Michael Vollmer

Grad student, fan of parentheses.

Understanding callback functions in Javascript

Callback functions are extremely important in Javascript. They're pretty much everywhere.

If you're coming from a background in functional programming, or you have experience in writing concurrent programs in other languages, you might not have any difficulty understanding why callback functions are used the way they are in Javascript programs, but I've talked to more than one programmer who found the concept very unintuitive.

Strangely, I haven't found any good introductions to callback functions online—I mainly found bits of documentation on the call() and apply() functions, or brief code snippits demonstrating their use—so I decided to try to write a simple introduction to callbacks myself.

Functions are objects

To understand callback functions you first have to understand regular functions. This might seen like a "duh" thing to say, but functions in Javascript are a bit different than (for example) methods in Java. Functions in Javascript are first-class, and are actually objects.

You can even create functions dynamically by passing strings of Javascript code to the `Function` constructor:

```
// you can create a function by passing the
// Function constructor a string of code
var func_multiply = new Function("arg1", "arg2", "return arg1 * arg2;");
func_multiply(5,10); // 50
```

One benefit of this function-as-object concept is that you can pass code to another function in the same way you would pass a regular variable or object (because the code is literally just an object).

Passing a function as a callback

Passing a function as an argument is easy.

```
// define our function with the callback argument
function some_function(arg1, arg2, callback) {
  // this generates a random number between
  // arg1 and arg2
  var my_number = Math.ceil(Math.random() * (arg1 - arg2) + arg2);
  // then we're done, so we'll call the callback and
  // pass our result
  callback(my_number);
}
// call the function
some_function(5, 15, function(num) {
  // this anonymous function will run when the
  // callback is called
  console.log("callback called! " + num);
});
```

It might seem silly to go through all that trouble when the value could just be returned normally, but there are situations where that's impractical and callbacks are necessary.

Don't block the way

Traditionally functions work by taking input in the form of arguments and returning a value using a return statement (ideally a single return statement at the end of the function: one entry point and one exit point). This makes sense. Functions are essentially mappings between input and output.

Javascript gives us an option to do things a bit differently. Rather than wait around for a function to finish by returning a value, we can use callbacks to do it asynchronously. This is useful for things that take a while to finish, like making an AJAX request, because we aren't holding up the browser. We can keep on doing other things while waiting for the callback to be called. In fact, very often we are required (or, rather, strongly encouraged) to do things asynchronously in Javascript.

Here's a more comprehensive example that uses AJAX to load an XML file, and uses the call() function to call a callback function in the context of the requested object (meaning that when we call the `this` keyword inside the callback function it will refer to the requested object):

```
function some_function2(url, callback) {
  var httpRequest; // create our XMLHttpRequest object
  if (window.XMLHttpRequest) {
    httpRequest = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    // Internet Explorer is stupid
    httpRequest = new
    ActiveXObject("Microsoft.XMLHTTP");
  }

  httpRequest.onreadystatechange = function() {
    // inline function to check the status
    // of our request
    // this is called on every state change
    if (httpRequest.readyState === 4 &&
      httpRequest.status === 200) {
      callback.call(httpRequest.responseXML);
      // call the callback function
    }
  };
  httpRequest.open('GET', url);
  httpRequest.send();
}
// call the function
some_function2("text.xml", function() {
  console.log(this);
});
console.log("this will run before the above callback");
```

In this example we create the `httpRequest` object and load an XML file. The typical paradigm of returning a value at the bottom of the function no longer works here. Our request is handled asynchronously, meaning that we start the request and tell it to call our function when it finishes.

We're using two anonymous functions here. It's important to remember that we could just as easily be using named functions, but for sake of brevity they're just written inline. The first anonymous function is run every time there's a state change in our `httpRequest` object. We ignore it until the state is 4 (meaning it's done) and the status is 200 (meaning it was successful). In the real world you'd want to check if the request failed, but we're assuming the file exists and can be loaded by the browser. This anonymous function is assigned to `httpRequest.onreadystatechange`, so it is not run right away but rather called every time there's a state change in our request.

When we finally finish our AJAX request using `call()`. Alternatively you could use `apply()` (the difference between the two is beyond the scope of this tutorial, but it involves how you pass arguments to the function).

The neat thing about using `call()` is that we set the context in which the function is executed. This means that when we use the `this` keyword inside our callback function it refers to whatever we passed as the first argument for `call()`. In this case, when we refer to this inside our anonymous callback function we are referring to the `responseXML` from the AJAX request.

Finally, the second console.log statement will run before the first, because the callback isn't executed until the request is over, and until that happens the rest of the code goes right on ahead and keeps running.

Wrapping it up

Hopefully now you should understand callbacks well enough to use them in your own code. I still find it hard to structure code that is based around callbacks (it ends up looking like spaghetti… my mind is too accustomed to regular structured programming), but they're a very powerful tool and one of the most interesting parts of the Javascript language.

PUBLISHED BY

Mike Vollmer
I'm a computer science PhD student at Indiana University, and I study programming languages and compilers. View all posts by Mike Vollmer →

March 22, 2011 Mike Vollmer javascript, tutorial