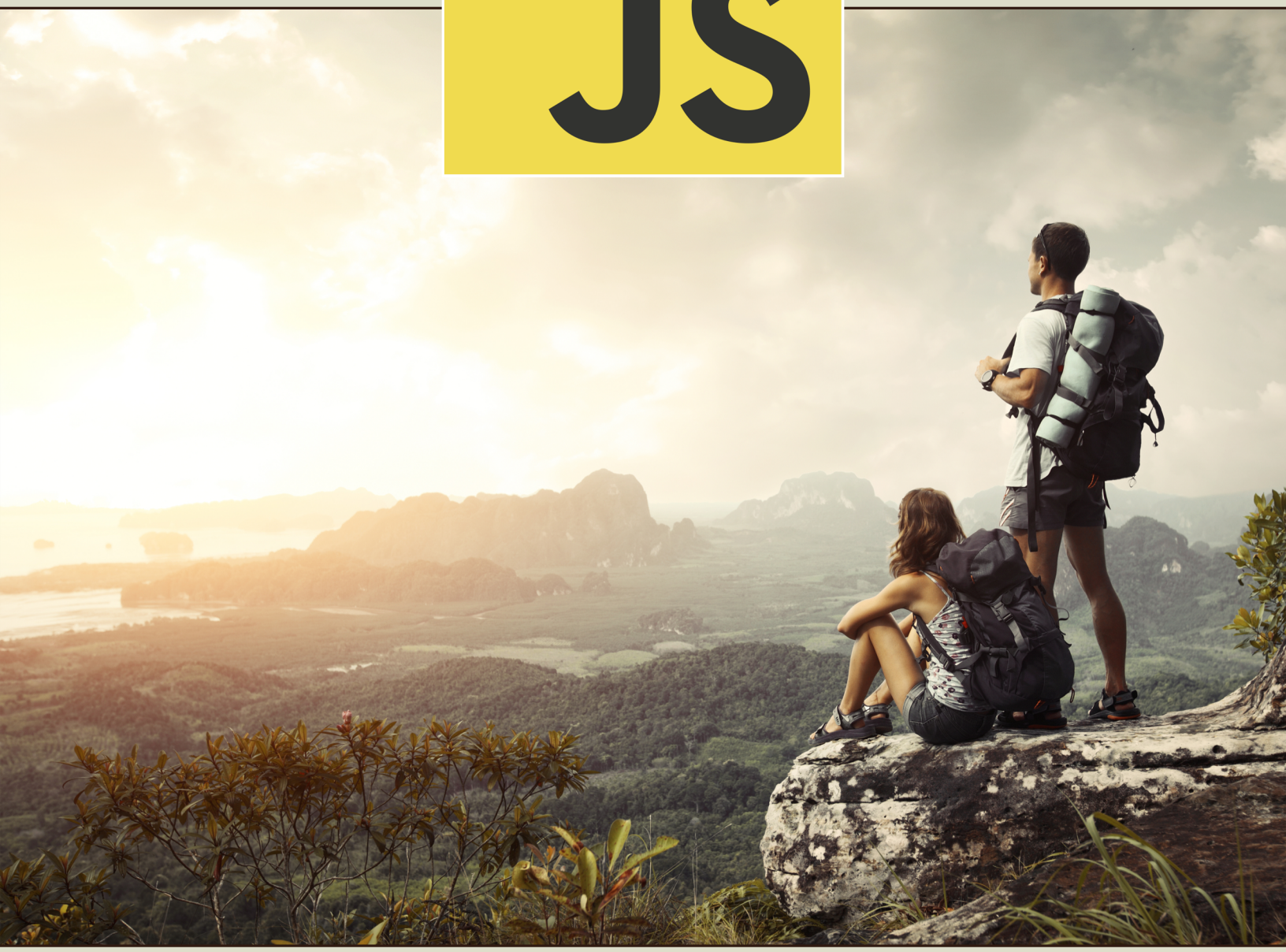
5 Rules For

Mastering JavaScript's "this"



A multi-day journey, exploring JavaScript's most notorious keyword

by Derick Bailey

5 Rules For Mastering JavaScript's "this"

A multi-day journey, exploring JavaScript's most notorious keyword

Derick Bailey

This book is for sale at http://leanpub.com/mastering-javascripts-this

This version was published on 2014-07-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 Muted Solutions, LLC. All Rights Reserved.

Tweet This Book!

Please help Derick Bailey by spreading the word about this book on Twitter!

The suggested hashtag for this book is #MasteringJavaScriptThis.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#MasteringJavaScriptThis

Also By Derick Bailey

Building Backbone Plugins

Prepare, Produce, Publish

Contents

About This Book

Wondering why JavaScript's "this" is pointing to that, when you thought it was pointing over there? Confused by the seemingly arbitrary value of "this" … if it even has a value?

You're not alone.

JavaScript's context variable is one of the most frustrating and confusing bits of important information that you need to understand. But don't worry – the rules for managing "this" are easier to understand than most people think. It only takes a little knowledge, a few examples to work with, and a willingness to travel the path.

The name says it all: 5 Rules For Mastering JavaScriot's "this". This book is split up in to multiple "days" - a short chapter each day, designed to guide you down the path of mastering "this". Every day, for 5 days, read the next chapter.

How This Book Is Laid Out

You'll start with "Day 0" an introduction that shows you the map that you'll be following. From there, Day 1 starts you out on what may be the most difficult leg of the journey. If you can get past Day 1, the rest of the journey will be smooth sailing!

Along the way, you'll learn lessons through examples and demonstrations of code that illustrate the rules and patterns for managing "this". But beware – this isn't a course that will give you all of the answers and demonstrations in a JSBin or JSFiddle. While you will see plenty of code samples, I have purposely left it up to you to run the code so that you can see what happens, first-hand. Work through the examples, study the text and put your knowledge to work in your applications. Taking a hands-on approach to learning "this" will equip you with the skills you need.

Day 0: Preparing For The Journey

I can't even count the number of times I've screamed in anger at my computer screen, wondering why "this" wasn't pointing to the right thing. It's such a frustrating experience - to set things up exactly like you've seen in the examples and demos, to run the code and get a big fat error message in your console because the method or attribute isn't there like you expect!

AAAARGH!

But no more! You're here, now, and you're in a safe place where the rules of managing JavaScript's "this" will become known to you!

The 5 Rules Of JavaScript's "this"

There are only 5 rules you need to understand, to master the value of "this" in any function and any object in JavaScript. Here's the kicker, though: these rules are all based on how you invoke the function! That's right - how you invoke the function in question. It's not the object that the method comes from, or the file, or the nesting or anything else - though these can play a part in how you invoke a function. Ultimately, though, the 5 rules center around JavaScript's function invocation pattern:

- Raw function calls set "this" to the global or undefined
- Method calls on objects set "this" to the object
- Function calls with "new" set "this" to a new object instance
- Using .call and .apply to invoke a function can override "this"
- Function binding with .bind set permanently set "this"

The Real Journey Begins Tomorrow

I've shown you the map, today, but the real journey starts with tomorrow's chapter. And I have to say, it's a bit of a rough ride ahead. While these rules are easily manageable and understandable by any developer, they can take a bit of deep thought and experimentation to wrap your head around them. So sit back and relax for now, and check your inbox tomorrow. Over the next 5 days, read each of the chapters explaining and demonstrating one of these rules. And by the time you're done reading this book, you will have everything you need to understand and apply a mastery of JavaScript's "this".

Day 1: Simple Function Invocation

Welcome to day 1 of your journey to master JavaScript's "this"! In this course, you will see sample code and discussion points that center around the 5 rules of JavaScript's most notorious little keyword. It's a bit of a journey, but it's well worth the effort if you're serious about improving your JavaScript skills. So grab your hiking boot, backpack and survival knife. Your journey starts now!

Simple Function Invocation

The single most basic way to invoke a function in JavaScript is to supply the () parenthesis to a function that is just floating around, waiting to be called.

```
function doStuff(){
  console.log("I'm doing things!");
  console.log(this);
}

doStuff();
```

This is just a raw function call - nothing special, here, except for the logging of "this" to the console. The question, then, is what does "this" evaluate as? The answer, like almost everything else in software development, is... it depends.

The Default Value of "this"

The good news is that you can classify the value of "this" as the default value, when a function is invoked in this manner. The default value is almost always the "global" object for the runtime - but not always. The bad news, then, is that this function call may or may not have a value for "this". Whether or not it has a value, depends on the runtime environment in which the function is being invoked (and whether or not the "new" keyword is used - you'll see more on the "new" keyword in a later chapter).

To figure out what the value of "this" is, you need to know what environment in which the function is being invoked. The basic environments that you'll need to be concerned with, are:

- A browser

- A browser w/ strict mode enabled
- NodeJS
- NodeJS w/ strict mode enabled

There are other environments, such as a MongoDB server, but those are special cases that won't be covered here.

Raw Function Calls In A Browser

The most basic function call, as shown above, will resolve "this" to the "window" object in a browser. This is the top level object that the JavaScript environment has - the object on which all "global" variables live. If you examine the output of console.log(this) in your browser, you'll see everything that the "window" object has as well. In fact, if you compare "this" and "window" in the above function, they will be equal.

```
function doStuff(){
  console.log(this === window); //=> true
}
doStuff();
```

For a raw function invocation in a browser, then, the value of "this" will be the "global" or "window" object.

Strict Mode In A Browser

Modern browsers have a "strict" mode that you can enable with a very simple command: "use strict". One interesting note about this, is that you have to actually put it in quotes - it's just a raw string. This is done for backward compatibility reasons with older browsers. When "use strict" is set inside of a function, that function is said to be in strict mode, and strict mode changes the default value of "this".

```
function doStuff(){
  "use strict";
  console.log(this);
}

doStuff();
```

If you examine the output of this function invocation, you'll notice that "this" comes out as "undefined". Strict mode, then, does not provide direct access to the "global" or "window" object via "this". It turns "this" in to an undefined value.

One of the effects of this change is that you cannot assign attributes to "this"... "this" is undefined, so calling code like "this.foo = bar" would throw an error saying "this" is undefined. There are other effects of using strict mode, as well (but that's another discussion for another time).

In NodeJS

The last environment that you need to know about at this point, is NodeJS. NodeJS is a server side runtime of the Google V8 JavaScript engine, allowing you to run JavaScript on the server. Since there is no browser in which NodeJS runs, there is no "window" object. There is, however, a "global" object that works similarly to the "window" object in a browser.

When you run the original function from the top, in NodeJS, you will console.log the "global" object.

NodeJS With Strict Mode

Similar to a modern browser, NodeJS provides "strict mode". And similarly to how the browser works, NodeJS' "strict mode" turns the default value of "this" in to an undefined value. Once again, this prevents you from accidentally creating globally accessible values. You must be very explicit in creating them, if you want them.

In Review

The default value of "this", when using a plain old function call can change on you, is either the global object or undefined - depending on whether or not you're in strict mode. Fortunately, you're on the cutting edge of JavaScript and you always set "use strict" anyways, right? So you can safely assume that "this" will be undefined in your normal function calls.

Of course there will be times when you don't want to, or can't set "use strict". In those cases, the possible values of "this" are:

- A browser: window
- A browser w/ strict mode enabled: undefined
- NodeJS: the global object
- NodeJS w/ strict mode enabled: undefined

Take a few minutes and try out the code from these samples in both a browser and in NodeJS. Reading about this is one thing, but writing the code yourself and seeing the actual results will really help to set the information in to your mind.

Coming Up Tomorrow

Wow, what a start to this course! Understanding "this" in a normal function call seems a bit more complex than it should be - but don't worry, this is the worst it gets. The remaining rules don't have as many exceptions or edge cases, like this. In fact, tomorrow's email on method calls from an object may be the simplest rule of them all.

So stay tuned! You're through the worst part of the journey in mastering JavaScript's "this" - it just gets easier, from here.

Not a NodeJS developer, yet? No problem!

I've got a blog post on WatchMeCode that gives you free videos on Getting Started With NodeJS: Installing And Writing Your First Code[1]! Take a few minutes to run through these videos, and you'll be all set to follow along with NodeJS.

[1] http://sub.watchmecode.net/getting-started-with-nodejs-installing-and-writing-your-first-code/

Day 2: Method Calls On Objects

Ok, yesterday was a bit rough. It turns out that seemingly simple method calls can have a number of different values for "this" depending on the runtime environment and whether or not you have set "use strict" for a function. Ugh. Fortunately, today's lesson will be easier to understand. In fact, this lesson can be boiled down in to a single, simple rule.

> *Method Calls Assign "this" To The Object From Which They Are Called*

Really, that's it. That's the entire lesson right there. I'll see you tomorrow.

No? Ok. Let's see some examples of what this means, and tear that tiny little rule down to see how it's built.

An Object With A Method

JavaScript can be used in an object-oriented manner. In fact, almost everything in JavaScript is an object, whether or not you want it to be. And the most basic and simple form of objects in JavaScript are object literals - the squirrelly bracket syntax with key: value pairs that you see all over the place:

```
var myObj = {
  doStuff: function(){
    console.log(this);
  }
};

myObj.doStuff();
```

In this example code, there is an object. That object - an object literal - is assigned to the "myObj" variable. That object also contains a method. A method in object oriented speak, is simply a function that is attached to an object. To invoke this method you only need to call "myObj.doStuff();" and the method will execute.

The question, then, is what does "this" evaluate to in the above method call?

Dissecting The Rule

Go back to the simple rule that I stated above, for a moment: method calls assign "this" to the object from which they are called. Let's dissect this and assign each part of the rule to the above code.

- **Method call**: a method is a function attach to an object. Therefore the method being called is "doStuff".
- **Object from which they are called**: this is the object to which the method is attached, via the "." notation. In this case, the "doStuff" method is attached to the "myObj" object via the "." notation.
- **Lastly, we come to: assign "this" to the object**: the actual assignment of "this" is the object on which the function was called

By deconstructing the rule and looking at each of the parts, we can deduce that "this" in the "myObj.doStuff()" method call will be the object literal that is assigned to the "myObj" variable.

But... Why?

It turns out the "." (dot-notation) method call is the important factor, here. When JavaScript sees an object-dot-method invocation, it uses the "." to determine what "this" gets assigned to. Quite simply, the object on the left hand side becomes the value of "this". Therefore, in the call to "myObj.doStuff()" the value of "this" is assigned to "myObj" - or, more accurately, the object to which the "myObj" variable points.

This is generally called method invocation, where a method is a function attached to an object. The value of "this" in method invocation is generally the object on which the method was called.

There are some exceptions to this rule, of course. The most notable of the exceptions is the use of the "new" keyword, which will be covered in tomorrow's chapter. There are also ways of overriding what "this" evaluates to, and even losing "this" along the way. I'll talk about overriding "this" in future chapters but I want to cover the loss of "this" briefly.

When A Method Becomes A Function

The fastest way to go insane is to assume that a function is assigned "this" based on the object on which it is defined. This is not the case at all, and the description above should allude to this, directly. In fact, the value of "this" can change dramatically, not based on which object the method is embedded within, but based on the invocation pattern of the method (or function).

Example: take the "doStuff" method of "myObj" and assign it to a variable. Then call the function from that variable using the standard function invocation you saw yesterday.

```
var fn = myObj.doStuff;
fn();
```

Can you predict what "this" will be in the "fn()" invocation?

Think back to yesterday's chapter where you saw how the standard function invocation assigned "this" to either the "global" object or "undefined". Does this invocation pattern look familiar? What, then, will "this" be assigned to? The answer is that it depends on the runtime and whether or not you're in strict mode!

But ... Why?

When you assigned the "doStuff" method to the "fn" variable, it ceased being a method - at least, in the "fn" variable. It became a function by virtue of not having an object on which it was invoked. The call to "fn()" was a simple, ordinary function call - an invocation pattern that you saw yesterday, which told you that the value of "this" was either the global object of the runtime, or undefined.

Therefore, it is the "." (dot-notation) syntax that turns a function invocation in to a method call on an object, and not the fact that the function was defined on that object.

In Summary

Method calls - functions that are invoked on an object - set the value of "this" to the object on which they are invoked. But defining a function on an object does not mean the function will be invoked on that object. It truly is the invocation pattern that determines the value of "this", and not just the location of the function definition.

Coming Up Tomorrow

Tomorrow, we'll continue to look at how objects affect "this". Only in this case, we'll be looking at how the "new" keyword is used to affect what "this" is on specific type of function: constructor functions! So stay glued to your inbox tomorrow, as we dive a little further down in to the depths of managing and mastering JavaScript's "this".

Want To Skip To The End?

If you'd like to take the shortcut to the end of this series, or if you're more of a visual learner, check out my screencast on JavaScript Context[2]. This video is a bit older - before I was doing NodeJS - but you can still see the basic rules applied to a browser. It also require a subscription to my WatchMeCode service. If you're not already a subscriber, check the "Thank You" chapter of this book for a discount code.

[2] http://watchmecode.net/javascript-context/

Day 3: The “new” Keyword

Yesterday you saw how the dot-notation of the method invocation pattern assigned “this” to the object on which a method was called. This pattern should be familiar to anyone that has done work with C#, Java or other similar languages. The difference, of course, is that the assignment of “this” happens because of the method invocation pattern and not because of where the method was defined.

Beyond the method invocation pattern, though, there is one more way in which an object will modify the value of “this” in a function. When a function call is combined with the “new” keyword, it is said that the function is a “constructor function” - a function that intends to construct (or create) a new object instance.

A Constructor Function

Consider the following function and call to the function:

```
1  function Foo(){
2    this.bar = "baz";
3    console.log(this);
4  }
5
6  var f = new Foo();
7  console.log(f.bar);
```

In this example, the “Foo” function is being used as a constructor function - a function that is designed to create object instances. Note that there is nothing in the JavaScript language that says this is a constructor function, nor is there anything that says a function must be set up in a certain way in order to be a constructor function. Rather, there are a few subtle hints in the idioms and patterns of JavaScript usage that tell us this is a constructor function.

The first hint is that the function name is capitalized. This is standard practice in JavaScript, though it has no meaning in the language itself. The second is the assignment of “this.bar” in the function. The final hint (and the most blatant / obvious one) is the use of the “new” keyword.

Given the above code, then, what do you expect will be displayed in the console?

You will see two things in the console. The first of which will be the logging of “this” from the function itself. That call will produce an object in the console, which contains an attribute of “bar” assigned to the value of “baz”. The second thing you will see is the string “baz”, logged from the “f.bar” attribute, after the object is instantiated.

But ... Why?

When a function is used as a constructor (and keep in mind that any function can be used as a constructor - though the results are not guaranteed if the function was not designed for this), the value of “this” within that function becomes something new and unique. It becomes an object instance, of the “type” that the function represents.

In the above example, then, the value of “this” is a new instance of a “Foo” object. When the function exits, the value of “this” is returned automatically - you don’t need to return anything manually.

How Does That Work?

Here’s what happens when a function is called with the “new” keyword (keep in mind that I am oversimplifying this - you can check out the details in the ECMAScript Spec if you want all the exact bits).

- A call to “new Foo()” creates a new object instance, said to be of type “Foo”
- The object instance is assigned a prototype of “Foo.prototype”
- The “Foo” function is then invoked with the new object instance assigned to “this” in that function invocation
- The “Foo” object instance is returned from the function call

Since the value of “this” is a new object of a “Foo” type, in this example, the assignment of “this.bar” adds a “bar” attribute to the Foo instance. Logging “this” from the constructor function will log the object in it’s current state (with it’s current attributes and methods). When the constructor function exits, the object instance is assigned to the “f” variable. The “f” variable, pointing to the object instance, then has the “bar” attribute available, which is then logged to the console.

In Summary

Ok, this is very quickly getting in to the realm of objects and prototypes in JavaScript. For now, though, you should know that the “new” keyword changes the behavior of a function. It turns that function in to a constructor function. The value of “this” inside of the constructor function is a new instance of that object type, where the “type” is defined by the constructor function, itself.

As you can see, the value of “this” is once again determined by the invocation pattern of the function and not by the function definition. Calling “new Foo.Bar.Baz()” for example, will assign an instance of a “Baz” type to “this” inside of the “Baz” function. The value of “this” is not something set at design time, when writing code. It is determined dynamically when a function is invoked and over the last three days you have seen the three core function invocation patterns:

- **Simple function invocation**: doStuff();

- **Object method invocation**: myObj.doStuff();
- **Constructor functions**: new Foo();

These three function invocation patterns make up the core of what JavaScript provides for it’s standard invocation patterns - but this isn’t everything that JavaScript provides. There are technically three more invocation patterns that we’ll be covering over the next 2 days. Two of these patterns - “.call” and “.apply” - are only variations of each other and are generally grouped together. These are not standard invocation patterns, though. Rather, they are methods that allow you to explicitly set the value of “this” when invoking a function.

Coming Up Tomorrow

Coming up tomorrow, then, you’ll see how to use “.call” and “.apply” to directly manipulate “this” on any function. So stay tuned and get ready to wield a light-saber against the dark side of managing “this” in your JavaScript functions!

Want A Deep Dive On Objects?

If you’d like to dive deeper in to objects and prototypes, I have a 40 minute screencast on JavaScript Objects & Prototypes[3]. It covers the core mechanics from object literals to constructor functions and prototypal inheritance. If you’re doing object oriented JavaScript, this resource is well worth the time and money. This screencast does require a subscription to WatchMeCode, though. If you don’t yet have a subscription, check the “Thank You” chapter of this book for a discount code!

[3] https://sub.watchmecode.net/javascript-objects/

Day 4: Using .call And .apply

In the last 3 days, you've seen the three core method invocation patterns in JavaScript and how those patterns manipulate the value of "this" in the function being invoked. These invocation patterns are not the only ways in which you can manage or manipulate "this", though. In today's chapter, you'll see two additional methods that allow you to explicitly manipulate "this" in nearly any JavaScript function call.

Functions With Methods

Nearly everything in JavaScript is an object - or at least, acts like an object when methods and attributes on that thing are accessed. Functions in JavaScript are no exception to this, either. They can be treated like most other objects, most of the time - they have methods and attributes that can be called and set or read (though some of these are protected and cannot be modified).

Among the methods that functions have, are the ".call" and ".apply" methods. These methods exist on all functions and they let you invoke a function with a specified context, supplying parameters to the function in one of two different ways.

Using .call

When using .call on a function, the first parameter will be the context or this variable. Any additional parameters passed in will be pass as arguments to the original function.

```
function doStuff(b){
  return this.a + b;
}

var myContext = {a: 1};
var result = doStuff.call(myContext, 2);

console.log(result); //=> 3
```

You can supply any number of arguments to the .call method, and all of them will be passed to the original function after setting the context.

```
doStuff.call(myContext, 1, 2, [... n]);
```

The .call method is most often used when you need to dynamically call a function with a given context and a set of values that you already have as variables.

Using .apply

The use of .apply is essentially the same as that of .call, except you pass in an array as the second parameter and that array gets passed through to the original object as the list of arguments.

```
function doStuff(b, c){
  return this.a + b + c;
}

var myContext = {a: 1};
var result = doStuff.apply(myContext, [2, 3]);

console.log(result); //=> 6
```

If you try to pass more than 2 parameters to the .apply method, you'll get an error.

The .apply method is most often used when you need to dynamically call a function with a given context and a set of values in an array, such as splitting the arguments object from another function.

Similar, But Different

Using .call or .apply is a very similar thing, but slightly different in how parameters are passed to the original function. With .call, parameters are passed as a normal list of parameters. With .apply, parameters are passed as an array. The subtle difference between these two invocation methods allows a lot of flexibility in your code. You with .apply, you can dynamically access a method and pass parameters to it without having to know how many parameters it needs. With .call, you can set the context of a method and pass in a specific list of parameters that you already know.

Override Invocation Patterns

One of the places where .call and .apply can trip you up, if you're not careful, is in their ability to override the context that would have been set by a function or method invocation. If, for example, you have an object with an attribute and a method:

```
var myObj = {
  foo: "bar",

  doStuff: function(){
    console.log(this.foo);
  }
};
```

and you invoke the method with .call or .apply, you will effectively override the object-dot-method invocation pattern.

```
myObj.doStuff.call({});
```

In this example, “doStuff.call” does not follow the object-dot-notation pattern that you learned about in Day 2’s chapter. Instead, .call overrides the value of “this” for the execution of “doStuff”. And with an emtpty object literal passed in, there will be no “bar” attribute and the result of the console.log will be “undefined”.

Where To Use This

The use of .call and .apply becomes important when dealing with objects and callback methods. For example:

```
var myObj = {
  foo: "bar",

  moreStuff: function(cb){
    cb();
  },

  doStuff: function(){
    console.log(this.foo);
  }
};

myObj.moreStuff(myObj.doStuff);
```

In this example (which is a very poor example, by the way, for illustration purposes) the call to “myObj.moreStuff” receives “myObj.doStuff” as a parameter. Since this parameter does not include the “()” parenthesis to invoke the function, it is passed in as a method pointer, effectively becoming a callback method. The callback method is then invoked with the standard function invocation pattern,

meaning the value of “this” will either be the global object or undefined (depending on your runtime - see Day 1’s chapter).

Since the “cb()” call points to the “doStuff” method, the value of “this” will not be “myObj” and therefore will not contain the expected “foo” attribute. The result of the console.log, then, will be some unknown value or most likely undefined.

The ramifications of this are significant. This combination of several patterns may lead you down several incorrect paths of analysis when trying to determine what the final value of “this” is. However, it is the simple function invocation pattern of “cb()” that determines the value of “this” and not any other invocation pattern shown here.

Coming Up Tomorrow

Did you know there’s one last trick to this example? In spite of the simple function invocation of “cb()” it is still possible to guarantee the value of “this” inside of the “doStuff” function. But I’ll save that for tomorrow’s final chapter in this series on Mastering JavaScript’s “this”.

Stay tuned for tomorrow’s chapter, then, where I show you how to guarantee the value of “this” for any function, using “.bind”.

Day 5: Using .bind

Yesterday you saw the use of .call and .apply for invoking functions. These methods allow you to specify the value of "this" for a function as well as any parameters needed for that function. There's one more method of changing the context for a function, though - and this one is particularly powerful. Not only does it let you set the value of "this" and override any of the previous methods, but it does it without immediately invoking the function, giving you a new function reference to call whenever and wherever you want.

Introducing The .bind Method

The .bind method call exists on all JavaScript functions, the same as .call and .apply - at least, since ECMAScript v5. The major difference between .bind and .call / .apply, is that .bind will not invoke the function in question. Instead, it returns a new function to you. Invoking this new function will in turn invoke the original function with the context that was specified in the .bind call.

In other words, .bind allows you to set the context of a function without invoking it.

```
function doStuff(b, c){
  return this.a + b + c;
}

//bind the function to a context
var myContext = {a: 1};
var boundStuff = doStuff.bind(myContext);

// call the bound function with additional params
var result = boundStuff(2, 3);
console.log(result); //=> 6
```

In this code, the original doStuff function is "bound" to the myContext object as the context - the value of "this". The result is a new function referenced in the "boundStuff" variable. When the boundStuff function is invoked - and no matter how it is invoked - "this" will always be set to the myContext object.

Brute Force Override

In spite of the simple function invocation pattern of "boundStuff(2, 3)", the value of "this" is not going to pay attention to the original rules you learned about on the first day of this course. Instead,

the bound call will force the "doStuff" function's "this" to be the "myContext" object. This is true for all of the patterns that you have previously learned, including the use of .call and .apply. The .bind call is a brute force way of ensuring that you always have the context you want in your functions.

But Not The Original Function

It's important to note that the brute force override is true only for invoking the boundStuff function, though. If you invoke the doStuff method directly, it will still be have all of the original rules for assigning "this" applied to it.

Consider the following:

```
var myObj = {
  foo: "bar",
  doStuff: function(){
    console.log(this.foo);
  }
};

var boundStuff = myObj.doStuff.bind({
  foo: "something else"
});

boundStuff();
myObj.doStuff();
```

When this code is executed, there will be two lines of output in the console. The first of which will state "something else" due to the .bind call and the execution of "boundStuff" as a function. However, the second line of output will be "bar" because the code is executing the original "doStuff" function and not the bound function.

Permanently Override The Context

If you wish to permanently override the context of a function on an object, you can do this by assigning the result of the ".bind" call to the original method name on that object. Modifying the previous example slightly will show this:

```
var myObj = {
  foo: "bar",
  doStuff: function(){
    console.log(this.foo);
  }
};

var boundStuff = myObj.doStuff.bind({
  foo: "something else"
});

// re-assign myObj.doStuff so that it is now the bound function
myObj.doStuff = boundStuff;

// continue with the bound call and what looks like the original method call
boundStuff();
myObj.doStuff();
```

In this example there will be two lines of console.log output again. Only in this case, both of the lines will be "something else" because the method at "myObj.doStuff" was replaced with the bound function prior to it being executed.

There is an important note on the timing in this code, though. Any call to "myObj.doStuff" that happens prior to the re-assignment of the "doStuff" method, will get the original version of this function. It's only after the re-assignment that calls to "myObj.doStuff" will end up with the bound function.

Using The .bind-and-replace Technique With Constructor Functions

The bind and replace technique is especially useful when you have callbacks in objects and you need to ensure the value of "this" is the object itself. When using constructor functions, for example, you can override a method on the object to ensure it's context is never changed.

```
function MyType(){
  this.doStuff = this.doStuff.bind(this);
}

MyType.prototype.doStuff = functio(){ /* ... */ };

var myObj = new MyType();
var d = myObj.doStuff();
d();
```

Now any call to the "doStuff" function, no matter the invocation pattern, will be executed with "this" set to the instance the "MyType" object, as shown here with the simple function invocation of "d()".

A Note On The Availability Of .bind

The .bind method is only available in ECMAScript 5 and above, as noted earlier. If you need to support older browsers and JavaScript runtimes that don't have this method built in to functions, you can use third party tools or shims to get this functionality. Underscore.js, for example, as a "_.bind" method that gives you this functionality.

Summary

The use of .bind allows you to brute force override the value of "this" for any function in JavaScript when you need to. When take the bound function and re-assign it to the original object and method that was bound, it becomes even more powerful, ensuring that the context of the method will always be what you set it to.

Be careful when wielding this atomic sledgehammer, though. You may think it sounds like a good idea off-hand, but it can prove to be a toe smashing experience if you are wielding it with blind ambition and no thought about where it's going.

An Epic Journey

At this point, you have completed the core journey on mastering JavaScript's "this"! Congratulations on doing that! If you work through all of the examples from all of the chapters that you have read so far, you will find that "this" is no longer the frustrating scourge of your code and bane of your life as a developer. And don't worry if you're still a bit lost at this point. There's a lot to take in, to practice and to learn. Just go back to the chapters that are still giving you trouble and practice, practice, practice.

Before I go, though, there's one more chapter that I want you to read, tomorrow. I can't show you the core of using .bind in this chapter without talking about the rest of what it can do you for. If you look at the documentation, for example, you'll see more than just context binding. Tomorrow, then, you'll get a bonus chapter that shows you how to use .bind for partial function application - supplying a partial list of parameters to a function so that they don't have to be specified later!

Want To Build Your Own .bind?

If you'd like to see how you can build your own version of .bind, how that would compare to Underscore.js' version of a .bind method, and go through the rest of these rules in a visual manner, seeing all of the code run through a browser, checkout my WatchMeCode screencast on JavaScript Context[4]. As I said before, it's an older episode that doesn't cover NodeJS, but it's well worth watching. If you're not yet a subscriber to WatchMeCode, though, check the "Thank You" chapter at the end of the book for a discount code.

[4] https://sub.watchmecode.net/javascript-context/

Bonus!

In yesterday's email, I showed you how to use .bind to bind the context of a function to an object of your choice. Using the .bind call also allows you to specify a list of parameters for the original function call to use. This is commonly referred to as "partial function application", as you are applying a partial list of parameters to the function, for later use.

Partial Function Application

To use partial application, you need to first bind the function to a context. After that, passing in additional parameters to the .bind call with hold those parameters and apply them when the bound function is called.

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

//bind and partially apply the add function
//ensuring the "a" parameter is always 1
var addOne = add.bind(null, 1);

//call the addOne method, supplying the
//2nd paramter for the original function
var result = addOne(2);

console.log(result); //=> 3
```

In the above code, a generic add method is set up to add two parameters. Just below that, the add function is bound to a null context (since this is never used in the function) and has a value of 1 stored as the first parameter. The result of this .bind call is a function that takes one parameter and applies it to the original function along with the specified context and parameter passed to the .bind call.

Later, when the addOne method is called, a single parameter is passed to it. Since the .bind call provided the first parameter for the original function, calling addOne(2) passes the value of 2 to the b parameter of the original add function. The result is then logged, with the expected value of 3.

This quick tip can be used in some fun and interesting ways. Try it out for yourself and see where it might be fun and might be dangerous to use in your app!

self = this;

In this email series, I've talked about how you can manage the value of "this" inside of nearly any function that you write. These techniques are powerful, but sometimes they are unnecessary. Sometimes you need something a little more simple than what I've shown, here. In those cases, I like to fall back to simple variable assignment, to grab on to the value of "this".

The most common example of this is in callback methods, like this:

```
var myObj = function(){
  doStuff: function(){

    // grab a reference to "this"
    var self = this;

    // inline callback function
    someOtherThing.doMoreStuff("some value", function(data){

      // call anotherThing, referencing "self" which points to "this" from above
      self.anotherThing(data);

    });
  },

  anotherThing: function(data){
    // ...
    // access to "this" is maintained as "myObj"
    // due to the dot-notation method invocation above
  }
}
```

Of course the variable name "self" is not important. The variable could be "that" or "what" or "batman" or whatever you want. The important point here, is to grab a reference to "this" and then use that reference inside of the callback function. This is yet another powerful technique that can greatly simplify code when you are dealing with inline callbacks and other situations where closures are appropriate.

FWIW, I don't really consider this to be part of the rules of managing "this", which is why it didn't get a full day of it's own. This is really just using variables and closures in a clever way, not managing the value of "this" at all. But it's still a technique you should be aware of and use when you need to.

Closures? What?

If you're interested in learning more about closures when you get the WatchMeCode discount code tomorrow, check out my episodes on Variable Scope in JavaScript, and JavaScript Zombies. Both of these cover closures and how to deal with them effectively.

Thank You

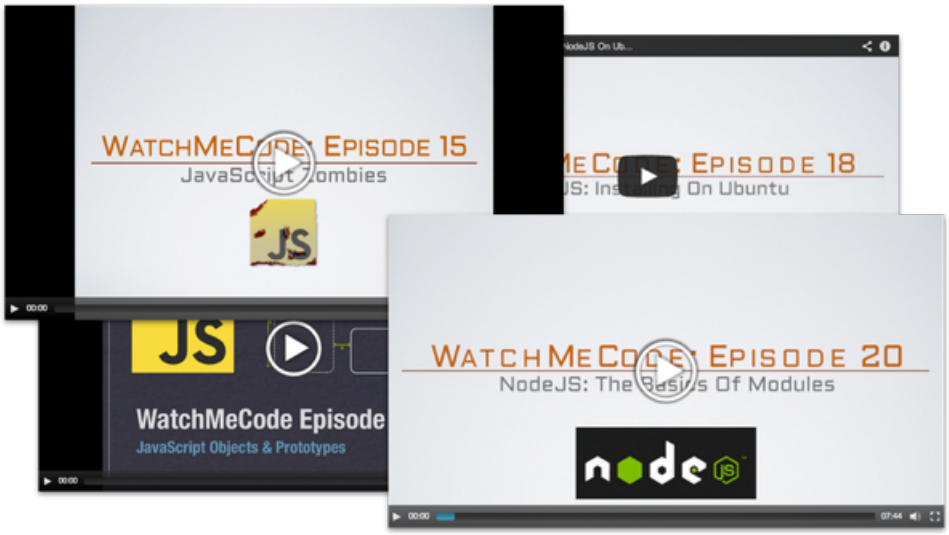
First off, I want to say congratulations on getting through this epic course on mastering JavaScript's "this". It may have been a rough ride, and you may be left with a few questions or points of confusion. Don't worry about this, though. It's normal and it's completely understandable. There's a lot of information to absorb in this course and it will take a little bit of time and practice to really understand it all.

Secondly, I want to say thank you! This has been an incredible journey for me, to re-write and re-build material that I have previously strewn about the internet in various places and forms. I'm extremely thankful that you've come along this journey with me, and I hope that you've both enjoyed it and learned from it.

As a way of saying thanks for sticking with it, and to help you further your JavaScript education, I have a parting offer for you: a discount of 35% off a monthly subscription to my WatchMeCode screencast service!

WatchMeCode: Byte-Sized JavaScript Screencasts, Delivered Weekly

WatchMeCode is the premiere screencast subscription site for learning the latest and greatest in JavaScript. I cover everything from the language fundamentals, as I mentioned throughout this short book, to the latest tools, techniques and tricks to make the most out of your JavaScript development experience.



And for completing this epic journey of mastering JavaScript's "this", I'm offering you a chance to get the entire catalog of screencasts at WatchMeCode for $9/mo - that's 35% off the normal price of $14/mo!

The Discount And Link

Just follow this link below and use the discount code provided. You'll get complete access to every JavaScript screencast that I offer, including the one on Objects and Prototypes. This is the screencast where I introduce the core mechanics of object oriented software development in JavaScript!

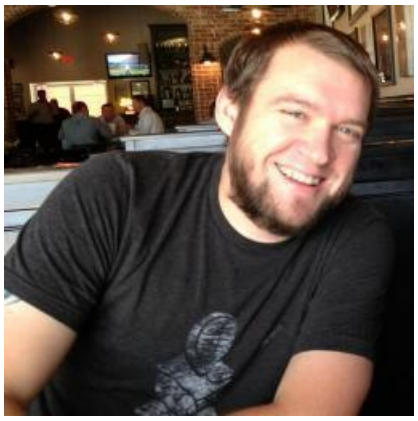
- **Head to this URL**: WatchMeCode.net/masteringthis[5]
- **Use this discount code**: masteringthis
- **Get a discounted subscription**: $9/mo

Thanks again, and enjoy both the book and the screencasts!

-Derick

[5] https://sub.watchmecode.net/masteringthis

About Derick



Hello, my name is Derick Bailey.

I'm a software developer and entreprenuer, a consultant, screencaster, blogger, speaker, trainer and more. I've been working professionally in software development since the late 90's and have been writing code since the late 80's.

My career has spanned many different tools, technologies, platforms and languages. I began working with JavaScript in Netscape 2.0, and found it to be both fun and powerful at the time. For the next 10 years, I had a love/hate relationship with the language, until I was introduced to Backbone.js in mid 2011. I almost immediately fell in love with Backbone, as I saw the potential for large scale, event-driven, composite applications being built in JavaScript with it. I've spent my time since then working with Backbone, blogging about it, training other developers to work on it, and building many different add-ons and plugins.

You can find those writings, plus much more, at my blog. I also produce screencasts, provide information about my consulting services, and more, through the following websites:

- My Company: MutedSolutions.com[6]
- My Blog: DerickBailey.com[7]
- Screencasts (free and paid): WatchMeCode.net[8]
- Open Source Projects: GitHub.com/DerickBailey[9]

If you have any questions, comments or concerns, you can contact me at the following:

[6] http://mutedsolutions.com
[7] http://derickbailey.com
[8] http://watchmecode.net
[9] http://github.com/derickbailey

- Email: derick@mutedsolutions.com[10]
- Twitter: @derickbailey[11]

Or contact me about the book, specifically:

- Email: backboneplugins@mutedsolutions.com[12]
- Twitter: @BackbonePlugins[13]

[10] mailto:derick@mutedsolutions.com
[11] http://twitter.com/derickbailey
[12] mailto:backboneplugins@mutedsolutions.com
[13] http://twitter.com/backboneplugins