What is a 'Closure'?

Asked 16 years, 3 months ago Modified 2 years, 3 months ago

Viewed 202k times

558

I asked a question about Currying and closures were mentioned. What is a closure? How does it relate to currying?

functional-programming computer-science terminology glossary

Share

Improve this question

Follow

edited Oct 2, 2012 at 15:51

community wiki

11 revs, 7 users 50%

Ben

35 Now what exactly is the closure??? Some answers say, the closure is the function. Some say it is the stack. Some answers say, it is the "hidden" value. To my understanding, it is the function + enclosed variables. – Roland Nov 24, 2009 at 15:18

5 Explains what a closure is: stackoverflow.com/questions/4103750/… – snakehiss May 17, 2011 at 15:51

2 Also have a look at What is a closure? at softwareengineering.stackexchange – Felix K. Dec 29, 2018 at 18:53

There are great answers here. Yet, it is worth going over stackoverflow.com/a/36878651/1029599 , specifically if you're mathematically inclined. – KGhatak Mar 19, 2021 at 7:04

22 Answers

Sorted by: Highest score (default)

962

Variable scope

When you declare a local variable, that variable has a scope. Generally, local variables exist only within the block or function in which you declare them.

```
function() {
  var a = 1;
  console.log(a); // works
}
console.log(a); // fails
```

If I try to access a local variable, most languages will look for it in the current scope, then up through the parent scopes until they reach the root scope.

```
var a = 1;
function() {
  console.log(a); // works
}
console.log(a); // works
```

When a block or function is done with, its local variables are no longer needed and are usually blown out of memory.

This is how we normally expect things to work.

A closure is a persistent local variable scope

A closure is a persistent scope which holds on to local variables even after the code execution has moved out of that block. Languages which support closure (such as JavaScript, Swift, and Ruby) will allow you to keep a reference to a scope (including its parent scopes), even after the block in which those variables were declared has finished executing, provided you keep a reference to that block or function somewhere.

The scope object and all its local variables are tied to the function and will persist as long as that function persists.

This gives us function portability. We can expect any variables that were in scope when the function was first defined to still be in scope when we later call the function, even if we call the function in a completely different context.

For example

Here's a really simple example in JavaScript that illustrates the point:

```
outer = function() {
  var a = 1;
  var inner = function() {
    console.log(a);
  }
  return inner; // this returns a function
}

var fnc = outer(); // execute outer to get inner
fnc();
```

Here I have defined a function within a function. The inner function gains access to all the outer function's local variables, including `a`. The variable `a` is in scope for the inner function.

Normally when a function exits, all its local variables are blown away. However, if we return the inner function and assign it to a variable `fnc` so that it persists after `outer` has exited, **all of the variables that were in scope when `inner` was defined also persist**. The variable `a` has been closed over -- it is within a closure.

Note that the variable `a` is totally private to `fnc`. This is a way of creating private variables in a functional programming language such as JavaScript.

As you might be able to guess, when I call `fnc()` it prints the value of `a`, which is "1".

In a language without closure, the variable `a` would have been garbage collected and thrown away when the function `outer` exited. Calling fnc would have thrown an error because `a` no longer exists.

In JavaScript, the variable `a` persists because the variable scope is created when the function is first declared and persists for as long as the function continues to exist.

`a` belongs to the scope of `outer`. The scope of `inner` has a parent pointer to the scope of `outer`. `fnc` is a variable which points to `inner`. `a` persists as long as `fnc` persists. `a` is within the closure.

Further reading (watching)

I made a YouTube video looking at this code with some practical examples of usage.

Share Improve this answer Follow

edited Nov 22, 2021 at 9:57

community wiki
17 revs, 6 users 77%
superluminary

2 Could I have an example of how this works in a library like JQuery as stated in the 2nd to last paragraph? I didn't totally understand that. – DPM Nov 8, 2013 at 19:06

8 Hi Jubbat, yes, open up jquery.js and take a look at the first line. You'll see a function is opened. Now skip to the end, you'll see window.jQuery = window.$ = jQuery. Then the function is closed and self executed. You now have access to the $ function, which in turn has access to the other functions defined in the closure. Does that answer your question? – superluminary Nov 8, 2013 at 21:48

4 I've been reading my textbook on this topic for two days and couldn't really grasp what was going on. Reading your answer took 4 minutes and it makes perfect sense. – Andrew Jun 10, 2018 at 1:57

3 @BlissRage - one of the main purposes is for event handlers. When you set up your handler you have access to a bunch of local variables. Later though, when the handler is invoked, those variables may have changed or may no longer exist. Closure gives you a reliable runtime environment. – superluminary Jan 4, 2019 at 12:25

1 @user1063287 A closure is created whenever a function persists after its lexical scope has disappeared. Classic examples are callbacks. The function is created and passed to the DOM, or to a setTimeout, or to a Promise. All functions have a closure scope. That closure scope will be cleaned up by the garbage collector when the function is de-referenced. If the function is never de-referenced, that closure scope persists forever. I made a video that might help explain it, linked in the article. – superluminary Nov 28, 2021 at 0:27

132

I'll give an example (in JavaScript):

```
function makeCounter () {
  var count = 0;
  return function () {
    count += 1;
    return count;
  }
}

var x = makeCounter();
x(); returns 1
x(); returns 2
...etc...
```

What this function, `makeCounter`, does is it returns a function, which we've called `x`, that will count up by one each time it's called. Since we're not providing any parameters to `x`, it must somehow remember the count. It knows where to find it based on what's called lexical scoping - it must look to the spot where it's defined to find the value. This "hidden" value is what is called a closure.

Here is my currying example again:

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

var add3 = add(3);

add3(4); returns 7
```

What you can see is that when you call `add` with the parameter `a` (which is 3), that value is contained in the closure of the returned function that we're defining to be `add3`. That way, when we call `add3`, it knows where to find the `a` value to perform the addition.

Share Improve this answer edited Sep 15, 2022 at 1:36

Follow

Pang
10.1k ● 146 ● 85 ● 124

answered Aug 31, 2008 at 4:49

Kyle Cronin
79k ● 45 ● 151 ● 167

4 IDK, what language (probably F#) you've used in above language. Could please give above example in pseudocode? I'm having hard time to understand this. – user May 13, 2012 at 3:57

1 @crucifiedsoul It's Scheme. ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/intro.txt – Kyle Cronin May 13, 2012 at 13:15

4 @KyleCronin Great example, thanks. Q: Is it more correct to say "the hidden value is called a closure", or is "the function that hides the value is the closure"? Or "the process of hiding the value is the closure"? Thanks! – user550738 Feb 24, 2013 at 0:25

3 @RobertHume Good question. Semantically, the term "closure" is somewhat ambiguous. My personal definition is that the combination of both the hidden value and the enclosing function's use of it constitutes the closure. – Kyle Cronin Feb 24, 2013 at 0:36

I am from the Java background. That currying example is so damn lusciously beautiful. – Abdullah Mohammad Motiullah Jan 8, 2018 at 7:37

111

First of all, contrary to what most of the people here tell you, **closure is *not* a function**! So what *is* it?
It is a *set* of symbols defined in a function's "surrounding context" (known as its *environment*) which make it a

+50

CLOSED expression (that is, an expression in which every symbol is defined and has a value, so it can be evaluated).

For example, when you have a JavaScript function:

```
function closed(x) {
  return x + 3;
}
```

it is a *closed expression* because all the symbols occurring in it are defined in it (their meanings are clear), so you can evaluate it. In other words, it is *self-contained*.

But if you have a function like this:

```
function open(x) {
  return x*y + 3;
}
```

it is an *open expression* because there are symbols in it which have not been defined in it. Namely, `y`. When looking at this function, we can't tell what `y` is and what does it mean, we don't know its value, so we cannot evaluate this expression. I.e. we cannot call this function until we tell what `y` is supposed to mean in it. This `y` is called a *free variable*.

This `y` begs for a definition, but this definition is not part of the function – it is defined somewhere else, in its "surrounding context" (also known as the *environment*). At least that's what we hope for :P

For example, it could be defined globally:

```
var y = 7;

function open(x) {
  return x*y + 3;
}
```

Or it could be defined in a function which wraps it:

```
var global = 2;

function wrapper(y) {
  var w = "unused";

  return function(x) {
    return x*y + 3;
  }
}
```

The part of the environment which gives the free variables in an expression their meanings, is the *closure*. It is called this way, because it turns an *open* expression into a *closed* one, by supplying these missing definitions for all of its *free variables*, so that we could evaluate it.

In the example above, the inner function (which we didn't give a name because we didn't need it) is an *open expression* because the variable `y` in it is *free* – its definition is outside the function, in the function which wraps it. The *environment* for that anonymous function is the set of variables:

```
{
  global: 2,
  w: "unused",
  y: [whatever has been passed to that wrapper
function as its parameter `y`]
}
```

Now, the *closure* is that part of this environment which *closes* the inner function by supplying the definitions for all its *free variables*. In our case, the only free variable in the inner function was `y`, so the closure of that function is this subset of its environment:

```
{
  y: [whatever has been passed to that wrapper
function as its parameter `y`]
}
```

The other two symbols defined in the environment are *not* part of the *closure* of that function, because it doesn't require them to run. They are not needed to *close* it.

More on the theory behind that here: https://stackoverflow.com/a/36878651/434562

It's worth to note that in the example above, the wrapper function returns its inner function as a value. The moment we call this function can be remote in time from the moment the function has been defined (or created). In particular, its wrapping function is no longer running, and its parameters which has been on the call stack are no longer there :P This makes a problem, because the inner function needs `y` to be there when it is called! In other words, it requires the variables from its closure to somehow *outlive* the wrapper function and be there when needed. Therefore, the inner function has to make a *snapshot* of these variables which make its closure and store them somewhere safe for later use. (Somewhere outside the call stack.)

And this is why people often confuse the term *closure* to be that special type of function which can do such snapshots of the external variables they use, or the data structure used to store these variables for later. But I hope you understand now that they are *not* the closure itself – they're just ways to *implement* closures in a programming language, or language mechanisms which allows the variables from the function's closure to be there when needed. There's a lot of misconceptions around closures which (unnecessarily) make this subject much more confusing and complicated than it actually is.

Share Improve this answer Follow

edited Jan 20, 2021 at 6:38

community wiki
5 revs, 5 users 80%
SasQ

8 An analogy that might help beginners to this is a closure *ties up all the loose ends*, which is what a person does when they *seek closure* (or it *resolves* all necessary references, or ...). Well, it helped me to think of it that way :o) – Will Crawford Mar 20, 2018 at 18:35

5 I've read a lot of definitions of closure over the years, but I think this one is my favorite so far. I guess we all have our own way of mentally mapping concepts like this and this one very much jives with mine. – Jason S. Feb 7, 2020 at 16:21

4 I've seen whole lot of explanation from google, youtube, books, blogs, etc., and they all make sense and good ones, but I think this is the most logically clearest explanation. – starriet 차주녕 Oct 4, 2020 at 1:21

Is the global object a closure, because it encloses all nested variables and functions? perhaps not, because unlike the other 'outer' enclosing structures described, the global object can never be executed and 'finished'? – user1063287 Nov 24, 2021 at 23:19

65

Kyle's answer is pretty good. I think the only additional clarification is that the closure is basically a snapshot of the stack at the point that the lambda function is created. Then when the function is re-executed the stack is restored to that state before executing the function. Thus as Kyle mentions, that hidden value (`count`) is available when the lambda function executes.

Share Improve this answer Follow

edited May 23, 2017 at 12:26
Community Bot
1 ● 1

answered Aug 31, 2008 at 9:08
Ben Childs
4,570 ● 1 ● 22 ● 9

16 It's not just the stack -- it's the enclosing lexical scope(s) that are preserved, regardless of whether they're stored on the stack or the heap (or both). – Matt Fenwick Nov 6, 2012 at 13:16

33

A closure is a function that can reference state in another function. For example, in Python, this uses the closure "inner":

```
def outer (a):
    b = "variable in outer()"
    def inner (c):
        print a, b, c
    return inner

# Now the return value from outer() can be saved for l
func = outer ("test")
func (1) # prints "test variable in outer() 1
```

Share Improve this answer

Follow

edited Nov 16, 2019 at 2:08

Nick Parsons
50.3k ● 6 ● 56 ● 74

answered Aug 31, 2008 at 4:54

John Millikin
201k ● 41 ● 215 ● 227

23

To help facilitate understanding of closures it might be useful to examine how they might be implemented in a procedural language. This explanation will follow a simplistic implementation of closures in Scheme.

To start, I must introduce the concept of a namespace. When you enter a command into a Scheme interpreter, it must evaluate the various symbols in the expression and obtain their value. Example:

```
(define x 3)

(define y 4)

(+ x y) returns 7
```

The define expressions store the value 3 in the spot for x and the value 4 in the spot for y. Then when we call (+ x y), the interpreter looks up the values in the namespace and is able to perform the operation and return 7.

However, in Scheme there are expressions that allow you to temporarily override the value of a symbol. Here's an example:

```
(define x 3)

(define y 4)

(let ((x 5))
   (+ x y)) returns 9

x returns 3
```

What the let keyword does is introduces a new namespace with x as the value 5. You will notice that it's still able to see that y is 4, making the sum returned to be 9. You can also see that once the expression has ended x is back to being 3. In this sense, x has been temporarily masked by the local value.

Procedural and object-oriented languages have a similar concept. Whenever you declare a variable in a function that has the same name as a global variable you get the same effect.

How would we implement this? A simple way is with a linked list - the head contains the new value and the tail contains the old namespace. When you need to look up a symbol, you start at the head and work your way down the tail.

Now let's skip to the implementation of first-class functions for the moment. More or less, a function is a set of instructions to execute when the function is called culminating in the return value. When we read in a function, we can store these instructions behind the scenes and run them when the function is called.

```
(define x 3)

(define (plus-x y)
  (+ x y))

(let ((x 5))
  (plus-x 4)) returns ?
```

We define x to be 3 and plus-x to be its parameter, y, plus the value of x. Finally we call plus-x in an environment where x has been masked by a new x, this one valued 5. If we merely store the operation, (+ x y), for the function plus-x, since we're in the context of x being 5 the result returned would be 9. This is what's called dynamic scoping.

However, Scheme, Common Lisp, and many other languages have what's called lexical scoping - in addition to storing the operation (+ x y) we also store the namespace at that particular point. That way, when we're looking up the values we can see that x, in this context, is really 3. This is a closure.

```
(define x 3)

(define (plus-x y)
  (+ x y))

(let ((x 5))
  (plus-x 4)) returns 7
```

In summary, we can use a linked list to store the state of the namespace at the time of function definition, allowing us to access variables from enclosing scopes, as well as providing us the ability to locally mask a variable without affecting the rest of the program.

Share Improve this answer Follow

edited Jan 23, 2011 at 0:21

answered Aug 31, 2008 at 16:30

Kyle Cronin

79k ● 45 ● 151 ● 167

okay, thanks to your answer, I think that I finally have some idea what closure is about. But there is one big question: "we can use a linked list to store the state of the namespace at the time of function definition, allowing us to access variables that otherwise would no longer be in scope." `Why do we want to access variables that are out of scope? when we say let x = 5, we want x to be 5 and not 3. What is happening?` – Lazer May 23, 2010 at 11:28

@Laser: Sorry, that sentence didn't make much sense, so I updated it. I hope it makes more sense now. Also, don't think of the linked list as an implementation detail (as it's very inefficient) but as a simple way of conceptualize how it could be done. – Kyle Cronin May 23, 2010 at 15:28

13

Functions containing no free variables are called pure functions.

Functions containing one or more free variables are called closures.

```
var pure = function pure(x){
  return x
  // only own environment is used
}

var foo = "bar"

var closure = function closure(){
  return foo
  // foo is a free variable from the outer environment
}
```

src: https://leanpub.com/javascriptallongesix/read#leanpub-auto-if-functions-without-free-variables-are-pure-are-closures-impure

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
2 revs, 2 users 76%
soundyogi

1 Why is this minused? It is actually much more "on the right track" with that distinction into free variables and bound variables, and pure/closed functions and impure/open functions, than most of the other clueless answers in here :P (discounting for confusing closures with functions being closed). – SasQ Apr 27, 2016 at 2:43

I have *no* Idea, really. This is why StackOverflow sucks. Just look at the source of my Answer. Who could argue with that? – soundyogi Apr 27, 2016 at 13:22

SO doesn't suck and I never heard of the term "free variable" – Kai May 20, 2020 at 13:44

It's hard to talk about closures without mentioning free variables. Just look them up. Standard CS terminology. – ComDubh May 20, 2020 at 23:37

1 "Functions containing one or more free variables are called closures" is not a correct definition though -- closures are always first-class objects. – ComDubh May 20, 2020 at 23:39

tl;dr

9

A closure is a function and its scope assigned to (or used as) a variable. Thus, the name closure: the scope and the function is enclosed and used just like any other entity.

In depth Wikipedia style explanation

According to Wikipedia, a closure is:

> Techniques for implementing lexically scoped name binding in languages with first-class functions.

What does that mean? Lets look into some definitions.

I will explain closures and other related definitions by using this example:

```
function startAt(x) {
    return function (y) {
        return x + y;
    }
}

var closure1 = startAt(1);
var closure2 = startAt(5);

console.log(closure1(3)); // 4 (x == 1, y == 3)
console.log(closure2(3)); // 8 (x == 5, y == 3)
```

Run code snippet   Expand snippet

First-class functions

Basically that means **we can use functions just like any other entity**. We can modify them, pass them as arguments, return them from functions or assign them for variables. Technically speaking, they are first-class citizens, hence the name: first-class functions.

In the example above, `startAt` returns an (anonymous) function which function get assigned to `closure1` and `closure2`. So as you see JavaScript treats functions just like any other entities (first-class citizens).

Name binding

Name binding is about finding out **what data a variable** (identifier) **references**. The scope is really important here, as that is the thing that will determine how a binding is resolved.

In the example above:

- In the inner anonymous function's scope, `y` is bound to `3`.
- In `startAt`'s scope, `x` is bound to `1` or `5` (depending on the closure).

Inside the anonymous function's scope, `x` is not bound to any value, so it needs to be resolved in an upper (`startAt`'s) scope.

Lexical scoping

As Wikipedia says, the scope:

> Is the region of a computer program where the binding is valid: **where the name can be used to refer to the entity**.

There are two techniques:

- Lexical (static) scoping: A variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on.
- Dynamic scoping: Calling function is searched, then the function which called that calling function, and so on, progressing up the call stack.

For more explanation, check out this question and take a look at Wikipedia.

In the example above, we can see that JavaScript is lexically scoped, because when `x` is resolved, the binding is searched in the upper (`startAt`'s) scope, based on the source code (the anonymous function that looks for x is defined inside `startAt`) and not based on the call stack, the way (the scope where) the function was called.

Wrapping (closuring) up

In our example, when we call `startAt`, it will return a (first-class) function that will be assigned to `closure1` and `closure2` thus a closure is created, because the passed variables `1` and `5` will be saved within `startAt`'s scope, that will be enclosed with the returned anonymous function. When we call this anonymous function via `closure1` and `closure2` with the same argument (`3`), the value of `y` will be found immediately (as that is the parameter of that function), but `x` is not bound in the scope of the anonymous function, so the resolution continues in the (lexically) upper function scope (that was saved in the closure) where `x` is found to be bound to either `1` or `5`. Now we know everything for the summation so the result can be returned, then printed.

Now you should understand closures and how they behave, which is a fundamental part of JavaScript.

Currying

Oh, and you also learned what currying is about: you use functions (closures) to pass each argument of an operation instead of using one functions with multiple parameters.

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
totymedli

9

Here's a real world example of why Closures kick ass... This is straight out of my Javascript code. Let me illustrate.

```
Function.prototype.delay = function(ms /*[, arg...]*/)
  var fn = this,
      args = Array.prototype.slice.call(arguments, 1);

  return window.setTimeout(function() {
      return fn.apply(fn, args);
  }, ms);
};
```

And here's how you would use it:

```
var startPlayback = function(track) {
  Player.play(track);
};
startPlayback(someTrack);
```

Now imagine you want the playback to start delayed, like for example 5 seconds later after this code snippet runs. Well that's easy with `delay` and it's closure:

```
startPlayback.delay(5000, someTrack);
// Keep going, do other things
```

When you call `delay` with `5000`ms, the first snippet runs, and stores the passed in arguments in it's closure. Then 5 seconds later, when the `setTimeout` callback happens, the closure still maintains those variables, so it can call the original function with the original parameters.
This is a type of currying, or function decoration.

Without closures, you would have to somehow maintain those variables state outside the function, thus littering code outside the function with something that logically belongs inside it. Using closures can greatly improve the quality and readability of your code.

Share Improve this answer Follow

edited Nov 16, 2019 at 2:11

community wiki
2 revs, 2 users 73%
adamJLev

1 It should be noted that extending language or host objects is generally considered a bad thing as they are part of the global namespace – Jon Cooke May 29, 2014 at 3:10

5

Closure is a feature in JavaScript where a function has access to its own scope variables, access to the outer function variables and access to the global variables.

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

The inner function can access the variables defined in its own scope, the outer function’s scope, and the global scope. And the outer function can access the variable defined in its own scope and the global scope.

Example of Closure:

```
var globalValue = 5;

function functOuter() {
  var outerFunctionValue = 10;

  //Inner function has access to the outer function va
  //and the global variables
  function functInner() {
    var innerFunctionValue = 5;
    alert(globalValue + outerFunctionValue + innerFunc
  }
  functInner();
}
functOuter();
```

Output will be 20 which sum of its inner function own variable, outer function variable and global variable value.

Share Improve this answer Follow

edited Nov 16, 2019 at 2:15

community wiki
3 revs, 2 users 74%
rahul sharma

4

In a normal situation, variables are bound by scoping rule: Local variables work only within the defined function. Closure is a way of breaking this rule temporarily for convenience.

```
def n_times(a_thing)
  return lambda{|n| a_thing * n}
end
```

in the above code, `lambda(|n| a_thing * n}` is the closure because `a_thing` is referred by the lambda (an anonymous function creator).

Now, if you put the resulting anonymous function in a function variable.

```
foo = n_times(4)
```

foo will break the normal scoping rule and start using 4 internally.

```
foo.call(3)
```

returns 12.

Share Improve this answer

Follow

answered Aug 31, 2008 at 5:31

Eugene Yokota

95.5k ● 45 ● 217 ● 320

2

In short, function pointer is just a pointer to a location in the program code base (like program counter). Whereas **Closure = Function pointer + Stack frame**.

.

Share Improve this answer Follow

answered Sep 7, 2013 at 13:50

community wiki
RoboAlex

2

Closures provide JavaScript with state.

State in programming simply means remembering things.

Example

```
var a = 0;

a = a + 1; // => 1
a = a + 1; // => 2
a = a + 1; // => 3
```

In the case above, state is stored in the variable "a". We follow by adding 1 to "a" several times. We can only do that because we are able to "remember" the value. The state holder, "a", holds that value in memory.

Often, in programming languages, you want to keep track of things, remember information and access it at a later time.

This, **in other languages**, is commonly accomplished through the use of classes. A class, just like variables, keeps track of its state. And instances of that class, in turns, also have state within them. State simply means information that you can store and retrieve later.

Example

```
class Bread {
  constructor (weight) {
    this.weight = weight;
  }

  render () {
    return `My weight is ${this.weight}!`;
  }
}
```

How can we access "weight" from within the "render" method? Well, thanks to state. Each instance of the class Bread can render its own weight by reading it from the "state", a place in memory where we could store that information.

Now, **JavaScript is a very unique language** which historically does not have classes (it now does, but under the hood there's only functions and variables) so Closures provide a way for JavaScript to remember things and access them later.

Example

```
var n = 0;
var count = function () {
  n = n + 1;
  return n;
};

count(); // # 1
count(); // # 2
count(); // # 3
```

The example above achieved the goal of "keeping state" with a variable. This is great! However, this has the disadvantage that the variable (the "state" holder) is now exposed. We can do better. We can use Closures.

Example

```
var countGenerator = function () {
  var n = 0;
  var count = function () {
    n = n + 1;
    return n;
  };

  return count;
};

var count = countGenerator();
count(); // # 1
count(); // # 2
count(); // # 3
```

This is fantastic.

Now our "count" function can count. It is only able to do so because it can "hold" state. The state in this case is the variable "n". This variable is now closed. Closed in time and space. In time because you won't ever be able to recover it, change it, assign it a value or interact directly with it. In space because it's geographically nested within the "countGenerator" function.

Why is this fantastic? Because without involving any other sophisticated and complicated tool (e.g. classes, methods, instances, etc) we are able to 1. conceal 2. control from a distance

We conceal the state, the variable "n", which makes it a private variable! We also have created an API that can control this variable in a pre-defined way. In particular, we can call the API like so "count()" and that adds 1 to "n" from a "distance". In no way, shape or form anyone will ever be able to access "n" except through the API.

JavaScript is truly amazing in its simplicity.

Closures are a big part of why this is.

Share Improve this answer Follow

answered Sep 19, 2019 at 14:44

community wiki
Claudio

0

Here is another real life example, and using a scripting language popular in games - Lua. I needed to slightly change the way a library function worked to avoid a problem with stdin not being available.

```
local old_dofile = dofile

function dofile( filename )
  if filename == nil then
    error( 'Can not use default of stdin.' )
  end

  old_dofile( filename )
end
```

The value of old_dofile disappears when this block of code finishes it's scope (because it's local), however the value has been enclosed in a closure, so the new redefined dofile function CAN access it, or rather a copy stored along with the function as an 'upvalue'.

Share Improve this answer Follow

answered May 12, 2011 at 5:57

community wiki
Nigel Atkinson

0

From Lua.org:

> When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is called lexical scoping. Although that may sound obvious, it is not. Lexical scoping, plus first-class functions, is a powerful concept in a programming language, but few languages support that concept.

Share Improve this answer Follow

answered Dec 30, 2015 at 23:06

community wiki
user5731811

0

If you are from the Java world, you can compare a closure with a member function of a class. Look at this example

```
var f=function(){
  var a=7;
  var g=function(){
    return a;
  }
  return g;
}
```

The function `g` is a closure: `g` closes `a` in. So `g` can be compared with a member function, `a` can be compared with a class field, and the function `f` with a class.

Share Improve this answer Follow

edited Apr 2, 2016 at 19:35

community wiki
3 revs
ericj

0

Closures Whenever we have a function defined inside another function, the inner function has access to the variables declared in the outer function. Closures are best explained with examples. In Listing 2-18, you can see that the inner function has access to a variable (variableInOuterFunction) from the outer scope. The variables in the outer function have been closed by (or bound in) the inner function. Hence the term closure. The concept in itself is simple enough and fairly intuitive.

```
Listing 2-18:
    function outerFunction(arg) {
     var variableInOuterFunction = arg;

     function bar() {
             console.log(variableInOuterFunction);
// Access a variable from the outer scope
     }
     // Call the local function to demonstrate
that it has access to arg
     bar();
    }
    outerFunction('hello closure!'); // logs hello
closure!
```

source: http://index-of.es/Varios/Basarat%20Ali%20Syed%20(auth.)-Beginning%20Node.js-Apress%20(2014).pdf

Share Improve this answer Follow

answered Aug 3, 2017 at 9:15

community wiki
shohan

0

Please have a look below code to understand closure in more deep:

```
        for(var i=0; i< 5; i++){
            setTimeout(function(){
                console.log(i);
            }, 1000);
        }
```

Here what will be output? `0,1,2,3,4` not that will be `5,5,5,5,5` because of closure

So how it will solve? Answer is below:

```
       for(var i=0; i< 5; i++){
           (function(j){     //using IIFE
                setTimeout(function(){
                               console.log(j);
                           },1000);
            })(i);
        }
```

Let me simple explain, when a function created nothing happen until it called so for loop in 1st code called 5 times but not called immediately so when it called i.e after 1 second and also this is asynchronous so before this for loop finished and store value 5 in var i and finally execute `setTimeout` function five time and print `5,5,5,5,5`

Here how it solve using IIFE i.e Immediate Invoking Function Expression

```
        (function(j){  //i is passed here
             setTimeout(function(){
                            console.log(j);
                        },1000);
         })(i);  //look here it called immediate
that is store i=0 for 1st loop, i=1 for 2nd loop,
and so on and print 0,1,2,3,4
```

For more, please understand execution context to understand closure.

- There is one more solution to solve this using let (ES6 feature) but under the hood above function is worked

  ```
   for(let i=0; i< 5; i++){
       setTimeout(function(){
                      console.log(i);
                  },1000);
   }

  Output: 0,1,2,3,4
  ```

=> More explanation:

In memory, when for loop execute picture make like below:

Loop 1)

```
      setTimeout(function(){
                     console.log(i);
                 },1000);
```

Loop 2)

```
      setTimeout(function(){
                     console.log(i);
                 },1000);
```

Loop 3)

```
      setTimeout(function(){
                     console.log(i);
                 },1000);
```

Loop 4)

```
      setTimeout(function(){
                     console.log(i);
                 },1000);
```

Loop 5)

```
      setTimeout(function(){
                     console.log(i);
                 },1000);
```

Here i is not executed and then after complete loop, var i stored value 5 in memory but it's scope is always visible in it's children function so when function execute inside `setTimeout` out five time it prints `5,5,5,5,5`

so to resolve this use IIFE as explain above.

Share Improve this answer Follow

edited Dec 15, 2018 at 16:14

community wiki
3 revs, 2 users 74%
arun

thanks for your answer. it would be more readable if you separated code from explanation. (don't indent lines that are not code) – eMBee Dec 15, 2018 at 16:07

0

Currying : It allows you to partially evaluate a function by only passing in a subset of its arguments. Consider this:

```
function multiply (x, y) {
  return x * y;
}

const double = multiply.bind(null, 2);

const eight = double(4);

eight == 8;
```

Closure: A closure is nothing more than accessing a variable outside of a function's scope. It is important to remember that a function inside a function or a nested function isn't a closure. Closures are always used when need to access the variables outside the function scope.

```
function apple(x){
   function google(y,z) {
    console.log(x*y);
   }
   google(7,2);
}

apple(3);

// the answer here will be 21
```

Share Improve this answer Follow

answered May 4, 2019 at 3:16

community wiki
user11335201

0

Closure is very easy. We can consider it as follows : Closure = function + its lexical environment

Consider the following function:

```
function init() {
    var name = "Mozilla";
}
```

What will be the closure in the above case ? Function init() and variables in its lexical environment ie name. **Closure** = init() + name

Consider another function :

```
function init() {
    var name = “Mozilla”;
    function displayName(){
        alert(name);
}
displayName();
}
```

What will be the closures here ? Inner function can access variables of outer function. displayName() can access the variable name declared in the parent function, init(). However, the same local variables in displayName() will be used if they exists.

Closure 1 : init function + (name variable + displayName() function) --> lexical scope

Closure 2 : displayName function + (name variable) --> lexical scope

Share Improve this answer Follow

answered May 29, 2019 at 17:48

community wiki
Rumel

0

A simple example in Groovy for your reference:

```
def outer() {
    def x = 1
    return { -> println(x)} // inner
}
def innerObj = outer()
innerObj() // prints 1
```

Share Improve this answer Follow

answered Sep 26, 2019 at 22:12

community wiki
GraceMeng

0

Here is an example illustrating a closure in the Scheme programming language.

First we define a function defining a local variable, not visible outside the function.

```
; Function using a local variable
(define (function)
  (define a 1)
  (display a) ; prints 1, when calling (function)
  )
(function) ; prints 1
(display a) ; fails: a undefined
```

Here is the same example, but now the function uses a global variable, defined outside the function.

```
; Function using a global variable
(define b 2)
(define (function)
  (display b) ; prints 2, when calling (function)
  )
(function) ; prints 2
(display 2) ; prints 2
```

And finally, here is an example of a function carrying its own closure:

```
; Function with closure
(define (outer)
  (define c 3)
  (define (inner)
    (display c))
  inner ; outer function returns the inner
function as result
  )
(define function (outer))
(function) ; prints 3
```

answered Feb 22, 2021 at 14:33

community wiki
Kim Mens