

by Angus Croll

JavaScript, JavaScript...

Understanding JavaScript Closures

In JavaScript, a closure is a function to which the variables of the surrounding context are bound by reference.

```
1  function getMeAClosure() {  
2      var canYouSeeMe = "here I am";  
3      return (function theClosure() {  
4          return {canYouSeeIt: canYouSeeMe ? "yes!": "no"};  
5      });  
6  }  
7  
8  var closure = getMeAClosure();  
9  closure().canYouSeeIt; // "yes!"
```

Every JavaScript function forms a closure on creation. In a moment I'll explain why and walk through the process by which closures are created. Then I'll address some common misconceptions and finish with some practical applications. But first a brief word from our sponsors: JavaScript closures are brought to you by and the ...

Lexical Scope

The word **lexical** pertains to words or language. Thus the function's physical placement within the written source code.

of a function is statically defined by the

Consider the following example:

```
1  var x = "global";  
2  
3  function outer() {  
4      var y = "outer";  
5  
6      function inner() {  
7          var x = "inner";  
8      }  
9  }
```

Function `inner` is physically surrounded by function `outer` which in turn is wrapped by the global context. We have formed a lexical hierarchy:

```
global  
  outer  
    inner
```

The outer lexical scope of any given function is defined by its ancestors in the lexical hierarchy. Accordingly, the outer lexical scope of function `inner` comprises the global object and function `outer`.

VariableEnvironment

The global object has an associated execution context. Additionally every invocation of a function establishes and enters a new execution context. The execution context is the dynamic counterpart to the static lexical scope. Each execution context defines a `VariableEnvironment` which is a repository for variables declared by that context. (ES 5 (<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>) 10.4, 10.5)

[Note in EcmaScript 3, the `VariableEnvironment` of a function was known as the `ActivationObject` – which is also the term I used in some older articles]

We could represent the `VariableEnvironment` with pseudo-code...

```

1  //variableEnvironment: {x: undefined, etc.};
2  var x = "global";
3  //variableEnvironment: {x: "global", etc.};
4
5  function outer() {
6      //variableEnvironment: {y: undefined};
7      var y = "outer";
8      //variableEnvironment: {y: "outer"};
9
10     function inner() {
11         //variableEnvironment: {x: undefined};
12         var x = "inner";
13         //variableEnvironment: {x: "inner"};
14     }
15 }
```

However, it turns out this is only part of the picture. Each `VariableEnvironment` will also inherit the `VariableEnvironment` of its lexical scope. [The hero enters (stage-left)....]

The `[[scope]]` property

When a given `VariableEnvironment` encounters a function definition in the code, a new function object is created with an internal property named `[[scope]]` (as in `Function.prototype`) which references the current `VariableEnvironment`. (ES 5 (<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>) 13.0-2)

Every function gets a `[[scope]]` property, and when the function is invoked the value of the scope property is assigned to the `[[scope]]` (or `scope`) property of its `VariableEnvironment`. (ES 5 (<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>) 10.4.3.5-7) In this way, each `VariableEnvironment` inherits from the `VariableEnvironment` of its lexical parent. This scope chaining runs the length of the lexical hierarchy starting from the global object.

Let's see how our pseudo-code looks now:

```

1  //VariableEnvironment: {x: undefined, etc.};
2  var x = "global";
3  //VariableEnvironment: {x: "global", etc.};
4
5  function outer() {
6      //VariableEnvironment: {y: undefined, outerLex: {x: "global", etc.}}
7      var y = "outer";
8      //VariableEnvironment: {y: "outer", outerLex: {x: "global", etc.}}
9
10     function inner() {
11         //VariableEnvironment: {x: undefined, outerLex: {y: "outer", etc.}}
12         var x = "inner";
13         //VariableEnvironment: {x: "inner", outerLex: {y: "outer", outerLex: {x: "global", etc.}}}
14     }
15 }

```

The `[[scope]]` property acts as a bridge between nested VariableEnvironments and enables the process by which outer variables are embedded by inner VariableEnvironments (and prioritized by lexical proximity). The `[[scope]]` property also enables closures, since without it the variables of an outer function would be dereferenced and garbage collected once the outer function returned.

So there we have it – closures are nothing but an unavoidable side-effect of lexical scoping 🤖

Dispelling the Myths

Now that we know how closures work, we can begin to address some of the more scurrilous rumors associated with them.

Myth 1. Closures are created only after an inner function has been returned

When the function is created, it is assigned a `[[scope]]` property which references the variables of the outer lexical scope and prevents them from being garbage collected. Therefore the closure is formed on function creation

There is no requirement that a function should be returned before it becomes a closure. Here's a closure that works without returning a function:

```

1  var callLater = function(fn, args, context) {
2      setTimeout(function(){fn.apply(context, args)}, 2000);
3  }
4
5  callLater(alert, ['hello']);

```

Myth 2. The values of outer variables get copied or “baked in” to the closure

As we've seen, the closure references variables not values.

```

1 //Bad Example
2 //Create an array of functions that add 1,2 and 3 respectively
3 var createAdders = function() {
4     var fns = [];
5     for (var i=1; i<4; i++) {
6         fns[i] = (function(n) {
7             return i+n;
8         });
9     }
10    return fns;
11 }
12
13 var adders = createAdders();
14 adders[1](7); //11 ??
15 adders[2](7); //11 ??
16 adders[3](7); //11 ??

```

All three adder functions point to the same variable `i`. By the time any of these functions is invoked, the value of `i` is 4.

One solution is to pass each argument via a self invoking function. Since every function invocation takes place in a unique execution context, we guarantee the uniqueness of the argument variable across successive invocations.

```

1 //Good Example
2 //Create an array of functions that add 1,2 and 3 respectively
3 var createAdders = function() {
4     var fns = [];
5     for (var i=1; i<4; i++) {
6         (function(i) {
7             fns[i] = (function(n) {
8                 return i+n;
9             });
10        })(i)
11    }
12    return fns;
13 }
14
15 var adders = createAdders();
16 adders[1](7); //8 (-:
17 adders[2](7); //9 (-:
18 adders[3](7); //10 (-:

```

Myth 3. Closures only apply to inner functions

Admittedly closures created by outer functions are not interesting because the `[[scope]]` property only references the global scope, which is universally visible in any case. Nevertheless it's important to note that the closure creation process is identical for every function, and every function creates a closure.

Myth 4. Closures only apply to anonymous functions

I've seen this claim in one too many articles. Enough said 🙄

Myth 5. Closures cause memory leaks

Closures do not of themselves create circular references. In our original example, function `inner` references outer variables via its `[[scope]]` property, but neither the referenced variables or function `outer` references function `inner` or its local variables.

Older versions of IE are notorious for memory leaks and these usually get blamed on closures. A typical culprit is a DOM element referenced by a function, while an attribute of that same DOM element references another object in the same lexical scope as the function. Between IE6 and IE8 these circular references have been mostly tamed.

Practical Applications

Function templates

Sometimes we want to define multiple versions of a function, each one conforming to a blueprint but modified by supplied arguments. For example, we can create a standard set of functions for converting units of measures:

```
1  function makeConverter(toUnit, factor, offset) {
2      offset = offset || 0;
3      return function(input) {
4          return (((offset+input)*factor).toFixed(2), toUnit)].join(" ");
5      }
6  }
7
8  var milesToKm = makeConverter('km', 1.60936);
9  var poundsToKg = makeConverter('kg', 0.45460);
10 var fahrenheitToCelsius = makeConverter('degrees C', 0.5556, -32);
11
12 milesToKm(10); // "16.09 km"
13 poundsToKg(2.5); // "1.14 kg"
14 fahrenheitToCelsius(98); // "36.67 degrees C"
```

If, like me, you're into functional abstraction the next logical step would be to curify (<https://javascriptweblog.wordpress.com/2010/04/05/curry-cooking-up-tastier-functions/>) this process (see below).

Functional JavaScript

Aside from the fact that JavaScript functions are first class objects, functional JavaScript's other best friend is closures.

The typical implementations of `bind`, `curry` (<https://javascriptweblog.wordpress.com/2010/04/05/curry-cooking-up-tastier-functions/>), `partial` (<https://javascriptweblog.wordpress.com/2010/05/17/partial-currys-flashy-cousin/>) and `compose` (<https://javascriptweblog.wordpress.com/2010/04/14/compose-functions-as-building-blocks/>) all rely on closures to provide the new function with a reference to the original function and arguments.

For example, here's `curry`:

```

1  Function.prototype.curry = function() {
2      if (arguments.length<1) {
3          return this; //nothing to curry with - return function
4      }
5      var __method = this;
6      var args = toArray(arguments);
7      return function() {
8          return __method.apply(this, args.concat([].slice.apply(null,
9      )
10 }

```

And here's our previous example re-done using curry

```

1  function converter(toUnit, factor, offset, input) {
2      offset = offset || 0;
3      return (((offset+input)*factor).toFixed(2), toUnit].join(" ");
4  }
5
6  var milesToKm = converter.curry('km',1.60936,undefined);
7  var poundsToKg = converter.curry('kg',0.45460,undefined);
8  var fahrenheitToCelsius = converter.curry('degrees C',0.5556, -32);
9
10 milesToKm(10); //"16.09 km"
11 poundsToKg(2.5); //"1.14 kg"
12 fahrenheitToCelsius(98); //"36.67 degrees C"

```

There are plenty of other nifty function modifiers that use closures. This little gem comes courtesy of Oliver Steele (<http://osteele.com/>)

```

1  /**
2   * Returns a function that takes an object, and returns the value of
3   */
4  var pluck = function(name) {
5      return function(object) {
6          return object[name];
7      }
8  }
9
10 var getLength = pluck('length');
11 getLength("SF Giants are going to the World Series!"); //40

```

The module pattern

This well known technique (<https://javascriptweblog.wordpress.com/2010/04/22/the-module-pattern-in-a-nutshell/>) uses a closure to maintain a private, exclusive reference to a variable of the outer scope. Here I'm using the module pattern to make a "guess the number" game. Note that in this example, the closure (`guess`) has exclusive access to the `secretNumber` variable, while the `responses` object references a copy of the variable's value at the time of creation.

```
1  var secretNumberGame = function() {
2      var secretNumber = 21;
3
4      return {
5          responses: {
6              true: "You are correct! Answer is " + secretNumber,
7              lower: "Too high!",
8              higher: "Too low!"
9          },
10
11         guess: function(guess) {
12             var key =
13                 (guess == secretNumber) ||
14                 (guess < secretNumber ? "higher": "lower");
15             alert(this.responses[key])
16         }
17     }
18 }
19
20 var game = secretNumberGame();
21 game.guess(45); //"Too high!"
22 game.guess(18); //"Too low!"
23 game.guess(21); //"You are correct! Answer is 21"
```

Wrap up

In programming terms, closures represent the height of grace and sophistication. They make code more compact, readable and beautiful and promote functional re-use. Knowing how and why closures work eliminates the uncertainty around their usage. I hope this article helps in that regard. Please feel free to comment with questions, thoughts or concerns.

Further Reading

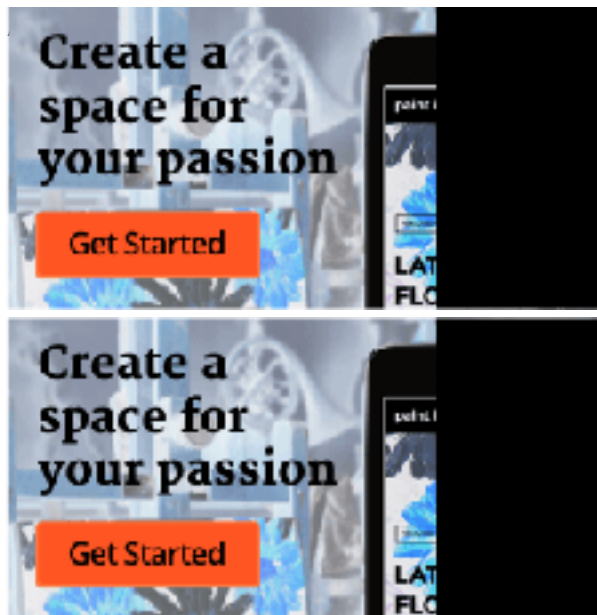
ECMA-262 5th Edition (<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>)

10.4 Creating the VariableEnvironment

10.4.3.5-7 Referencing the [[scope]] property in the VariableEnvironment

10.5 Populating the VariableEnvironment

13.0-2 Assigning the [[scope]] property when a function is created



JavaScript (<https://javascriptweblog.wordpress.com/category/javascript/>)

Angus Croll

October 25, 2010October 25, 2010

[closures](#) /
[functional](#) /
[functions](#)

50 thoughts on “Understanding JavaScript Closures”

Pingback: Understanding JavaScript Closures | JavaScript, JavaScript » Web Coding Unravelled

Pingback: Understanding JavaScript Closures | JavaScript, JavaScript – javascript - dowiedz się więcej!

Pingback: HTML Scripts Tips and Secrets » Blog Archive » Understanding JavaScript Closures | JavaScript, JavaScript

Rob Mills says:

October 25, 2010 at 10:33

Very nice explanation of closures. I am particularly fond of closures and use them regularly in my code.

Reply

Angus Croll says:

October 25, 2010 at 15:45

Thanks Rob – me too.

Reply

Pingback: Twitted by RobLL

Dmitry A. Soshnikov says:

October 25, 2010 at 12:17

>>The closure is created in two stages

Formally, a closure (a combination of a function code and the parent lexical environment, i.e. that exactly a function object is) is formed at the function's creation. The function then may never be called in future, but the parent environment is already saved, already, i.e. it.

I.e. when an inner function is returned, it already (theoretically, and practically in some implementations, e.g. Rhino) may access closed data without any execution:

```
// form a closure
var foo = (function () {
    var x = 10;
    return function () {};
})();

// and access a closed variable,
// without activation of "foo"
alert(foo.__parent__.x); // 10
```

And what is formed at activation of `foo` is already its own environment, which may access the parent (closed) environment. At this stage only a correct structure is formed (with saving reference in the own environment) in order to access the parent's data. But the is made at creation.

>>Admittedly closures created by outer functions are not interesting because the `[[scope]]` property only references the global scope, which is universally visible in any case.

Such global functions are interesting from the demonstrating of the again lexical scope (that exactly a requirement for closures). You may show, that a normal global function is also a closure, since it uses lexically saved (but, i.e. the environment of a) environment in the identifier resolution:

```
// global "x"
var x = 10;

// global function
function foo() {
  console.log(x);
}

(function (funArg) {

  // local "x"
  var x = 20;

  // there is no ambiguity,
  // because we use global "x",
  // which was statically saved in
  // [[Scope]] of the "foo" function,
  // but not the "x" of the caller's scope,
  // which activates the "funArg"

  funArg(); // 10, but not 20

})(foo); // pass "down" foo as a "funarg"
```

This example relates for “downward funarg problem”, when a function is passed a an argument (as a “funarg”). And this is the one of the reasons why closures where created — to distinguish a dynamic scope from the static (lexical) one.

>>The solution is to pass each argument via a self invoking function

You may want to notice, that this is just a one of the solutions. At least there are 4-5 alternative solutions: while, saving as properties of objects, Mozilla’s let, etc (some even without forming an additional object in the scope chain).

Dmitry.


P.S.: It’s good that you already use the new terminology of environments (actually, the general concept from the common theory).

In addition — Chapter 6. Closures.

Reply

Angus Croll says:

October 25, 2010 at 12:44

Nice input Dmitry. Also I am relieved that you are generally in agreement with my article  (Most of what I wrote about closures is contrary to the literature I have read – so your endorsement of my general approach is important to me)

Reply

Ashish Soni says:

April 21, 2014 at 03:00

@Angus Croll, When fn object is created `[[scope]]` property will be added in scope chain. Also when fn is invoked activation object with fn args, & local variables will be added in scope chain.

As per explanation of Nicholas Zakas , activation object has high priority as compared to lexical scope property for property lookup.

In above example `funArg()` is called in context of window object and it returns `window.x` value which is 10. That is quite obvious...

Correct me if I understood wrong ??

Pingback: HTML all you need to know» Blog Archive » Understanding JavaScript Closures | JavaScript, JavaScript

Chris says:

October 26, 2010 at 03:39

Superb article Angus, and very good input from Dmitry as well. Really helped solidify my understanding of closures – I didn't know about internal `[[Scope]]` property until now. Thank you!

Reply

joseanpg says:

October 26, 2010 at 07:50

As always, a nice article with a refreshing point of view. I would like highlight the following paragraph

The word lexical pertains to words or language. Thus the lexical scope of a function is statically defined by the function's physical placement within **the written source code**.

There are many places where this idea isn't so clearly expressed.

Reply

Angus Croll says:

October 26, 2010 at 09:33

@joseanpg @Chris

Thanks glad you liked it!

Reply

Pingback: Twitted by nickplekhanov

Mike says:

October 30, 2010 at 21:16

allright, about the closures, how this get resolved:

```

function Thunderbolt(intensity, startEnergy, rate) {
  //private
  var rateEnergyLoose = rate || 1,
      baseEnergy = startEnergy || 100,
      ready = false;
  function checkEnergy() {
    return baseEnergy > 0;
  }
  //this function intends to be privated and access public stuff
  //it needs to change a public value, but can only be call in a pri
  function decreaseEnergy(obj) {
    baseEnergy -= rateEnergyLoose * obj.strength;
  }

  //building the object that will be bind to the closure
  return {
    strength : intensity,
    fires : function (target) {
      if (!ready) { return;}
      if (target.hit) {
        target.hit(strength); //this are not visible here
      }
      ready = false;
      decreaseEnergy(this);
      chargeUp(); //this are not visible here
    },
    chargeUp: function (p) {
      var loadTime = p || 40;
      if (ready){
        return "already loaded";
      }

      if (!checkEnergy()) {
        return "it's empty";
      }

      function recharge () {
        if (loadTime <= 0) {
          ready = true;
          decreaseEnergy(...); //was a bad try
          return "loaded";
        };
        loadTime--;
        setTimeout(this,100);
      }

      setTimeout(recharge,100);
    }
  };
};

```

So, to resolve the private member access to public, i did the follow, but the closure wont let me work nicely in public access. How to workaround that?

```

function Thunderbolt(intensity, startEnergy, rate) {
  //private
  var rateEnergyLoose = rate || 1,
      baseEnergy = startEnergy || 100,
      ready = false,
      checkEnergy = function () {
        return baseEnergy > 0;
      },

function decreaseEnergy(obj) {
  baseEnergy = rateEnergyLoose * obj.strength;
}

  late = {
    strength : intensity,
    fires : function (target) {
      if (!ready) { return;}
      if (target.hit) {
        target.hit(strength); //still broken
      }
      ready = false;
      decreaseEnergy(this);
      //will work this out of this
chargeUp(); //this are not visible here
    },
    chargeUp: function (p) {
      var loadTime = p || 40;
      if (ready){
        return "already loaded";
      }

      if (!checkEnergy()) {
        return "it's empty";
      }

      function recharge () {
        if (loadTime <= 0) {
          ready = true;
          decreaseEnergy(); //now it has access
          return "loaded";
        };
        loadTime--;
        setTimeout(this,100);
      }

      setTimeout(recharge,100);
    }
  };
  function decreaseEnergy(obj) {
    baseEnergy -= rateEnergyLoose * late.strength; //it seems ok
  }

```

```
    return late;  
};
```

Sorry for the bad english

Reply

joseanpg says:

November 3, 2010 at 13:46

On the other hand, may perhaps be interesting to consider this version of ChargeUp. Your asynchronous loop has no body:

```
1  function chargeUp(loadTime) {  
2    if (ready){return "already loaded";}   
3    if (!checkEnergy()) {return "it's empty";}   
4    loadTime = loadTime || 40;  
5    setTimeout( function(){  
6      ready=true;  
7      decreaseEnergy();  
8      //return "loaded"; a return from the event loop  
9      // goes with Saito and  
10     },loadTime*100);  
11  }
```

Reply

joseanpg says:

November 3, 2010 at 13:32

I have tried homogenize the methods:

```

1  function Thunderbolt(intensity, startEnergy, rate) {
2
3      var rateEnergyLoose = rate || 1,
4          baseEnergy = startEnergy || 100,
5          ready = false;
6
7      function checkEnergy() {
8          return baseEnergy > 0;
9      }
10     function decreaseEnergy() { //Unique use of thisThunder
11         baseEnergy -= rateEnergyLoose * thisThunder.strength;
12     }
13     function fires(target) {
14         if (!ready) { return; }
15         if (target.hit) { target.hit(strength); }
16         ready = false;
17         decreaseEnergy();
18         chargeUp();
19     }
20
21     function chargeUp(loadTime) {
22         var loadTime = loadTime || 40;
23         if (ready) { return "already loaded"; }
24         if (!checkEnergy()) { return "it's empty"; }
25         function recharge () {
26             if (loadTime <= 0) {
27                 ready = true;
28                 decreaseEnergy();
29                 return "loaded";
30             };
31             loadTime--;
32             setTimeout(recharge, 100);
33         }
34         setTimeout(recharge, 100);
35     }
36
37     var thisThunder = {
38         strength : intensity, //Only used by private method decreaseEne
39         //Publish public methods
40         fires : fires,
41         chargeUp : chargeUp
42     };
43
44     return thisThunder;
45 };

```

Reply

Angus Croll says:

November 4, 2010 at 19:43

@Mike @Jose

I like the way Jose's returns thisThunder object referencing public functions. Clean.
More on this later...I need to study the code – (sorry been really busy)

Reply

joseanpg says:

November 5, 2010 at 09:32

There is an error in **fires**. Here is the correction


```

1  function fires(target) {
2      if (!ready) { return;}
3      if (target.hit) {target.hit(thisThunder.strength);}
4      ready = false;
5      decreaseEnergy();
6      chargeUp();
7  }

```

As **fires** also is a public method , the code below is also valid

```

1  function fires(target) {
2      if (!ready) { return;}
3      if (target.hit) {target.hit(this.strength);}
4      ready = false;
5      decreaseEnergy();
6      chargeUp();
7  }

```

Angus Croll says:

November 5, 2010 at 10:38

@joseanpg This is great – you tidied up Mike’s original code big time

chargeUp doesn’t appear to actually make base energy grow (as you pointed out)

Also why do we want recharge to fail if there is no energy left?

Finally I think recharge can be self invoking – doesn’t need a timeout to invoke

or just do this (I think its cleaner)

```

1  function chargeUp(loadTime) {
2      var loadTime = loadTime || 40;
3      if (ready) {
4          return "already loaded";}
5      }
6      if (!checkEnergy()) {
7          return "it's empty"; //but so what? (-:
8      }
9      recharge();
10     function recharge () {
11         if (loadTime <= 0) {
12             ready = true;
13             decreaseEnergy();
14             return "loaded";
15         };
16         loadTime--;
17         setTimeout(recharge,100);
18     }
19 }

```

Reply

Mike says:

November 6, 2010 at 09:52

@Angus Croll

the **chargeUp** method is to reload the thunderbolt. So, the last time that it fires, the **chargeUp** return that it’s empty.

@joseanpg

I prefer your first correction in the **chargeUp** method for the default assignment. It really doesn’t need the var statement and it’s better to come after the checks. I will stick with that. I really liked the way to

declare functions and make it public in the object literal. thanks.

I did this just to test the closures, and now i think it's a little senseless.

I really like the improvements in the code.

And just after I commented here, I found out that `this` is bound to the object in a object literal statement. But i didn't knew if it's the way it is specified.

Now i saw the joseanpg's code, i'm more confident that is the right way. (jsLint also does not complain about that).

i think the anonymous function is more elegant in the asynchronous loop.

But the Angus' loop actually does count right and i will stick with that.

joseanpg's and mine's, counts 41 secs instead of 40. (we could change the logic statement `(loadTime >= 0)` to `(loadTime > 1)`, but it will impossible to `chargeUp` instantly.)

Thank you all for the tips. It's very difficult to find people who knows javascript as a real programming Language. i'm very glad i found this blog. =]

My only doubt remaining is about the private method accessing the public attribute. I still think that may exist a better way to do this.

Thanks again. Sorry for the bad english.

Reply

Pingback: Understanding JavaScript Closures | Josh Hammock

Pingback: Links^2 – November 16th, 2010 « Devign | Web Development, JavaScript, CSS, MooTools | Elad Ossadon

Pingback: Closures en java... script | aninki.net

Pingback: Isolating your code with closures | Unobtrusive Javascript

javascriptweblog.wordpress.com says:

April 3, 2011 at 20:47

Understanding javascript closures.. Keen 😊

Reply

vapour says:

May 17, 2011 at 17:36

The curry function has a error

```
return __method.apply(this, args.concat([].slice.apply(arguments)));
```

Reply

Timothy Van Heest says:

July 19, 2013 at 07:01

I found this too. I was going crazy trying to figure out what I was missing. Aside from that, this is a very helpful article.

Reply

odytrice says:

February 8, 2012 at 07:17

Reblogged this on Tech Genius Blog and commented:

Understanding Closures in Javascript

Reply

Sarfraz Ahmed says:

February 11, 2012 at 02:27

This is simply the best coverage on closures, clears all sorts of confusions regarding them. Thanks for sharing.

Reply

Pingback: Understanding Javascript closures | Buffalo Billion

Shawn says:

July 18, 2012 at 18:13

nice article..

The guess game has an error though !

```
var key = (guess==secret) || (guess<secret ? "lower":"higher");
```

because the conditional operator returns the first value if the condition holds true and returns the value after the colon if the condition is false 🤔

Reply

milanchandna says:

September 10, 2012 at 03:56

Simply brilliant!

Thanks for explanation of internal working of functions in JS.

Reply

Pingback: 码工作坊 » Javascript 闭包

Sahil says:

November 19, 2012 at 13:13

Hi Angus,

Thank you for a detailed explanation on closures. I have just started with javascript and find your blog quite useful.

I have a doubt and was hoping you could help me with it.

In the GOOD Example in MYTH 2 you have used (i) at the end of the for loop. can you please explain why is (i) used there? what purpose does it serve?

Reply

Kristina says:

December 3, 2012 at 02:51

Very helpful. Thanks!

Reply

Pingback: Overriding jquery UI widget | Vaibhav Gupta

Pingback: Link Dump – January 5, 2013 Edition | Wern Ancheta

Pingback: Understanding JavaScript Closures and Scope | TechSlides

Pingback: JavaScript: Required Reading | James on JavaScript

Gabriel Gatzsche says:

April 17, 2013 at 00:58

Thank you very much for this great post! Particularly section myth2 saved my day! Thanks

Reply

Anonymous says:

July 11, 2013 at 19:51

```
var callLater = function(fn, args, context) {
```

```
// args is accessible via closure to the anonymous function that is called when timeout is triggered.
```

```
// context is undefined within the anonymous function so I don't know why it was included
```

```
// in argument list.  
// Original sample used the commented out setTimeout code and I don't know why they used  
// apply since context is undefined.  
// setTimeout(function(){fn.apply(context, args)}, 2000);  
// This seems to work just as well.  
setTimeout(function(){fn(args)}, 2000);  
}  
callLater(alert,['hello']);
```

Reply

Pingback: Day 85, closure | Shao

Anonymous says:

October 4, 2013 at 05:28

Very good article, however I prefer OOP approach far more than closures, which produce bunch of copies of the same function that differs only in parameter (outer, not direct argument). In my opinion it violates the notion of what function actually is. Here's how I would write a converter example:

```
function Converter(unit, factor, offset) {  
  this.unit = unit;  
  this.factor = factor;  
  this.offset = offset || 0;  
}  
  
Converter.prototype = {  
  calculate: function(v) {  
    return (((this.offset + v) * this.factor).toFixed(2), this.unit).join(' ');  
  }  
};  
  
var milesToKm = new Converter('km', 1.60936);  
var poundsToKg = new Converter('kg', 0.45460);  
var fahrenheitToCelsius = new Converter('degrees C', 0.5556, -32);  
  
console.log(milesToKm.calculate(10));  
console.log(poundsToKg.calculate(2.5));  
console.log(fahrenheitToCelsius.calculate(98));
```

I believe it's more readable, testable, reusable, maintainable. One may say: ok, you are not duplicating the functions, but instead create the same number of objects. That's correct, but I copy the data (which are different for each object) and not functionality (which is same for all objects).

Reply

arthwood says:

October 4, 2013 at 05:30

Very good article, however I prefer OOP approach far more than closures, which produce bunch of copies of the same function that differs only in parameter (outer, not direct argument). In my opinion it violates the notion of what function actually is. Here's how I would write a converter example:

```
function Converter(unit, factor, offset) {  
  this.unit = unit;  
  this.factor = factor;  
  this.offset = offset || 0;  
}
```

```
Converter.prototype = {  
  calculate: function(v) {  
    return (((this.offset + v) * this.factor).toFixed(2), this.unit).join(' ');  
  }  
};
```

```
var milesToKm = new Converter('km', 1.60936);  
var poundsToKg = new Converter('kg', 0.45460);  
var fahrenheitToCelsius = new Converter('degrees C', 0.5556, -32);
```

```
console.log(milesToKm.calculate(10));  
console.log(poundsToKg.calculate(2.5));  
console.log(fahrenheitToCelsius.calculate(98));
```

I believe it's more readable, testable, reusable, maintainable. One may say ok, you are not duplicating the functions but instead create the same number of objects. That's correct but I copy the data (which is different for each object) and not functionality (which is same for all objects).

Reply

Pingback: Useful Website Articles & Tutorials | Code Chewing

Pingback: Function Templates at Work | She Dev

Ashish Soni says:

April 21, 2014 at 03:03

Very detailed article...

Reply

Pingback: Exploring JavaScript Closures | Binarymist

kasun says:

June 6, 2014 at 10:10

hey in ur explained each execution context has variable environment. But in your sample code you write 2 environmentvariable

```
function outer() {
```

```
1=> //VariableEnvironment: {y: undefined, outerLex: {x: "global", etc.}};
```

```
var y = "outer";
```

```
1=>2 //VariableEnvironment: {y: "outer", outerLex: {x: "global", etc.}};. Why is that
```

Reply

Create a free website or blog at WordPress.com.