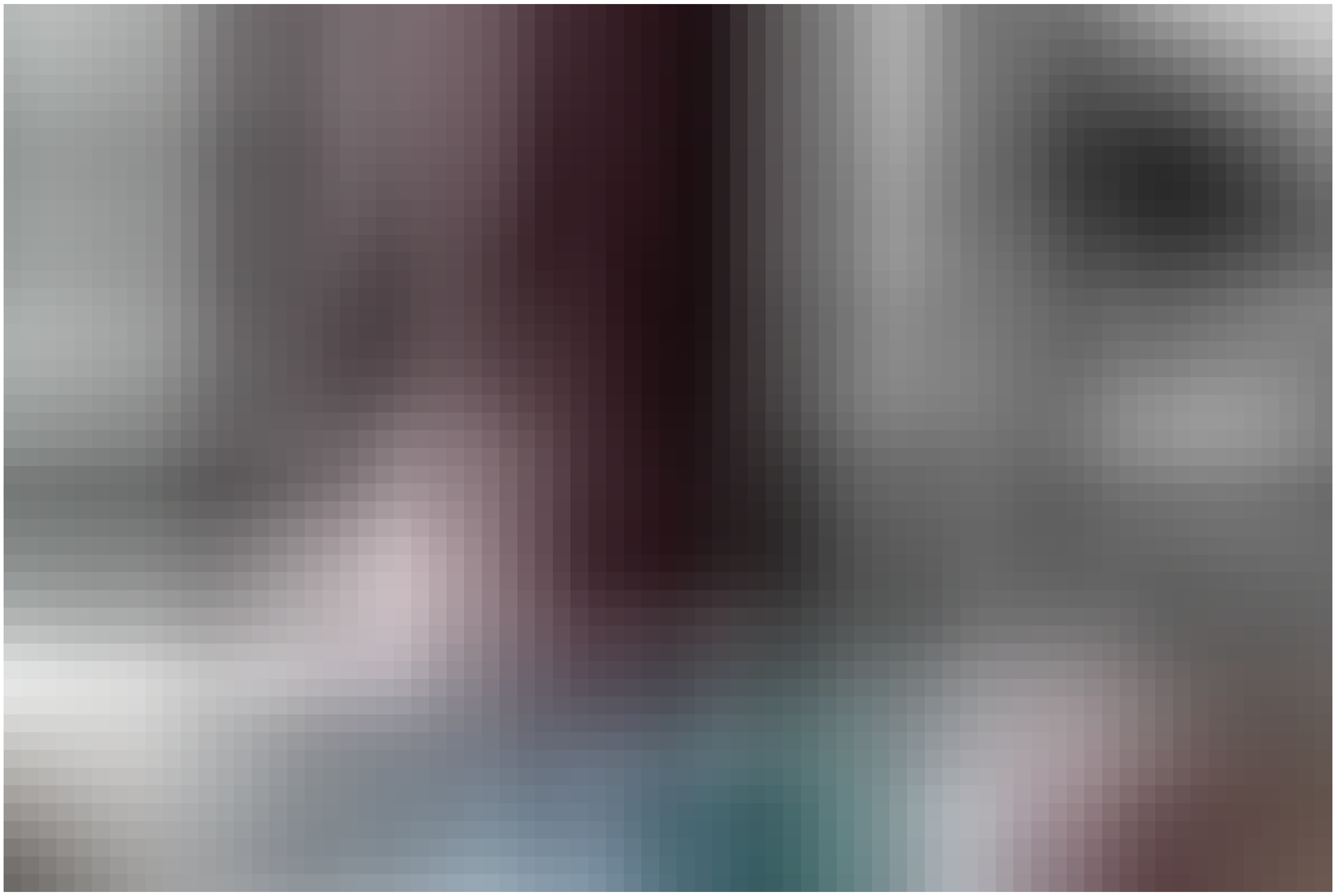
Kevin Kononenko Follow

Founder of Manual (https://www.rtfmanual.io). Self-taught web developer. Passionate about not making sa…

Jul 19, 2016 · 5 min read



JavaScript Closures Explained by Mailing a Package

If you have mailed a package or letter in the past, then you can understand closures in JavaScript.

On your journey to becoming an intermediate or advanced JavaScript dev, you may have come across closures. After reading a technical resource on the subject… you also probably ran in the opposite direction.

Here is the awesome thing about closures: they allow you to write functions with an intermediate step that can capture data from your site at a specific moment in time. It's like adding a 'pause' button to your function. You can run your function and save the value of a variable at that particular point in time. Then, when you want to resume the function at a later point and use values of variables that have changed within your app… you can do that with a **closure,** or a function **within the original function**.

This gets easier, I promise.

So, when the heck would you use a closure?

Let's say you are building an interactive map of tourist landmarks in New York City using the Google Maps API. You have a JSON object with a bunch of map markers that you want to add to the map- the Statue of Liberty, Empire State Building, Coney Island, you name it. You want to add all these markers to the map, but you also want to add a click event to each marker. When you click the marker, you want to show dynamic information about that marker, including live weather data.

```
var touristPlaces= {…};


for(var i=0; i< touristPlaces.length; i++){
  var marker= touristPlaces[i];
  $(marker).click(function(){
    showToolTip(i)
  });
}
```

Here's the issue- if you write it like this, it will not work. The 'for' loop will finish before the callback in the click event can register the appropriate i value. You need to capture this intermediate point so you can call the function later with the appropriate i.

What do you need to know first?

1. Variable scoping
2. The concept of callbacks (I wrote a guide on this too!)

If you are looking for a technical explanation of closures, the guide on MDN is probably the best.

Closures have the same process as mailing a package.

Let’s look at some basic code that uses a closure to mail a package.

```
1   function packBox(item){
2       // Code that puts item in the box
3       console.log('Put ' +item+ ' in the box');
4       function addressPackage(address){
5           // Code that writes the address label
6           console.log('Addressed the box to ' +address+' and
7       }
8       return addressPackage;
9   }
10
```

The addressPackage() function is a **closure**! It can be called at any time after the packBox function has been called. It also has access to the variables and arguments from the time when packBox() was originally called.

Notice how the console.log output does not show until lines 14 and 15? **This is extremely important.** If you ran this code after line 11, you would simply see ‘Put jersey in box’. There would be no error, but the closure, addressPackage(), would not run at that point.

When you are mailing a package, you would probably agree that your job is not done until the package is filled and the address is written. Likewise, the packBox() function waits until the closure has also been called. Let’s go through this line-by-line.

Line 11: You create the variable brotherGift, which is an **instance** of the packBox() function. You are sending a jersey to your brother.

Line 3: Your code logs a statement about the jersey.

Line 8: The packBox() function returns… another function? Huh?

Let’s stop here, and assume that line 13 has not run yet. Here is what is happening: The packBox() function will not return the “ready to send” line until you also call the addressPackage() function with an argument. Just like there are two steps to sending a package: first, filling it, and second, addressing it. Your package is worthless if it has no contents or it does not have an address! That being said, you do not necessarily need to address the package directly after you fill the contents. You can wait a few days before addressing it. You might need to go to your computer to look up the address. You might be waiting for your brother to officially change his address!

Regardless, if you do not address the package immediately, this does not mean that the package will somehow magically empty itself. **The contents will still be there when you return to address it!** So, any time we call brotherGift, the first argument, jersey, will still be available.

…Waiting…Waiting…Now let’s run line 13.

Line 13: Alright, let’s finish off this **instance**! You are ready to add the address, so you call brotherGift and offer the address as an argument. Remember from line 11, brotherGift is an **instance** of packBox with the ‘jersey’ argument. So when you call it, you are offering another argument, which will then be sent to the closure: addressPackage();

Line 3: The console.log will show since we are now running the code from line 13.

Line 4: We now offer the second argument to addressPackage();

Line 6: addressPackage logs a statement related to the address argument.

Line 8: The return statement can fire for this instance.

Again, closures allow us to have this intermediate instance where one argument has been filled, but brotherGift is left unfulfilled until we add the second argument. **If we wanted to do this in one line**, we would write: packBox(‘jersey’)(‘123 Main Street, Anywhere USA 01234’);

One More Example

Let’s say that you wanted to send a gift to each member of your family. You might pack each box before adding the addresses to each. This is what that looks like in code.

```
 1  var brotherGift = packBox('jersey')
 2  var motherGift = packBox('iTunesCard')
 3  var fatherGift = packBox('golfclubs')
 4  var sisterGift = packBox('lacrossestick')
 5
 6  brotherGift('123 Main Street, Anywhere USA 01234')
 7  //Put jersey in the box
 8  // Addressed the box to 123 Main Street, Anywhere USA 01234
 9  motherGift('123 High Street, Los Angeles USA 01234')
10  //Put iTunesCard in the box
11  // Addressed the box to 123 High Street, Los Angeles USA 01
12  fatherGift('123 Upper East Street, New York City NY 01234')
```

Another magical feature of closures! Each instance is able to use the correct gift item with the correct address, even thought we run the function with 4 separate gift/address pairs. In a traditional function, there is no concept of memory. You would need to explicitly restate the original gifts in lines 6–15 if you wanted to use a traditional function.

Where You Will Use This

You will frequently encounter closures in Node.js. If you are just interested in the front-end, think back to our original example. If you want to write a function that considers user input at two separate stages of your app, you may want to consider a closure!

Did you enjoy this guide? Or are you having trouble with another JavaScript topic? Give it a heart and let me know in the comments!

Looking for other JavaScript concepts explained? Check out these past articles in the series.

JavaScript Promises Explained By Gambling At A Casino

Model-View-Controller (MVC) Explained Through Ordering Drinks At The Bar

JavaScript Callbacks Explained Using Minions