Saved from Callback Hell

By Camilo Reyes (https://www.sitepoint.com/author/creyes/) October 04, 2016

This article was peer reviewed by Mallory van Achterberg (https://www.linkedin.com/in/stommepoes), Dan Prince (https://www.sitepoint.com/author/dprince) and Vildan Softic (https://www.sitepoint.com/author/vildanssoftic). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

More from this author

Quick Tip: How to Throttle Scroll Events (https://www.sitepoint.com/throttle-scroll-events/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

Getting Started with the Raspberry Pi GPIO Pins in Node.js (https://www.sitepoint.com/getting-started-with-the-raspberry-pi-gpio-pins-in-node-js/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

Callback hell is real. Often developers see callbacks as pure evil, even to the point of avoiding them. JavaScript's flexibility does not help at all with this. From the surface, it seems callbacks are the perfect foot gun, so it is best to replace them.

The good news is there are simple steps to get saved from callback hell. I feel eliminating callbacks in your code is like amputating a good leg. A callback function is one of the pillars of JavaScript and one of its good parts. When you replace callbacks, you are often just swapping problems.

A friend tells me callbacks are ugly warts and the reason to study better languages. Well, are callbacks that ugly?

Wielding callbacks in JavaScript has its own set of rewards. There is no reason to avoid JavaScript because callbacks can turn into ugly warts.

Let's dive into what sound programming has to offer with callbacks. My preference is to stick to SOLID principles (https://en.wikipedia.org/wiki/SOLID_) and see where this takes us.

What Is Callback Hell?

I know what you may be thinking, what the hell is a callback and why should I care? In JavaScript, a callback is a function that acts as a delegate. The delegate executes at an arbitrary moment in the future. In JavaScript, the delegation (https://en.wikipedia.org/wiki/Delegation_) happens when the receiving function calls the callback. The receiving function may do so at any arbitrary point in its execution.

In short, a callback is a function passed in as an argument to another function. There is no immediate execution since the receiving function decides when to call it. The following code sample (https://github.com/beautifulcoder/FunWithCallbacks/commit/9b5fad82a3b8dbd848c879dabfe1c6be84792c1a) illustrates:

```
function receiver(fn) {
  return fn();
}

function callback() {
  return 'foobar';
}

var callbackResponse = receiver(callback);
// callbackResponse == 'foobar'
```

If you have ever written an Ajax request, then you have encountered callback functions. Asynchronous code uses this approach since there is no guarantee when the callback will execute.

The problem with callbacks stems from having async code that depends on another callback. I will illustrate the use of `setTimeout` to simulate async calls with callback functions.

Feel free to follow along, the repo is out on GitHub (https://github.com/beautifulcoder/FunWithCallbacks). Most code snippets will come from there so you can play along.

Behold, the pyramid of doom!

```
setTimeout(function (name) {
  var catList = name + ',';

  setTimeout(function (name) {
    catList += name + ',';

    setTimeout(function (name) {
      catList += name + ',';

      setTimeout(function (name) {
        catList += name + ',';

        setTimeout(function (name) {
          catList += name;

          console.log(catList);
        }, 1, 'Lion');
      }, 1, 'Snow Leopard');
    }, 1, 'Lynx');
  }, 1, 'Jaguar');
}, 1, 'Panther');
```

Looking at the above, `setTimeout` gets a callback function that executes after one millisecond. The last parameter just feeds the callback with data. This is like an Ajax call except the return `name` parameter would come from the server.

There is a good overview of the setTimeout function (https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setTimeout) on MDN.

I am gathering a list of ferocious cats through asynchronous code. Each callback gives me a single cat name and I append that to the list. What I am attempting to achieve sounds reasonable. But, given the flexibility of JavaScript functions, this is a nightmare.

Anonymous Functions

You may notice the use of anonymous functions in that previous example. Anonymous functions are unnamed function expressions that get assigned to a variable or passed as an argument to other functions.

Using anonymous functions in your code is not recommended by some programming standards (http://nodeguide.com/style.html#named-closures). It is better to name them, so `function getCat(name){}` instead of `function (name){}`. Putting names in functions adds clarity to your

programs. These anonymous functions are easy to type but send you barreling down on a highway to hell. When you go down this winding road of indentations, it is best to stop and rethink.

One naive approach to breaking this mess of callbacks is to use function declarations:

```
setTimeout(getPanther, 1, 'Panther');

var catList = '';

function getPanther(name) {
  catList = name + ',';

  setTimeout(getJaguar, 1, 'Jaguar');
}

function getJaguar(name) {
  catList += name + ',';

  setTimeout(getLynx, 1, 'Lynx');
}

function getLynx(name) {
  catList += name + ',';

  setTimeout(getSnowLeopard, 1, 'Snow Leopard');
}

function getSnowLeopard(name) {
  catList += name + ',';

  setTimeout(getLion, 1, 'Lion');
}

function getLion(name) {
  catList += name;

  console.log(catList);
}
```

You will not find this snippet on the repo, but the incremental improvement is on this commit (https://github.com/beautifulcoder/FunWithCallbacks/commit/59017f3c859603cfdcbba8e009c123a1da8456af).

Each function gets its own declaration. One upside is we no longer get the gruesome pyramid. Each function gets isolated and laser focused on its own specific task. Each function now has one reason to change, so it is a step in the right direction. Note that `getPanther()`, for example, gets assigned to the parameter. JavaScript doesn't care how you create callbacks. But, what are the downsides?

For a full breakdown of the differences, see this SitePoint article on Function Expressions vs Function Declarations (https://www.sitepoint.com/function-expressions-vs-declarations/).

A downside, though, is that each function declaration no longer gets scoped inside the callback. Instead of using callbacks as a closure, each function now gets glued to the outer scope. Hence why `catList` gets declared in the outer scope, as this grants the callbacks access to the list. At times, clobbering the global scope is not an ideal solution. There is also code duplication, as it appends a cat to the list and calls the next callback.

These are code smells inherited from callback hell. Sometimes, striving to enter callback freedom needs perseverance and attention to detail. It may start to feel as if the disease is better than the cure. Is there a way to code this better?

Dependency Inversion

The dependency inversion principle says we should code to abstractions, not to implementation details. At the core, take a large problem and break it into little dependencies. These dependencies become independent to where implementation details are irrelevant.

This SOLID principle states (https://en.wikipedia.org/wiki/Dependency_inversion_principle):

> When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.

So what does this blob of text mean? The good news is by assigning a callback to a parameter, guess what? You are already doing this! At least in part, to get decoupled, think of callbacks as dependencies. This dependency becomes a contract. From this point forward you are doing SOLID programming.

One way to gain callback freedom is to create a contract:

```
fn(catList);
```

This defines what I plan to do with the callback. It needs to keep track of a single parameter, that is, my list of ferocious cats.

This dependency can now get fed through a parameter:

```
function buildFerociousCats(list, returnValue, fn) {
  setTimeout(function asyncCall(data) {
    var catList = list === '' ? data : list + ',' + data;

    fn(catList);
  }, 1, returnValue);
}
```

Note function expression `asyncCall` gets scoped to the closure `buildFerociousCats`. This technique is powerful when coupled with callbacks in async programming. The contract executes asynchronously and gains the `data` it needs, all with sound programming. The contract gains the freedom it needs as it gets decoupled from the implementation. Code that is beautiful uses JavaScript's flexibility to its own advantage.

The rest of what needs to happen becomes self-evident. One can do:

```
buildFerociousCats('', 'Panther', getJaguar);

function getJaguar(list) {
  buildFerociousCats(list, 'Jaguar', getLynx);
}

function getLynx(list) {
  buildFerociousCats(list, 'Lynx', getSnowLeopard);
}

function getSnowLeopard(list) {
  buildFerociousCats(list, 'Snow Leopard', getLion);
}

function getLion(list) {
  buildFerociousCats(list, 'Lion', printList);
}

function printList(list) {
  console.log(list);
}
```

Note there is no code duplication. The callback now keeps track of its own state without global variables. A callback, for example, `getLion` can get chained with anything that follows the contract. That is any abstraction that takes a list of ferocious cats as a parameter. This sample code is up on GitHub (https://github.com/beautifulcoder/FunWithCallbacks/blob/master/callbackDependencyInversion.js).

Polymorphic Callbacks

What the heck, let's get a little crazy. What if I wanted to change the behavior from creating a comma separated list to a pipe delimited one? One problem I see is `buildFerociousCats` got glued to an implementation detail. Note the use of `list + ',' + data` to do this.

The simple answer is polymorphic behavior with callbacks. The principle remains: treat callbacks like a contract and make the implementation irrelevant. Once the callback elevates to an abstraction the specific details can change at will.

Polymorphism opens up new ways of code reuse in JavaScript. Think of a polymorphic callback as a way to define a strict contract, while allowing enough freedom that implementation details no longer matter. Note that we are still talking about dependency inversion. A polymorphic callback is just a fancy name that points out one way to take this idea further.

Let's define the contract. One can use the `list` and `data` parameters in this contract:

```
cat.delimiter(cat.list, data);
```

Then take `buildFerociousCats` and make a few tweaks:

```
function buildFerociousCats(cat, returnValue, next) {
  setTimeout(function asyncCall(data) {
    var catList = cat.delimiter(cat.list, data);

    next({ list: catList, delimiter: cat.delimiter });
  }, 1, returnValue);
}
```

The JavaScript object `cat` now encapsulates the `list` data and `delimiter` function. The `next` callback chains async callbacks, this was formerly called `fn`. Note there is freedom to group parameters at will with a JavaScript object. The `cat` object expects two specific keys, both `list` and `delimiter`. This JavaScript object is now part of the contract. The rest of the code remains the same.

To fire this up, one can do:

```
buildFerociousCats({ list: '', delimiter: commaDelimiter }, 'Panther', getJagua
buildFerociousCats({ list: '', delimiter: pipeDelimiter }, 'Panther', getJaguar)
```

The callbacks get swapped. As long as contracts get fulfilled, implementation details are irrelevant. One can change the behavior with ease. The callback, which is now a dependency, gets inverted into a high-level contract. This idea takes what we already know about callbacks and raises it to a new level. By reducing callbacks into contracts, it lifts up abstractions and decouples software modules.

What is so radical is that from independent modules naturally flow unit tests. The `delimiter` contract is a pure function. This means, given a number of inputs, one gets the same output every single time. This level of testability adds confidence that the solution will work. After all, modular independence grants the right to self-assess.

An effective unit test around the pipe delimiter could look something like this:

```
describe('A pipe delimiter', function () {
  it('adds a pipe in the list', function () {
    var list = pipeDelimiter('Cat', 'Cat');

    assert.equal(list, 'Cat|Cat');
  });
});
```

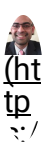
I'll let you imagine what the implementation details look like. Feel free to check out the commit on GitHub (https://github.com/beautifulcoder/FunWithCallbacks/commit/f5cd1988ecb29d97b10180cbe1ec246adbc53d03).

Conclusion

Mastering callbacks in JavaScript is understanding all the minutiae. I hope you see the subtle variations in JavaScript functions. A callback function becomes misunderstood when one lacks the fundamentals. Once JavaScript functions are clear, SOLID principles soon follow. It requires a strong grasp of the fundamentals to get a shot at SOLID programming. The inherent flexibility in the language places the burden of responsibility on the programmer.

What I love the most is JavaScript empowers good programming. A good grasp of all the minutiae and fundamentals will take you far in *any* language. This approach is super important with callback functions in vanilla JavaScript. By necessity, all the nooks and crannies will take your skills to the next level.

More: Opinion (https://www.sitepoint.com/tag/opinion/), Refactoring (https://www.sitepoint.com/tag/refactoring/), vanilla javascript (https://www.sitepoint.com/tag/vanilla-javascript/)


(http

Meet the author

Camilo Reyes (https://www.sitepoint.com/author/creyes/) (https://plus.google.com/117305015265276164398/about) (http://www.linkedin.com/profile/view?id=244644387&trk=nav_responsive_tab_profile) (https://github.com/beautifulcoder)

Husband, father, and software engineer living in Houston Texas. Passionate about JavaScript, C#, and webbing all the things.

- *M S i N Lund*
  I must be tired, because i can make no sense out of this.
  It seems like a really convoluted way to explain …something?

  I think it would really help, if you just put the code in the article, instead of linking to the user-hostile mess that is github

  - *pbruna*
    I second that. Also, the code still looks ugly.

    - *Camilo Reyes*
      Thanks for the feedback. Highly recommend downloading the code samples. They are meant to get acquainted with callbacks and SOLID principles. There is only so much one can cover within a single article.

      https://github.com/beautifulcoder/FunWithCallbacks (https://github.com/beautifulcoder/FunWithCallbacks)

      - *tomByrer*
        To me, the examples make it hard to imagine where you would use callbacks.
        I personally understand since 2 months ago I had many chained callbacks to concoct API calls into 1 large JSON array. But if I did not have that experience, I would think your examples are solving a pointless problem. Perhaps a better real life example would be better.

        + I wish your GH repo examples were sorted by when they appeared in the article (eg 01-simple-callback.js), & mixed case filenames seem amateurish.

        - *Camilo Reyes*
          If you swamp out the `setTimeout` for an Ajax call, does that help? I agree it's more of a toy example. The idea is to get you to think in SOLID terms. The `setTimeout` essentially simulates a real world example.

- *Alexandro Perez*
  Awesome! I feel like I can come to like callbacks if I use them this way.

- *Bob Jones*
  Can I ask why you pass in a param into the setTimeout of buildFerociousCats, instead of just:

```
function buildFerociousCats(list, data, fn) {
setTimeout(function asyncCall() {
var catList = list === '' ? data : list + ',' + data;
fn(catList);
}, 1);
}
```

Maybe for some memory leak reason, though not 100% sure..

- *Camilo Reyes*
  Good question, the `setTimeout` simulates an Ajax call, for example. In a real world scenario, the `data` is not known until you execute the callback. Hence `function asyncCall(data) { .. }`, since in Ajax, that's when the callback gets data from the server.

- *steve*
  So the conclusion to this article is – callbacks are ok when provided with their own argument and named:
  SomeFunc(someArg, function myCallback(myCallbackArg){/*dostuff with myCallbackArg only*/

  I agree with the above. I think the reason many developers mostly don't like it is 1, many levels of nesting make it difficult to reason with. So another key is to keep nesting callbacks to a minimum, maybe 4.

  And 2, many developers from more classically trained backgrounds are used to classes which have their own async method where they can access 'this' keyword from them so its just a different syntax that wouldn't enable nesting.

~~React The ES6 Way~~

LATEST BOOKS ›
(/premium/books/)

PREMIUM BOOK

ECMAScript 2015: A SitePoint Anthology (https://www.sitepoint.com/premium/books/ecmascript-2015-a-sitepoint-anthology)

PREMIUM BOOK

Jump Start Git (https://www.sitepoint.com/premium/books/jump-start-git)

PREMIUM BOOK

Full Stack JavaScript Development with MEAN (https://www.sitepoint.com/premium/books/full-stack-javascript-development-with-mean)

Get the latest in JavaScript, once a week, for free.

Enter your email

Subscribe

Facebook LinkedIn Twitter

(/)

ABOUT

Our Story (/about-us/)