables (the named function arguments), and (2) that copying numbers creates a new number, but copying an object reference gives you another reference to the same object. These are also good to know but again completely unrelated to closures. It is also very similar to the example in this answer but a bit shorter and less abstract. It does not cover the point of this answer or this comment, which is that JavaScript makes it difficult to plug the *current* value of a loop variable into your inner function: The "plugging in" step can only be done with a helper function that encloses your inner function and is invoked on each loop iteration. (Strictly speaking, the inner function accesses the helper function's copy of the variable, rather than having anything plugged in.) Again, very useful when creating closures, but not part of what a closure is or how it works. There is additional confusion due to closures working differently in functional languages like ML, where variables are bound to values rather than to storage space, providing a constant stream of people who understand closures in a way (namely the "plugging in" way) that is simply incorrect for JavaScript, where variables are always bound to storage space, and never to values.

** Any outer function, if several are nested, or even in the global context, as this answer points out clearly.

Share Improve this answer Follow

edited May 23, 2017 at 12:10

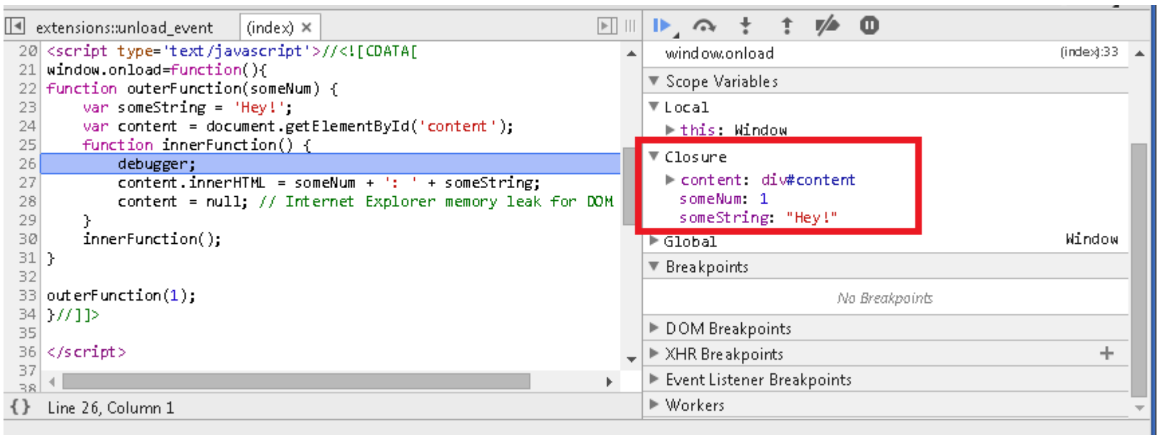
community wiki
7 revs, 2 users 81%
Matt

214

Can you explain closures to a 5-year-old?*

I still think Google's explanation works very well and is concise:

```
/*
*    When a function is defined in another function an
*    has access to the outer function's context even a
*    the outer function returns.
*
* An important concept to learn in JavaScript.
*/

function outerFunction(someNum) {
    var someString = 'Hey!';
    var content = document.getElementById('content');
    function innerFunction() {
        content.innerHTML = someNum + ': ' + someStrin
        content = null; // Internet Explorer memory le
    }
    innerFunction();
}

outerFunction(1);
```


extensions::unload_event (index)

```
<script type='text/javascript'>//<![CDATA[
window.onload=function(){
function outerFunction(someNum) {
    var someString = 'Hey!';
    var content = document.getElementById('content');
    function innerFunction() {
        debugger;
        content.innerHTML = someNum + ': ' + someString;
        content = null; // Internet Explorer memory leak for DOM
    }
    innerFunction();
}

outerFunction(1);
}//]]>

</script>
```

Line 26, Column 1

```
window.onload                          (index):33
▼ Scope Variables
▼ Local
  ▶ this: Window
▼ Closure
  ▶ content: div#content
    someNum: 1
    someString: "Hey!"
▶ Global                               Window
▼ Breakpoints
              No Breakpoints
▶ DOM Breakpoints
▶ XHR Breakpoints
▶ Event Listener Breakpoints
▶ Workers
```

*A C# question

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
7 revs, 3 users 70%
Chris S

190

I tend to learn better by GOOD/BAD comparisons. I like to see working code followed by non-working code that someone is likely to encounter. I put together a jsFiddle that does a comparison and tries to boil down the differences to the simplest explanations I could come up with.

Closures done right:

```
console.log('CLOSURES DONE RIGHT');

var arr = [];

function createClosure(n) {
    return function () {
        return 'n = ' + n;
    }
}

for (var index = 0; index < 10; index++) {
    arr[index] = createClosure(index);
}

for (var index of arr) {
    console.log(arr[index]());
}
```

- In the above code `createClosure(n)` is invoked in every iteration of the loop. Note that I named the variable `n` to highlight that it is a **new** variable created in a new function scope and is not the same variable as `index` which is bound to the outer scope.
- This creates a new scope and `n` is bound to that scope; this means we have 10 separate scopes, one for each iteration.
- `createClosure(n)` returns a function that returns the n within that scope.
- Within each scope `n` is bound to whatever value it had when `createClosure(n)` was invoked so the nested function that gets returned will always return the value of `n` that it had when `createClosure(n)` was invoked.

Closures done wrong:

```
console.log('CLOSURES DONE WRONG');
```

```
function createClosureArray() {
    var badArr = [];

    for (var index = 0; index < 10; index++) {
        badArr[index] = function () {
            return 'n = ' + index;
        };
    }
    return badArr;
}

var badArr = createClosureArray();

for (var index of badArr) {
    console.log(badArr[index]());
}
```

- In the above code the loop was moved within the `createClosureArray()` function and the function now just returns the completed array, which at first glance seems more intuitive.
- What might not be obvious is that since `createClosureArray()` is only invoked once only one scope is created for this function instead of one for every iteration of the loop.
- Within this function a variable named `index` is defined. The loop runs and adds functions to the array that return `index`. Note that `index` is defined within the `createClosureArray` function which only ever gets invoked one time.
- Because there was only one scope within the `createClosureArray()` function, `index` is only bound to a value within that scope. In other words, each

time the loop changes the value of `index`, it changes it for everything that references it within that scope.

- All of the functions added to the array return the SAME `index` variable from the parent scope where it was defined instead of 10 different ones from 10 different scopes like the first example. The end result is that all 10 functions return the same variable from the same scope.
- After the loop finished and `index` was done being modified the end value was 10, therefore every function added to the array returns the value of the single `index` variable which is now set to 10.

Result

> CLOSURES DONE RIGHT
> n = 0
> n = 1
> n = 2
> n = 3
> n = 4
> n = 5
> n = 6
> n = 7
> n = 8
> n = 9
>
> CLOSURES DONE WRONG
> n = 10

> n = 10
>
> n = 10
>
> n = 10
>
> n = 10
>
> n = 10
>
> n = 10
>
> n = 10
>
> n = 10
>
> n = 10

Share Improve this answer Follow

edited Oct 9, 2022 at 3:20

community wiki
7 revs, 3 users 72%
Chev

171

Wikipedia on closures:

> In computer science, a closure is a function together with a referencing environment for the nonlocal names (free variables) of that function.

Technically, in JavaScript, **every function is a closure**. It always has an access to variables defined in the surrounding scope.

Since **scope-defining construction in JavaScript is a function**, not a code block like in many other languages,

what we usually mean by *closure* in JavaScript is a **function working with nonlocal variables defined in already executed surrounding function**.

Closures are often used for creating functions with some hidden private data (but it's not always the case).

```
var db = (function() {
    // Create a hidden object, which will hold the dat
    // it's inaccessible from the outside.
    var data = {};

    // Make a function, which will provide some access
    return function(key, val) {
        if (val === undefined) { return data[key] } //
        else { return data[key] = val } // Set
    }
    // We are calling the anonymous surrounding functi
    // returning the above inner function, which is a
})();

db('x')    // -> undefined
db('x', 1) // Set x to 1
db('x')    // -> 1
// It's impossible to access the data object itself.
// We are able to get or set individual it.
```

ems

The example above is using an anonymous function, which was executed once. But it does not have to be. It can be named (e.g. `mkdb`) and executed later, generating a database function each time it is invoked. Every generated function will have its own hidden database object. Another usage example of closures is when we don't return a function, but an object containing multiple functions for different purposes, each of those function having access to the same data.

Share Improve this answer Follow

edited Dec 18, 2013 at 16:48

community wiki
6 revs, 3 users 78%
mykhal

142

> The children will never forget the secrets they have shared with their parents, even after their parents are gone. This is what closures are for functions.

The secrets for JavaScript functions are the private variables

```
var parent = function() {
 var name = "Mary"; // secret
}
```

Every time you call it, the local variable "name" is created and given the name "Mary". And every time the function exits the variable is lost and the name is forgotten.

As you may guess, because the variables are re-created every time the function is called, and nobody else will know them, there must be a secret place where they are stored. It could be called **Chamber of Secrets** or **stack** or **local scope** but it doesn't matter. We know they are there, somewhere, hidden in the memory.

But, in JavaScript, there is this very special thing that functions which are created inside other functions, can also know the local variables of their parents and keep them as long as they live.

```
var parent = function() {
  var name = "Mary";
  var child = function(childName) {
    // I can also see that "name" is "Mary"
  }
}
```

So, as long as we are in the parent -function, it can create one or more child functions which do share the secret variables from the secret place.

But the sad thing is, if the child is also a private variable of its parent function, it would also die when the parent ends, and the secrets would die with them.

So to live, the child has to leave before it's too late

```
var parent = function() {
  var name = "Mary";
  var child = function(childName) {
    return "My name is " + childName  +", child of " +
  }
  return child; // child leaves the parent ->
}
var child = parent(); // < - and here it is outside
```

And now, even though Mary is "no longer running", the memory of her is not lost and her child will always remember her name and other secrets they shared during their time together.

So, if you call the child "Alice", she will respond

```
child("Alice") => "My name is Alice, child of Mary"
```

That's all there is to tell.

Share Improve this answer Follow

edited Aug 15, 2022 at 9:23

community wiki
4 revs, 4 users 91%
Tero Tolonen

140

I put together an interactive JavaScript tutorial to explain how closures work. [What's a Closure?](#)

Here's one of the examples:

```
var create = function (x) {
    var f = function () {
        return x; // We can refer to x here!
    };
    return f;
};
// 'create' takes one argument, creates a function

var g = create(42);
// g is a function that takes no arguments now
```

```
var y = g();
// y is 42 here
```

Share Improve this answer Follow

edited Oct 25, 2014 at 22:38

community wiki
3 revs, 3 users 89%
Nathan Whitehead

112

I do not understand why the answers are so complex here.

Here is a closure:

```
var a = 42;

function b() { return a; }
```

Yes. You probably use that many times a day.

> There is no reason to believe closures are a complex design hack to address specific problems. No, closures are just about using a variable that comes from a higher scope **from the perspective of where the function was declared (not run)**.

> Now what it *allows* you to do can be more spectacular, see other answers.

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
2 revs, 2 users 97%
floribon

It’s true that this is a closure, but the answer doesn’t explain *anything* about how it works, or even *why* it’s a closure. In particular, the equivalent code would work e.g. in C, which doesn’t have closures. – Konrad Rudolph Sep 23, 2021 at 10:51

99

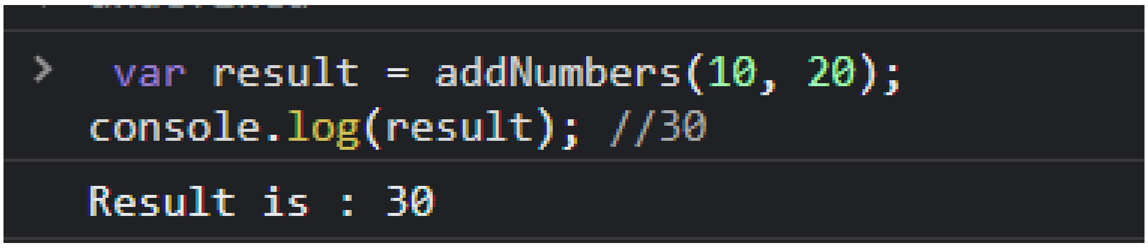
A closure is where an inner function has access to variables in its outer function. That's probably the simplest one-line explanation you can get for closures.

And the inner function has access not only to the outer function’s variables, but also to the outer function’s parameters as in the example below

Example

```
// Closure Example
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";

    // This inner function has access to the outer fun
```

```
parameters
    function add()
    {
        return returnValue + (firstNumber + secondNumb
    }
    return add();
}

var result = addNumbers(10, 20);
console.log(result); //30
```



```
> var result = addNumbers(10, 20);
  console.log(result); //30
  Result is : 30
```

Share Improve this answer Follow

edited Apr 10, 2023 at 20:01

community wiki
3 revs, 3 users 87%
Vinod Srivastav

98

Example for the first point by dlaliberte:

> A closure is not only created when you return an inner function. In fact, the enclosing function does not need to return at all. You might instead assign your inner function to a variable in an outer scope, or pass it as an argument to another function where it could be used immediately. Therefore, the closure of the

> enclosing function probably already exists at the time that enclosing function was called since any inner function has access to it as soon as it is called.

```
var i;
function foo(x) {
    var tmp = 3;
    i = function (y) {
        console.log(x + y + (++tmp));
    }
}
foo(2);
i(3);
```

Share Improve this answer Follow

edited Jan 16, 2016 at 2:39

community wiki
3 revs, 3 users 64%
someisaac

89

I know there are plenty of solutions already, but I guess that this small and simple script can be useful to demonstrate the concept:

```
// makeSequencer will return a "sequencer" function
var makeSequencer = function() {
    var _count = 0; // not accessible outside this fun
    var sequencer = function () {
        return _count++;
    }
    return sequencer;
}
```

```
var fnext = makeSequencer();
var v0 = fnext();     // v0 = 0;
var v1 = fnext();     // v1 = 1;
var vz = fnext._count // vz = undefined
```

Share Improve this answer Follow

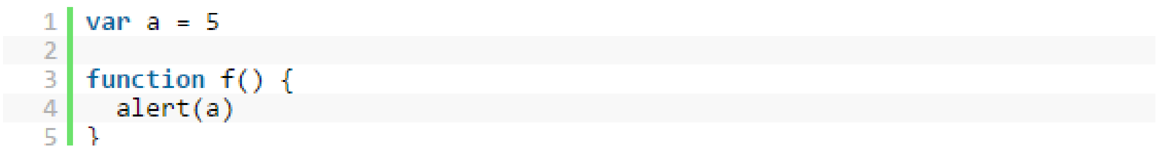
edited May 9, 2016 at 11:32

community wiki
3 revs, 2 users 90%
Gerardo Lima

86

The author of *Closures* has explained closures pretty well, explaining the reason why we need them and also explaining LexicalEnvironment which is necessary to understanding closures.
Here is the summary:

What if a variable is accessed, but it isn't local? Like here:



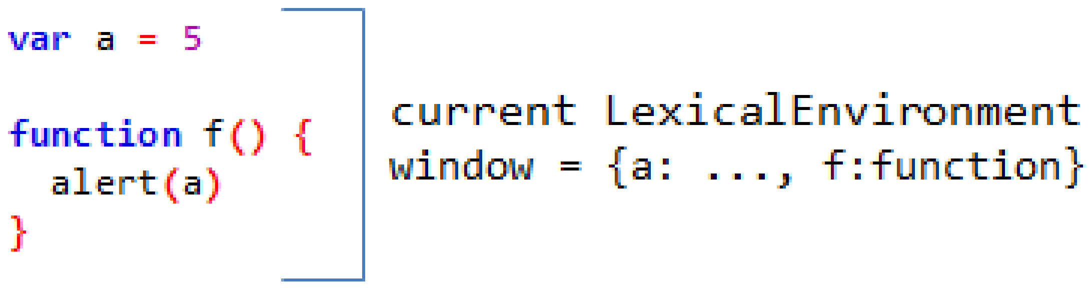
```
var a = 5

function f() {
  alert(a)
}
```

In this case, the interpreter finds the variable in the outer `LexicalEnvironment` object.

The process consists of two steps:

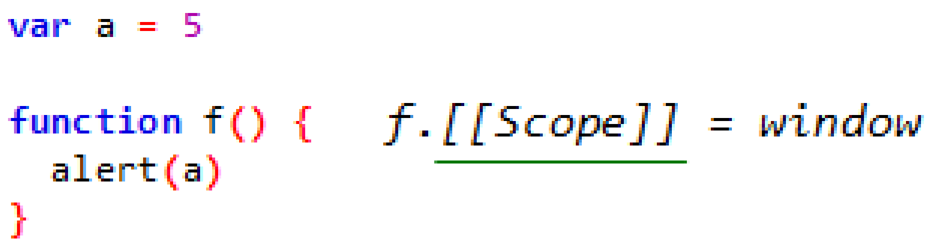
1. First, when a function f is created, it is not created in an empty space. There is a current LexicalEnvironment object. In the case above, it's

window (a is undefined at the time of function creation).



```
var a = 5

function f() {
  alert(a)
}
```

When a function is created, it gets a hidden property, named [[Scope]], which references the current LexicalEnvironment.

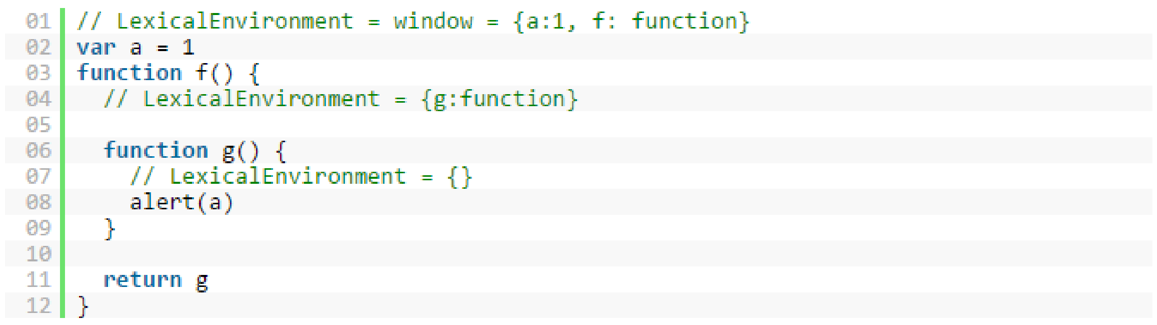


```
var a = 5

function f() {
  alert(a)
}
```

If a variable is read, but can not be found anywhere, an error is generated.

Nested functions

Functions can be nested one inside another, forming a chain of LexicalEnvironments which can also be called a scope chain.

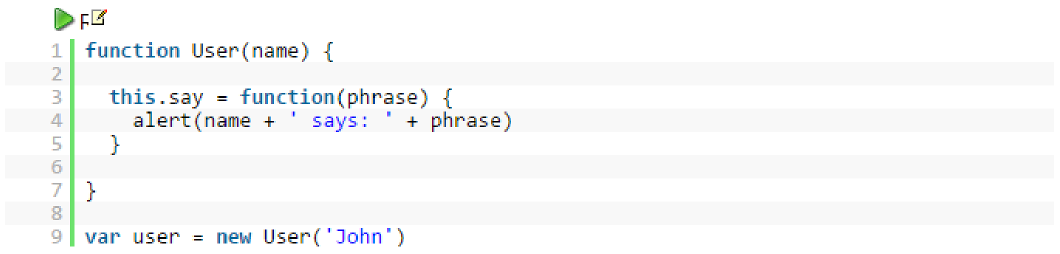


```
// LexicalEnvironment = window = {a:1, f: function}
var a = 1
function f() {
  // LexicalEnvironment = {g:function}

  function g() {
    // LexicalEnvironment = {}
    alert(a)
  }

  return g
}
```

So, function g has access to g, a and f.

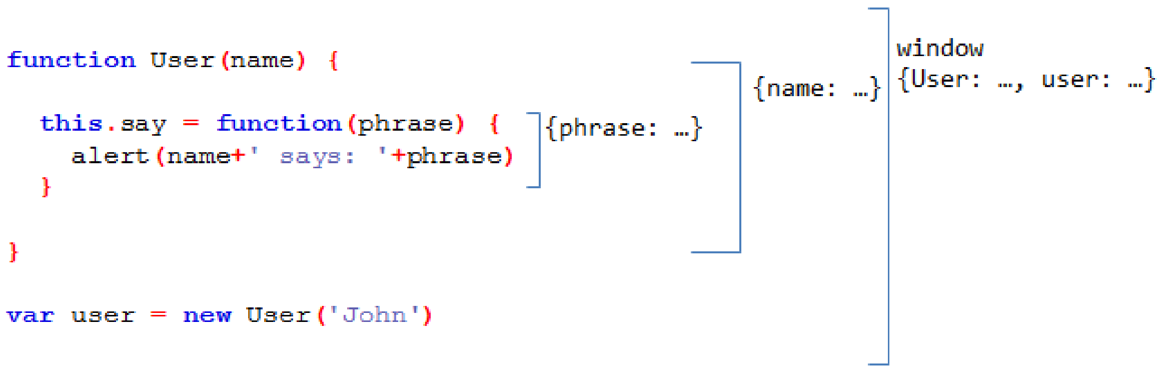
Closures

A nested function may continue to live after the outer function has finished:



```
function User(name) {

  this.say = function(phrase) {
    alert(name + ' says: ' + phrase)
  }

}

var user = new User('John')
```

Marking up LexicalEnvironments:



```
function User(name) {

  this.say = function(phrase) {
    alert(name+' says: '+phrase)
  }

}

var user = new User('John')
```

As we see, `this.say` is a property in the user object, so it continues to live after User completed.

And if you remember, when `this.say` is created, it (as every function) gets an internal reference `this.say.[[Scope]]` to the current LexicalEnvironment. So, the LexicalEnvironment of the current User execution stays in memory. All variables of User also are its properties, so they are also carefully kept, not junked as usually.

The whole point is to ensure that if the inner function wants to access an outer variable in the future, it is able to do so.

To summarize:

1. The inner function keeps a reference to the outer LexicalEnvironment.
2. The inner function may access variables from it any time even if the outer function is finished.
3. The browser keeps the LexicalEnvironment and all its properties (variables) in memory until there is an inner function which references it.

This is called a closure.

Share Improve this answer Follow

edited May 14, 2018 at 20:51

community wiki
7 revs, 2 users 81%
Arvand

85

You're having a sleep over and you invite Dan. You tell Dan to bring one XBox controller.

Dan invites Paul. Dan asks Paul to bring one controller. How many controllers were brought to the party?

```
function sleepOver(howManyControllersToBring) {

    var numberOfDansControllers = howManyControllersTo

    return function danInvitedPaul(numberOfPaulsContro
        var totalControllers = numberOfDansControllers
numberOfPaulsControllers;
        return totalControllers;
    }
}
```

```
var howManyControllersToBring = 1;

var inviteDan = sleepOver(howManyControllersToBring);

// The only reason Paul was invited is because Dan was
// So we set Paul's invitation = Dan's invitation.

var danInvitedPaul = inviteDan(howManyControllersToBri

alert("There were " + danInvitedPaul + " controllers b
```

Share Improve this answer Follow

edited Jul 20, 2011 at 15:16

community wiki
StewShack

79

JavaScript functions can access their:

1. Arguments
2. Locals (that is, their local variables and local functions)
3. Environment, which includes:
    - globals, including the DOM
    - anything in outer functions

If a function accesses its environment, then the function is a closure.

Note that outer functions are not required, though they do offer benefits I don't discuss here. By accessing data in its environment, a closure keeps that data alive. In the subcase of outer/inner functions, an outer function can create local data and eventually exit, and yet, if any inner function(s) survive after the outer function exits, then the inner function(s) keep the outer function's local data alive.

Example of a closure that uses the global environment:

Imagine that the Stack Overflow Vote-Up and Vote-Down button events are implemented as closures, voteUp_click and voteDown_click, that have access to external variables isVotedUp and isVotedDown, which are defined globally. (For simplicity's sake, I am referring to StackOverflow's Question Vote buttons, not the array of Answer Vote buttons.)

When the user clicks the VoteUp button, the voteUp_click function checks whether isVotedDown == true to determine whether to vote up or merely cancel a down vote. Function voteUp_click is a closure because it is accessing its environment.

```
var isVotedUp = false;
var isVotedDown = false;

function voteUp_click() {
  if (isVotedUp)
    return;
  else if (isVotedDown)
    SetDownVote(false);
  else
    SetUpVote(true);
}

function voteDown_click() {
  if (isVotedDown)
    return;
  else if (isVotedUp)
    SetUpVote(false);
  else
    SetDownVote(true);
}

function SetUpVote(status) {
  isVotedUp = status;
  // Do some CSS stuff to Vote-Up button
}

function SetDownVote(status) {
  isVotedDown = status;
  // Do some CSS stuff to Vote-Down button
}
```

All four of these functions are closures as they all access their environment.

Share Improve this answer Follow

edited Jun 8, 2016 at 22:16

community wiki
4 revs, 3 users 81%
John Pick

60

As a father of a 6-year-old, currently teaching young children (and a relative novice to coding with no formal education so corrections will be required), I think the lesson would stick best through hands-on play. If the 6-year-old is ready to understand what a closure is, then they are old enough to have a go themselves. I'd suggest pasting the code into jsfiddle.net, explaining a bit, and leaving them alone to concoct a unique song. The explanatory text below is probably more appropriate for a 10 year old.

```
function sing(person) {

    var firstPart = "There was " + person + " who swal

    var fly = function() {
        var creature = "a fly";
        var result = "Perhaps she'll die";
        alert(firstPart + creature + "\n" + result);
    };

    var spider = function() {
        var creature = "a spider";
        var result = "that wiggled and jiggled and tic
        alert(firstPart + creature + "\n" + result);
    };

    var bird = function() {
        var creature = "a bird";
        var result = "How absurd!";
        alert(firstPart + creature + "\n" + result);
    };

    var cat = function() {
        var creature = "a cat";
        var result = "Imagine That!";
        alert(firstPart + creature + "\n" + result);
    };

    fly();
    spider();
    bird();
    cat();
}

var person="an old lady";

sing(person);
```

INSTRUCTIONS

DATA: Data is a collection of facts. It can be numbers, words, measurements, observations or even just descriptions of things. You can't touch it, smell it or taste it. You can write it down, speak it and hear it. You could use it to *create* touch smell and taste using a computer. It can be made useful by a computer using code.

CODE: All the writing above is called *code*. It is written in JavaScript.

JAVASCRIPT: JavaScript is a language. Like English or French or Chinese are languages. There are lots of languages that are understood by computers and other electronic processors. For JavaScript to be understood by a computer it needs an interpreter. Imagine if a teacher who only speaks Russian comes to teach your class at school. When the teacher says "все садятся", the class would not understand. But luckily you have a Russian pupil in your class who tells everyone this means "everybody sit down" - so you all do. The class is like a computer and the Russian pupil is the interpreter. For JavaScript the most common interpreter is called a browser.

BROWSER: When you connect to the Internet on a computer, tablet or phone to visit a website, you use a browser. Examples you may know are Internet Explorer, Chrome, Firefox and Safari. The browser can understand JavaScript and tell the computer what it needs to do. The JavaScript instructions are called functions.

FUNCTION: A function in JavaScript is like a factory. It might be a little factory with only one machine inside. Or it might contain many other little factories, each with many machines doing different jobs. In a real life clothes factory you might have reams of cloth and bobbins of thread going in and T-shirts and jeans coming out. Our JavaScript factory only processes data, it can't sew, drill a hole or melt metal. In our JavaScript factory data goes in and data comes out.

All this data stuff sounds a bit boring, but it is really very cool; we might have a function that tells a robot what to make for dinner. Let's say I invite you and your friend to my house. You like chicken legs best, I like sausages, your friend always wants what you want and my friend does not eat meat.

I haven't got time to go shopping, so the function needs to know what we have in the fridge to make decisions. Each ingredient has a different cooking time and we want everything to be served hot by the robot at the same time. We need to provide the function with the data about what we like, the function could 'talk' to the fridge, and the function could control the robot.

A function normally has a name, parentheses and braces. Like this:

```
function cookMeal() {  /*  STUFF INSIDE THE FUNCTION
```

Note that `/*...*/` and `//` stop code being read by the browser.

NAME: You can call a function just about whatever word you want. The example "cookMeal" is typical in joining two words together and giving the second one a capital letter at the beginning - but this is not necessary. It can't have a space in it, and it can't be a number on its own.

PARENTHESES: "Parentheses" or `()` are the letter box on the JavaScript function factory's door or a post box in the street for sending packets of information to the factory. Sometimes the postbox might be marked *for example* `cookMeal(you, me, yourFriend, myFriend, fridge, dinnerTime)`, in which case you know what data you have to give it.

BRACES: "Braces" which look like this `{}` are the tinted windows of our factory. From inside the factory you can see out, but from the outside you can't see in.

THE LONG CODE EXAMPLE ABOVE

Our code begins with the word *function*, so we know that it is one! Then the name of the function *sing* - that's my own description of what the function is about. Then parentheses *()*. The parentheses are always there for a function. Sometimes they are empty, and sometimes they have something in. This one has a word in: `(person)`. After this there is a brace like this `{` . This marks the start of the function *sing()*. It has a partner which marks the end of *sing()* like this `}`

```
function sing(person) {  /* STUFF INSIDE THE FUNCTION
```

So this function might have something to do with singing, and might need some data about a person. It has instructions inside to do something with that data.

Now, after the function *sing()*, near the end of the code is the line

```
var person="an old lady";
```

VARIABLE: The letters *var* stand for "variable". A variable is like an envelope. On the outside this envelope is marked "person". On the inside it contains a slip of paper with the information our function needs, some letters and spaces joined together like a piece of string (it's called a string) that make a phrase reading "an old lady". Our envelope could contain other kinds of things like numbers (called integers), instructions (called functions), lists (called *arrays*). Because this variable is written outside of all the braces `{}`, and because you can see out through the tinted windows when you are inside the braces, this variable can be seen from anywhere in the code. We call this a 'global variable'.

GLOBAL VARIABLE: *person* is a global variable, meaning that if you change its value from "an old lady" to "a young man", the *person* will keep being a young man until you decide to change it again and that any other function in the code can see that it's a young man. Press the `F12` button or look at the Options settings to open the developer console of a browser and type "person" to see what this value is. Type `person="a young man"` to change it and then type "person" again to see that it has changed.

After this we have the line

```
sing(person);
```

This line is calling the function, as if it were calling a dog

> "Come on *sing*, Come and get *person*!"

When the browser has loaded the JavaScript code an reached this line, it will start the function. I put the line at the end to make sure that the browser has all the information it needs to run it.

Functions define actions - the main function is about singing. It contains a variable called *firstPart* which applies to the singing about the person that applies to each of the verses of the song: "There was " + person + " who swallowed". If you type *firstPart* into the console, you won't get an answer because the variable is locked up in a function - the browser can't see inside the tinted windows of the braces.

CLOSURES: The closures are the smaller functions that are inside the big *sing()* function. The little factories inside the big factory. They each have their own braces which mean that the variables inside them can't be seen from the outside. That's why the names of the variables (*creature* and *result*) can be repeated in the closures but with different values. If you type these variable names in the console window, you won't get its value because it's hidden by two layers of tinted windows.

The closures all know what the *sing()* function's variable called *firstPart* is, because they can see out from their tinted windows.

After the closures come the lines

```
fly();
spider();
bird();
cat();
```

The sing() function will call each of these functions in the order they are given. Then the sing() function's work will be done.

Share Improve this answer Follow

edited Jun 8, 2016 at 22:11

community wiki
5 revs, 3 users 77%
grateful

Okay, talking with a 6-year old child, I would possibly use following associations.

58

> Imagine - you are playing with your little brothers and sisters in the entire house, and you are moving around with your toys and brought some of them into your older brother's room. After a while your brother returned from the school and went to his room, and he locked inside it, so now you could not access toys left there anymore in a direct way. But you could knock the door and ask your brother for that toys. This is called toy's *closure*; your brother made it up for you, and he is now into outer *scope*.

Compare with a situation when a door was locked by draft and nobody inside (general function execution), and then some local fire occur and burn down the room (garbage collector:D), and then a new room was build and now you may leave another toys there (new function instance), but never get the same toys which were left in the first room instance.

For an advanced child I would put something like the following. It is not perfect, but it makes you feel about what it is:

```
function playingInBrothersRoom (withToys) {
  // We closure toys which we played in the brother's
and lock the door
  // your brother is supposed to be into the outer [[s
god you could communicate with him.
  var closureToys = withToys || [],
      returnToy, countIt, toy; // Just another closure
inner use.
```

```
  var brotherGivesToyBack = function (toy) {
    // New request. There is not yet closureToys on br
him a time.
    returnToy = null;
    if (toy && closureToys.length > 0) { // If we ask
brother is going to search for it.

      for ( countIt = closureToys.length; countIt; cou
        if (closureToys[countIt - 1] == toy) {
          returnToy = 'Take your ' + closureToys.splic
little boy!';
          break;
        }
      }
      returnToy = returnToy || 'Hey, I could not find
Look for it in another room.';
    }
    else if (closureToys.length > 0) { // Otherwise, j
he has in the room.
      returnToy = 'Behold! ' + closureToys.join(', ')
      closureToys = [];
    }
    else {
      returnToy = 'Hey, lil shrimp, I gave you everyth
    }
    console.log(returnToy);
  }
  return brotherGivesToyBack;
}
// You are playing in the house, including the brother
var toys = ['teddybear', 'car', 'jumpingrope'],
    askBrotherForClosuredToy = playingInBrothersRoom(t

// The door is locked, and the brother came from the s
cheat and take it out directly.
console.log(askBrotherForClosuredToy.closureToys); //

// But you could ask your brother politely, to give it
askBrotherForClosuredToy('teddybear'); // Hooray, here
askBrotherForClosuredToy('ball'); // The brother would
askBrotherForClosuredToy(); // The brother gives you a
askBrotherForClosuredToy(); // Nothing left in there
```

As you can see, the toys left in the room are still accessible via the brother and no matter if the room is locked. Here is [a jsbin]() to play around with it.

Share Improve this answer Follow

edited Oct 25, 2014 at 22:52

community wiki
9 revs, 2 users 88%
dmi3y

52

A function in JavaScript is not just a reference to a set of instructions (as in C language), but it also includes a hidden data structure which is composed of references to all nonlocal variables it uses (captured variables). Such two-piece functions are called closures. Every function in JavaScript can be considered a closure.

Closures are functions with a state. It is somewhat similar to "this" in the sense that "this" also provides state for a function but function and "this" are separate objects ("this" is just a fancy parameter, and the only way to bind it permanently to a function is to create a closure). While "this" and function always live separately, a function cannot be separated from its closure and the language provides no means to access captured variables.

Because all these external variables referenced by a lexically nested function are actually local variables in the chain of its lexically enclosing functions (global variables can be assumed to be local variables of some root function), and every single execution of a function creates new instances of its local variables, it follows that every execution of a function returning (or otherwise transferring it out, such as registering it as a callback) a nested function creates a new closure (with its own potentially unique set of referenced nonlocal variables which represent its execution context).

Also, it must be understood that local variables in JavaScript are created not on the stack frame, but on the heap and destroyed only when no one is referencing them. When a function returns, references to its local variables are decremented, but they can still be non-null if during the current execution they became part of a closure and are still referenced by its lexically nested functions (which can happen only if the references to these nested functions were returned or otherwise transferred to some external code).

An example:

```
function foo (initValue) {
   //This variable is not destroyed when the foo funct
   //It is 'captured' by the two nested functions retu
   var value = initValue;

   //Note that the two returned functions are created 
   //If the foo function is called again, it will retu
   //new functions referencing a different 'value' var
   return {
       getValue: function () { return value; },
       setValue: function (newValue) { value = newValu
   }
}
```

```
function bar () {
    //foo sets its local variable 'value' to 5 and ret
    //two functions still referencing that local varia
    var obj = foo(5);

    //Extracting functions just to show that no 'this'
    var getValue = obj.getValue;
    var setValue = obj.setValue;

    alert(getValue()); //Displays 5
    setValue(10);
    alert(getValue()); //Displays 10

    //At this point getValue and setValue functions ar
    //(in reality they are destroyed at the next itera
collector).
    //The local variable 'value' in the foo is no long
    //anything and is destroyed too.
}

bar();
```

Share Improve this answer Follow

edited May 5, 2016 at 16:04

community wiki
6 revs, 2 users 77%
srgstm

No one has said this but you might be correct. `Also, it must be understood that local variables in JavaScript are created not on the stack frame, but on the heap and destroyed only when no one is referencing them` So I am just telling that I heard primitive data types stored on stack and non-primitive on heap, do you mean all are stored on heap and stack is never used for any variables ? Just asking – Yusuf Nov 8, 2023 at 18:01

1 How variables are actually stored is implementation detail of the execution engine. You just shouldn't mentally picture them stored on the stack to get the understanding on how this really works. This is because captured variables (even primitive ones!) cannot be stored on the stack frame because when the function exits its stack frame is destroyed, but local variables (primitive or not) that get captured must live past that moment. It is of course possible for optimal performance that variables that aren't being captured be created on the stack. – srgstm Nov 24, 2023 at 10:53

you make sense and storing closured variables, even primitive ones, on heap will serve its purpose. but only closured ones are stored on heap or all variables ? that i could not find exact reference anywhere, atleast for v8 ? – Yusuf Nov 24, 2023 at 18:55

51

An answer for a six-year-old (assuming he knows what a function is and what a variable is, and what data is):

Functions can return data. One kind of data you can return from a function is another function. When that new function gets returned, all the variables and arguments used in the function that created it don't go away. Instead, that parent function "closes." In other words, nothing can look inside of it and see the variables it used except for the function it returned. That new function has a special ability to look back inside the function that created it and see the data inside of it.

```
function the_closure() {
  var x = 4;
  return function () {
    return x; // Here, we look back inside the_closure
    }
}

var myFn = the_closure();
myFn(); //=> 4
```

Another really simple way to explain it is in terms of scope:

Any time you create a smaller scope inside of a larger scope, the smaller scope will always be able to see what is in the larger scope.

Share Improve this answer Follow

edited Oct 25, 2014 at 23:02

community wiki
2 revs, 2 users 90%
Stupid Stupid

51

Perhaps a little beyond all but the most precocious of six-year-olds, but a few examples that helped make the concept of closure in JavaScript click for me.

A closure is a function that has access to another function's scope (its variables and functions). The easiest way to create a closure is with a function within a function; the reason being that in JavaScript a function always has access to its containing function’s scope.

```
function outerFunction() {
    var outerVar = "monkey";
    function innerFunction() {
        alert(outerVar);
    }

    innerFunction();
}

outerFunction();
```

Run code snippet   Expand snippet

ALERT: monkey

In the above example, outerFunction is called which in turn calls innerFunction. Note how outerVar is available to innerFunction, evidenced by its correctly alerting the value of outerVar.

Now consider the following:

```
function outerFunction() {
    var outerVar = "monkey";

    function innerFunction() {
        return outerVar;
    }

    return innerFunction;
}

var referenceToInnerFunction = outerFunction();
alert(referenceToInnerFunction());
```

Run code snippet   Expand snippet

ALERT: monkey

referenceToInnerFunction is set to outerFunction(), which simply returns a reference to innerFunction. When referenceToInnerFunction is called, it returns outerVar. Again, as above, this demonstrates that innerFunction has access to outerVar, a variable of outerFunction. Furthermore, it is interesting to note that it retains this access even after outerFunction has finished executing.

And here is where things get really interesting. If we were to get rid of outerFunction, say set it to null, you might think that referenceToInnerFunction would loose its access to the value of outerVar. But this is not the case.

```
function outerFunction() {
    var outerVar = "monkey";

    function innerFunction() {
        return outerVar;
    }

    return innerFunction;
}

var referenceToInnerFunction = outerFunction();
alert(referenceToInnerFunction());

outerFunction = null;
alert(referenceToInnerFunction());
```

Run code snippet   Expand snippet

ALERT: monkey ALERT: monkey

But how is this so? How can referenceToInnerFunction still know the value of outerVar now that outerFunction has been set to null?

The reason that referenceToInnerFunction can still access the value of outerVar is because when the closure was first created by placing innerFunction inside of outerFunction, innerFunction added a reference to outerFunction's scope (its variables and functions) to its scope chain. What this means is that innerFunction has a pointer or reference to all of outerFunction's variables, including outerVar. So even when outerFunction has finished executing, or even if it is deleted or set to null, the variables in its scope, like outerVar, stick around in memory because of the outstanding reference to them on the part of the innerFunction that has been returned to referenceToInnerFunction. To truly release outerVar and the rest of outerFunction's variables from memory you would have to get rid of this outstanding reference to them, say by setting referenceToInnerFunction to null as well.

//////////

Two other things about closures to note. First, the closure will always have access to the last values of its containing function.

```
function outerFunction() {
    var outerVar = "monkey";

    function innerFunction() {
        alert(outerVar);
    }

    outerVar = "gorilla";

    innerFunction();
}

outerFunction();
```

Run code snippet   Expand snippet

ALERT: gorilla

Second, when a closure is created, it retains a reference to all of its enclosing function’s variables and functions; it doesn’t get to pick and choose. And but so, closures should be used sparingly, or at least carefully, as they can be memory intensive; a lot of variables can be kept in memory long after a containing function has finished executing.

Share Improve this answer Follow

edited Apr 29, 2015 at 15:37

community wiki
2 revs, 2 users 99%
Michael Dziedzic

48

I'd simply point them to the Mozilla Closures page. It's the best, most **concise and simple explanation** of closure basics and practical usage that I've found. It is highly recommended to anyone learning JavaScript.

And yes, I'd even recommend it to a 6-year old -- if the 6-year old is learning about closures, then it's logical they're ready to comprehend the *concise and simple explanation* provided in the article.

edited Oct 25, 2014 at 22:54

community wiki
3 revs, 2 users 50%
mjmoody383