JavaScript's Apply, Call, and Bind Methods are Essential for JavaScript Professionals

JULY. 10 2013 103

Prerequisite:
— Understand JavaScript's "this" With Ease, and Master It.
— JavaScript Objects
— Understand JavaScript Closures
(This is an intermediate to advanced topic)

Duration: About 40 minutes.

menu

Functions are objects in JavaScript, as you should know by now, if you have read any of the prerequisite articles. And as objects, functions have methods, including the powerful *Apply*, *Call*, and *Bind* methods. On the one hand, *Apply* and *Call* are nearly identical and are frequently used in JavaScript for borrowing methods and for setting the *this* value explicitly. We also use Apply for variable-arity functions; you will learn more about this in a bit.

On the other hand, we use Bind for setting the *this* value in methods and for currying functions.

We will discuss every scenario in which we use these three methods in JavaScript. While *Apply* and *Call* come with ECMAScript 3 (available on IE 6, 7, 8, and modern browsers), ECMAScript 5 (available on only modern browsers) added the Bind method. These 3 *Function methods* are workhorses and sometimes you absolutely need one of them. Let's begin with the *Bind* method.

Table of Contents

JavaScript's Bind Method

We use the *Bind ()* method primarily to call a function with the *this* value set explicitly. It other words, *bind ()* allows us to easily set which specific object will be bound to *this* when a function or method is invoked.

This might seem relatively trivial, but often the *this* value in methods and functions **must** be set explicitly when you need a specific object bound to the function's *this* value.

The need for *bind* usually occurs when we use the *this* keyword in a method and we call that method from a receiver object; in such cases, sometimes *this* is not bound to the object that we expect it to be bound to, resulting in errors in our applications. Don't worry if you don't fully comprehend the preceding sentence. It will become clear like teardrop in a moment.

Before we look at the code for this section, we should understand the *this* keyword in JavaScript. If you don't already understand *this* in JavaScript, read my article, Understand JavaScript's "this" With Clarity, and Master It. If you don't understand *this* well, you will have trouble

understanding some of the concepts discussed below. In fact, many of the concepts regarding setting the "this" value that I discuss in this article I also discussed in the *Understand JavaScript's "this"* article.

JavaScript's Bind Allows Us to Set the *this* Value on Methods

When the button below is clicked, the text field is populated with a random name.

```
//          <button>Get Random Person</button>
//       <input type="text">


var user = {
    data       :[
        {name:"T. Woods", age:37},
        {name:"P. Mickelson", age:43}
    ],
    clickHandler:function (event) {
        var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number between 0 and 1

        // This line is adding a random person from the data array to the text field
        $ ("input").val (this.data[randomNum].name + " " + this.data[randomNum].age);
    }

}

// Assign an eventHandler to the button's click event
$ ("button").click (user.clickHandler);

```

When you click the button, you get an error because *this* in the clickHandler () method is bound to the button HTML element, since that is the object that the clickHandler method is executed on.

This particular problem is quite common in JavaScript, and JavaScript frameworks like Backbone.js and libraries like jQuery automatically do the bindings for us, so that *this* is always bound to the object we expect it to be bound to.

To fix the problem in the preceding example, we can use the *bind* method thus:
Instead of this line:

```
$ ("button").click (user.clickHandler);
```

We simply have to bind the clickHandler method to the *user* object like this:

```
$ ("button").click (user.clickHandler.bind (user));
```

Consider this other way to fix the *this* value: You can pass an anonymous callback function to the click () method and jQuery will bind *this* inside the anonymous function to the button object.

Because ECMAScript 5 introduced the Bind method, it (Bind) is unavailable in IE < 9 and Firefox 3.x. Include this Bind implementation in your code, if you are targeting older browsers:

```
// Credit to Douglas Crockford for this bind method
if (!Function.prototype.bind) {
    Function.prototype.bind = function (oThis) {
        if (typeof this !== "function") {
            // closest thing possible to the ECMAScript 5 internal IsCallable function
            throw new TypeError ("Function.prototype.bind - what is trying to be bound is not callab
        }

        var aArgs = Array.prototype.slice.call (arguments, 1),
            fToBind = this,
            fNOP = function () {
            },
            fBound = function () {
                return fToBind.apply (this instanceof fNOP && oThis
                    ? this
                    : oThis,
                    aArgs.concat (Array.prototype.slice.call (arguments)));
            };

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP ();

```

```
            return fBound;
        };
    }

```

Let's continue with the same example we used above. The *this* value is also bound to another object if we assign the method (where *this* is defined) to a variable. This demonstrates:

```
// This data variable is a global variable
var data = [
    {name:"Samantha", age:12},
    {name:"Alexis", age:14}
]

var user = {
    // local data variable
    data    :[
        {name:"T. Woods", age:37},
        {name:"P. Mickelson", age:43}
    ],
    showData:function (event) {
        var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number between 0 and 1

        console.log (this.data[randomNum].name + " " + this.data[randomNum].age);
    }

}

// Assign the showData method of the user object to a variable
var showDataVar = user.showData;

showDataVar (); // Samantha 12 (from the global data array, not from the local data array)

```

When we execute the showDataVar () function, the values printed to the console are from the global data array, not the data array in the user object. This happens because showDataVar () is

executed as a global function and use of this inside showDataVar () is bound to the global scope, which is the window object in browsers.

Again, we can fix this problem by specifically setting the "this" value with the bind method:

```
1	// Bind the showData method to the user object
2	var showDataVar = user.showData.bind (user);
3
4	// Now the we get the value from the user object because the this keyword is bound to the user
5	showDataVar (); // P. Mickelson 43
6
```

Bind () Allows us to Borrow Methods

In JavaScript, we can pass functions around, return them, borrow them, and the like. And the bind () method makes it super easy to borrow methods.

Here is an example using bind () to borrow a method:

```
1	// Here we have a cars object that does not have a method to print its data to the console
2	var cars = {
3	    data:[
4	        {name:"Honda Accord", age:14},
5	        {name:"Tesla Model S", age:2}
6	    ]
7
8	}
9
10	// We can borrow the showData () method from the user object we defined in the last example.
11	// Here we bind the user.showData method to the cars object we just created.
12	cars.showData = user.showData.bind (cars);
13	cars.showData (); // Honda Accord 14
14
15
```

One problem with this example is that we are adding a new method (showData) on the cars object and we might not want to do that just to borrow a method because the cars object might already have a property or method name *showData*. We don't want to overwrite it accidentally. As we will see in our discussion of *Apply* and *Call* below, it is best to borrow a method using either the *Apply* or *Call* method.

JavaScript's Bind Allows Us to Curry a Function

Function **Currying**, also known as *partial function application*, is the use of a function (that accept one or more arguments) that returns a new function with some of the arguments already set. The function that is returned has access to the stored arguments and variables of the outer function. This sounds way more complex than it actually is, so let's code.

Let's use the *bind ()* method for currying. First we have a simple greet () function that accepts 3 parameters:

```
function greet (gender, age, name) {
    // if a male, use Mr., else use Ms.
    var salutation = gender === "male" ? "Mr. " : "Ms. ";

    if (age > 25) {
        return "Hello, " + salutation + name + ".";
    }
    else {
        return "Hey, " + name + ".";
    }
}

```

And we use the bind () method to curry (preset one or more of the parameters) our greet () function. The first argument of the bind () method sets the *this* value, as we discussed earlier:

```
// So we are passing null because we are not using the "this" keyword in our greet function.
var greetAnAdultMale = greet.bind (null, "male", 45);

greetAnAdultMale ("John Hartlove"); // "Hello, Mr. John Hartlove."
```

```
5
6       var greetAYoungster = greet.bind (null, "", 16);
7       greetAYoungster ("Alex"); // "Hey, Alex."
8       greetAYoungster ("Emma Waterloo"); // "Hey, Emma Waterloo."
9
```

When we use the bind () method for currying, all the parameters of the greet () function, except the last (rightmost) argument, are preset. So it is the rightmost argument that we are changing when we call the new functions that were curried from the greet () function. Again, I discuss currying at length in a separate blog post, and you will see how we can easily create very powerful functions with Currying and Compose, two Functional JavaScript concepts.

So, with the bind () method, we can explicitly set the this value for invoking methods on objects, we can borrow
and copy methods, and assign methods to variable to be executed as functions. And as outlined in the Currying Tip
earlier,
you can use bind for currying.

JavaScript's Apply and Call Methods

The **Apply** and **Call** methods are two of the most often used Function methods in JavaScript, and for good reason: they allow us to borrow functions and set the *this* value in function invocation. In addition, the *apply* function in particular allows us to execute a function with an array of parameters, such that each parameter is passed to the function individually when the function executes—great for **variadic** functions; a *variadic* function takes varying number of arguments, not a set number of arguments as most functions do.

Set the *this* value with Apply or Call

Just as in the *bind ()* example, we can also set the *this* value when invoking functions by using the Apply or Call methods. The first parameter in the call and apply methods set the this value to the object that the function is invoked upon.

Here is a very quick, illustrative example for starters before we get into more complex usages of Apply and Call:

```
// global variable for demonstration
var avgScore = "global avgScore";

//global function
function avg (arrayOfScores) {
    // Add all the scores and return the total
    var sumOfScores = arrayOfScores.reduce (function (prev, cur, index, array) {
        return prev + cur;
    });

    // The "this" keyword here will be bound to the global object, unless we set the "this" with
    this.avgScore = sumOfScores / arrayOfScores.length;
}

var gameController = {
    scores  :[20, 34, 55, 46, 77],
    avgScore:null
}

// If we execute the avg function thus, "this" inside the function is bound to the global windo
avg (gameController.scores);
// Proof that the avgScore was set on the global window object
console.log (window.avgScore); // 46.4
console.log (gameController.avgScore); // null

// reset the global avgScore
avgScore = "global avgScore";

// To set the "this" value explicitly, so that "this" is bound to the gameController,
// We use the call () method:
avg.call (gameController, gameController.scores);

console.log (window.avgScore); //global avgScore
console.log (gameController.avgScore); // 46.4

```

```
36
```

Note that the first argument to call () sets the *this* value. In the preceding example, it is set to
the *gameController* object. The other arguments after the first argument are passed as parameters to the
avg () function.

The apply and call methods are almost identical when setting the *this* value except that you pass the function parameters to apply () as an array, while you have to list the parameters individually to pass them to the call () method. More on this follows. Meanwhile, the apply () method also has another feature that the call () method doesn't have, as we will soon see.

Use Call or Apply To Set *this* in Callback Functions

I borrowed this little section from my article, Understand JavaScript Callback Functions and Use Them.

```
1  // Define an object with some properties and a method
2  // We will later pass the method as a callback function to another function
3  var clientData = {
4  id: 094545,
5  fullName: "Not Set",
6  // setUserName is a method on the clientData object
7  setUserName: function (firstName, lastName)  {
8  // this refers to the fullName property in this object
9  this.fullName = firstName + " " + lastName;
10 }
11 }
```

```
1    function getUserInput (firstName, lastName, callback, callbackObj) {
2       // The use of the Apply method below will set the "this" value to callbackObj
3       callback.apply (callbackObj, [firstName, lastName]);
4    }
5
6
```

The *Apply* method sets the *this* value to callbackObj. This allows us to execute the *callback* function with the *this* value set explicitly, so the parameters passed to the callback function will be set on the clientData object:

```
1    // The clientData object will be used by the Apply method to set the "this" value
2    getUserInput ("Barack", "Obama", clientData.setUserName, clientData);
3    // the fullName property on the clientData was correctly set
4    console.log (clientData.fullName); // Barack Obama
5
```

The Apply, Call, and Bind methods are all used to set the *this* value when invoking a method, and they do it in slightly different ways to allow use direct control and versatility in our JavaScript code. The this value in JavaScript is as important as any other part of the language, and we have the 3 aforementioned methods are the essential tools to setting and using *this* effectively and properly.

Borrowing Functions with Apply and Call (A Must Know)

The most common use for the *Apply* and *Call* methods in JavaScript is probably to borrow functions. We can borrow functions with the *Apply* and *Call* methods just as we did with the *bind* method, but in a more versatile manner.

Consider these examples:

- **Borrowing Array Methods**

  Arrays come with a number of useful methods for iterating and modifying arrays, but unfortunately, *Objects* do not have as many native methods. Nonetheless, since an Object can be expressed in a manner similar to an Array (known as an array-like object), and most important, because all of the Array methods are generic (except *toString* and *toLocaleString*), we can borrow Array methods and use them on objects that are array-like.

  An **array-like** object is an object that has its keys defined as non-negative integers. It is best to specifically add a *length* property on the object that has the length of the

object, since the a *length* property does not exist on objects it does on Arrays.

I should note (for clarity, especially for new JavaScript developers) that in the following examples, when we call **Array.prototype** , we are reaching into the Array object and on its prototype (where all its methods are defined for inheritance). And it is from there—the source—that we are borrowing the Array methods. Hence the use of code like *Array.prototype.slice*—the slice method that is defined on the Array prototype.

Let's create an array-like object and borrow some array methods to operate on the our array-like object. Keep in mind the array-like object is a real object, it is not an array at all:

```
// An array-like object: note the non-negative integers used as keys
var anArrayLikeObj = {0:"Martin", 1:78, 2:67, 3:["Letta", "Marieta", "Pauline"], length:4 }

```

Now, if wish to use any of the common Array methods on our object, we can:

```
// Make a quick copy and save the results in a real array:
// First parameter sets the "this" value
var newArray = Array.prototype.slice.call (anArrayLikeObj, 0);

console.log (newArray); // ["Martin", 78, 67, Array[3]]

// Search for "Martin" in the array-like object
console.log (Array.prototype.indexOf.call (anArrayLikeObj, "Martin") === -1 ? false :

// Try using an Array method without the call () or apply ()
console.log (anArrayLikeObj.indexOf ("Martin") === -1 ? false : true); // Error: Objec

// Reverse the object:
console.log (Array.prototype.reverse.call (anArrayLikeObj));
// {0: Array[3], 1: 67, 2: 78, 3: "Martin", length: 4}

// Sweet. We can pop too:
console.log (Array.prototype.pop.call (anArrayLikeObj));
console.log (anArrayLikeObj); // {0: Array[3], 1: 67, 2: 78, length: 3}
```

```
20
21          // What about push?
22          console.log (Array.prototype.push.call (anArrayLikeObj, "Jackie"));
23          console.log (anArrayLikeObj); // {0: Array[3], 1: 67, 2: 78, 3: "Jackie", length: 4}
24
```

We get all the great benefits of an object and we are still able to use Array methods on our object, when we setup our object as an array-like object and borrow the Array methods. All of this is made possible by the virtue of the *call* or *apply* method.

The **arguments** object that is a property of all JavaScript functions is an array-like object, and for this reason, one of the most popular uses of the call () and apply () methods is to extract the parameters passed into a function from the *arguments* object.

Here is an example I took from the Ember.js source, with comments I added:

```
1           function transitionTo (name) {
2              // Because the arguments object is an array-like object
3              // We can use the slice () Array method on it
4              // The number "1" parameter means: return a copy of the array from index 1 to th
5
6              var args = Array.prototype.slice.call (arguments, 1);
7
8              // I added this bit so we can see the args value
9              console.log (args);
10
11             // I commented out this last line because it is beyond this example
12             //doTransition(this, name, this.updateURL, args);
13          }
14
15          // Because the slice method copied from index 1 to the end, the first item "contact"
16          transitionTo ("contact", "Today", "20"); // ["Today", "20"]
17
```

The *args* variable is a real array. It has a copy of all the parameters passed to the transitionTo function.

From this example, we learn that a quick way to get all the arguments (as an array) passed to a function is to do:

```
// We do not define the function with any parameters, yet we can get all the argumer
function doSomething () {
    var args = Array.prototype.slice.call (arguments);
    console.log (args);
}

doSomething ("Water", "Salt", "Glue"); // ["Water", "Salt", "Glue"]

```

We will discuss how to use the *apply* method with the *arguments* array-like object again for variadic functions. More on this later.

Borrowing String Methods with Apply and Call

Like the preceding example, we can also use *apply ()* and *call ()* to borrow String methods. Since Strings are immutable, only the non-manipulative arrays work on them, so you cannot use *reverse*, *pop* and the like.

Borrow Other Methods and Functions

Since we are borrowing, lets go all in and borrow from our own custom methods and functions, not just from *Array* and *String*:

```
var gameController = {
    scores  :[20, 34, 55, 46, 77],
    avgScore:null,
    players :[
        {name:"Tommy", playerID:987, age:23},
        {name:"Pau", playerID:87, age:33}
    ]
}

var appController = {
    scores  :[900, 845, 809, 950],
    avgScore:null,
    avg     :function () {
```

```
14
15              var sumOfScores = this.scores.reduce (function (prev, cur, index, array) {
16                  return prev + cur;
17              });
18
19              this.avgScore = sumOfScores / this.scores.length;
20          }
21      }
22
23      // Note that we are using the apply () method, so the 2nd argument has to be an ar
24      appController.avg.apply (gameController);
25      console.log (gameController.avgScore); // 46.4
26
27      // appController.avgScore is still null; it was not updated, only gameController.avgSc
28      console.log (appController.avgScore); // null
29
```

Sure, it is just as easy, even recommended, to borrow our own custom methods and functions. The gameController object borrows the appController object's avg () method. The "this" value defined in the avg () method will be set to the first parameter—the gameController object.

You might be wondering what will happen if the original definition of the method we are borrowing changes. Will the borrowed (copied) method change as well, or is the copied method a full copy that does not refer back to the original method? Let's answer these questions with a quick, illustrative example:

```
1   appController.maxNum = function () {
2       this.avgScore = Math.max.apply (null, this.scores);
3   }
4
5   appController.maxNum.apply (gameController, gameController.scores);
6   console.log (gameController.avgScore); // 77
7
```

As expected, if we change the original method, the changes are reflected in the borrowed instances of that method. This is expected for good reason: we never made a full copy of the method, we simply borrowed it (referred directly to its current implementation).

Use Apply () to Execute Variable-Arity Functions

To wrap up our discussion on the versatility and usefulness of the Apply, Call, and Bind methods, we will discuss a neat, little feature of the Apply method: execute functions with an array of arguments.

We can pass an array with of arguments to a function and, by virtue of using the apply () method, the function will execute the
items in the array as if we called the function like this:

```
createAccount (arrayOfItems[0], arrayOfItems[1], arrayOfItems[2], arrayOfItems[3]);
```

This technique is especially used for creating **variable-arity**, also known as **variadic** functions.
These are functions that accept any number of arguments instead of a fixed number of arguments. The *arity* of a function specifies the number of arguments the function was defined to accept.

The *Math.max()* method is an example of a common variable-arity function in JavaScript:

```
// We can pass any number of arguments to the Math.max () method
console.log (Math.max (23, 11, 34, 56)); // 56
```

But what if we have an array of numbers to pass to Math.max? We cannot do this:

```
var allNumbers = [23, 11, 34, 56];
// We cannot pass an array of numbers to the the Math.max method like this
console.log (Math.max (allNumbers)); // NaN
```

This is where the *apply ()* method helps us execute variadic functions. Instead of the above, we have to pass the array of numbers using apply () thus:

```
var allNumbers = [23, 11, 34, 56];
// Using the apply () method, we can pass the array of numbers:
```

```
3    console.log (Math.max.apply (null, allNumbers)); // 56
```

As we have learned earlier, the fist argument to apply () sets the "this" value, but "this" is not used in the Math.max () method, so we pass null.

Here is an example of our own variadic function to further illustrate the concept of using the apply () method in
this capacity:

```
1    var students = ["Peter Alexander", "Michael Woodruff", "Judy Archer", "Malcolm Khan"];
2
3    // No specific parameters defined, because ANY number of parameters are accepted
4    function welcomeStudents () {
5        var args = Array.prototype.slice.call (arguments);
6
7        var lastItem = args.pop ();
8        console.log ("Welcome " + args.join (", ") + ", and " + lastItem + ".");
9    }
10
11   welcomeStudents.apply (null, students);
12   // Welcome Peter Alexander, Michael Woodruff, Judy Archer, and Malcolm Khan.
```

Final Words

The Call, Apply, and Bind methods are indeed workhorses and should be part of your JavaScript repertoire for setting the this *value* in functions, for creating and executing variadic functions, and for borrowing methods and functions. As a JavaScript developer, you will likely encounter and use these functions time and again. So be sure you understand them well.

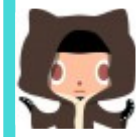
Be Good. Imagine. Create.

Posted in: 16 Important JavaScript Concepts, Advanced javaScript, JavaScript / Tagged: Advanced JavaScript, Apply, Bind, Call, Intermediate Javascript, Learn JavaScript

Richard

Thanks for your time; please come back soon. Email me here: javascriptissexy at gmail email, or use the contact form.

103 Comments



ECOVOX

July 15, 2013 at 1:46 pm / Reply

The bind from douglas crockford example is really hard to understand.

Richard Bovell (Author)

July 17, 2013 at 3:22 pm / Reply

I agree. Crockford is a complex guy. Later (when I have a bit of time), I will expound on his bind method to make it clearer.

Ecovox

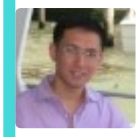
July 18, 2013 at 1:33 am / Reply

Do you recommend JS:The good part? I think it like the book that tech you how to write code like Mr.Douglas Crockford.

Richard Bovell (Author)

July 31, 2013 at 12:43 am / Reply

That is a good book, but not to use as your first book. It should be your second or third book, after you have already learned the basics and understand JavaScript well.



Angelo

July 17, 2013 at 10:40 am / Reply

You are awesome as usual! Thanks for your time!

Richard Bovell (Author)

July 17, 2013 at 3:25 pm / Reply

Thank you for reading the blog, and you are welcome.

Pat

July 17, 2013 at 5:51 pm / Reply

I've learned a lot since I stumbled on this blog, thank you! Keep sharing.

Richard Bovell (Author)

July 30, 2013 at 9:24 am / Reply

Great. I am very happy to hear that.

Robert

July 18, 2013 at 2:46 am / Reply

great post thanks richard!

Richard Bovell (Author)

July 31, 2013 at 12:43 am / Reply

Thank you, Robert.

Sri

July 23, 2013 at 1:03 pm / Reply

Ever since i have stumbled upon your blog, i must say that my JS concepts have improved a lot. You have unique ability to simplify things in such a way it is easy to grasp.

Keep up the good work.. May be someday you should author a book 🙂

Richard Bovell (Author)

July 31, 2013 at 1:58 am / Reply

Sri, you are wonderful. Thanks for the wonderful words of encouragement.

> *Keep up the good work.. May be someday you should author a book*

I am actually writing a book—more on it later 😉

Thomas

July 25, 2013 at 10:16 am / Reply

How come your articles dont appear in the "Javascript Weekly" news letter? Another well explained JS concept.

Richard Bovell (Author)

July 31, 2013 at 2:01 am / Reply

The OOP in JavaScript post was published in JavaScript Weekly. That was the only one, so I know what you mean.

I don't know the criteria for being selected, but you can send the editor a Tweet:
https://twitter.com/JavaScriptDaily

Thanks.

Ritesh

August 6, 2013 at 2:03 am / Reply

I learn something new every time i visit your blog. Thanks richard 🙂

Richard Bovell (Author)

August 6, 2013 at 8:56 pm / Reply

Thank you, Ritesh, and you are welcome 🙂

Emre Camasuvi

August 6, 2013 at 8:30 pm / Reply

Hi Richard,
I really love reading your articles and appreciate your work in here; even simple examples guide n00bies very well. Thank you and wish to see more of them.
I've got a question, in your greetAYoungster function you didnt define a second parameter so it should return "Hey, Ms. Alex." instead of "Hey, Alex." because "" (empty string) is falsey i guess.

Richard Bovell (Author)

August 6, 2013 at 8:56 pm / Reply

Emre, your suggestion that falsy will be the result from a nonexistent (or empty) parameter is correct. But in the example code, if the person's age is 25 or younger, the *salutation* variable is not being used at all, so the returned result will always be without "Mr." or "Ms."

Here is a working example on JSbin:
http://jsbin.com/okohut/2/edit

Emre Camasuvi

August 6, 2013 at 9:01 pm / Reply

Oops i missed that, my mistake.
Thank you! 🙂

Martin Felsin

August 19, 2013 at 5:26 am / Reply

Could you please provide example using Microsoft Visual Basic 6. I think VB6 is more performant and powerful than the Java scripting language

Ecovox

August 19, 2013 at 8:43 am / Reply

hey this is js blog for web dev not ms vb.

yougen

August 23, 2013 at 5:10 am / Reply

thanks for this blog. When I read "borrow function", I suddenly realize what Array.prototype.slice(arguments) really do, which confused me a long time. Ye, borrow is the key 🙂

yougen

August 23, 2013 at 5:12 am / Reply

sorry, not Array.prototype.slice(argument), but Array.prototype.slice.apply(argument)

Richard Bovell (Author)

August 23, 2013 at 11:34 am / Reply

You are very welcome, Yougen. I am glad this article was helpful for you.

yougen

August 23, 2013 at 11:55 pm / Reply

Hi Richard,
I find a tricky part when borrow method, I modify the code as following:

```
var gameController = {
scores: [20, 10, 60, 50]
}

var appController = {
scores: [900, 300, 600, 400],
avgScore: null,
maxNum: function(){
this.avgScore = Math.max.apply(null,
this.scores);
}
}

appController.maxNum.apply(gameController,
gameController.scores);
console.log(gameController.avgScore);// 60
```

Since gameController does NOT have avsScore property,
but it still works and get the result 60.

charles

September 27, 2013 at 11:18 am / Reply

Currying is not the same thing as partial function application.

Richard Of Stanley (Author)

October 1, 2013 at 2:47 pm / Reply

No, they are not, but many books on functional programming do use the "partial application" for currying, since currying is a type of partial application.

charles

October 1, 2013 at 3:14 pm / Reply

It's probably closer to correct to say that partial function application is a specialization of currying.

Vinod

October 14, 2013 at 7:44 am / Reply

Similar post can be found here

http://simplr.co.in/blog/learning-the-chaining-pattern-with-javascript-constructors/

watertype

October 16, 2013 at 6:32 pm / Reply

Hello Richard!

Thank you for illustrating the JS "ABC's" so clearly. This explanation rounded out my understanding of each respective methods coming from Reginald Braithwaite's JavaScript Allonge (definitely recommend if you haven't checked it out already!).

I found a code 'hiccup' in one of your examples. As a note, I was working through most of the examples using the REPL available to Node v.10.20 and the Google Chrome Console Version 30.0.1599.101.

Within the section 'Set the this value with Apply or Call,' under your first code example, your global function 'avg' is missing the 'return' keyword before 'this.avgScore = sumOfScores / arrayOfScores.length; .' Without the 'return' keyword, I had a bunch of 'undefined' values pop up.

May you please update the code sample to include 'return' for future readers? If the function should work without the 'return,' I'd like to know more about why one could leave it out. 🙂

Again, much thanks. Please keep up the great work on your blog!

#!/watertype

Richard Of Stanley (Author)

October 23, 2013 at 3:10 pm / Reply

HI Watertype,

Thanks for taking the time to share this with the rest of the readers. It is very helpful. I take it you are referring to this function:

```
1
2       function avg (arrayOfScores) {
3          // Add all the scores and return the total
4          var sumOfScores = arrayOfScores.reduce (function (prev, cur, inde
5             return prev + cur;
6          });
```

```
// *************
        this.avgScore = sumOfScores / arrayOfScores.length;
    }

    var gameController = {
        scores  :[20, 34, 55, 46, 77],
        avgScore:null
    }

```

To be clear, are you saying I need a return stament where I have this // *************

If so, I can't have the return statement there, or else the last line will never be reached. I am not sure why the code example had some error with REPL.

What happens if you add the return statement after the line:
this.avgScore = sumOfScores / arrayOfScores.length;
// Put return here

Does that work?

watertype

October 23, 2013 at 4:07 pm / Reply

Hello again Richard,

Thanks for the reply!
I realized that my interpretation of the intent of that code example was incorrect. Originally, I thought that the avg function was to immediately return the average value of a given array (if say calling console.log upon it). Looking at it again, the avg function was meant to update the `this` object's avgScore, and then checking it with console.log().

I realized that the example was meant to show the `.call` method's usage in different `this` contexts. So on my part, it was actually me with the code hiccup, haha!

In all, it's clear to me now. Again, thanks!

#!/watertype

Richard Of Stanley (Author)

October 23, 2013 at 4:16 pm / Reply

You are welcome, and thanks for following up.

Ruslan

December 11, 2013 at 11:29 am / Reply

Great article! it's really clear and comprehensive with a few examples you've given. I've been searching for this kind of article to nail down the bind() method and additionally got 2 other methods;) Cool, thank you Richard!

Tim

January 7, 2014 at 12:00 pm / Reply

Your articles are helpful and well explained. Thank you. But could I please ask you to remove any leading spaces from the lines in your code samples? (In future posts; I wouldn't ask you to edit all the old ones.) Your layout is so narrow that the extra spaces at the beginning of each line make nearly every line of code wrap. It makes it harder to read your code. — Thanks

Sabin

March 4, 2014 at 8:06 pm / Reply

I am confused about why the apply and call methods might be needed for getting the arguments.

Can't you do inside a function:
var args = arguments;
Instead of:
var args = Array.prototype.slice.call(arguments);

Roger T

April 3, 2014 at 1:30 am / Reply

Hi Richard, love this website! I'd be lost in a sea of confusing and disparate tutorials without it 🙂

Above you mention the following:

"Note that another way to fix the this value is that you can pass an anonymous callback function to the click () method and jQuery will bound this inside the anonymous function to the button object."

I'm not exactly sure what you mean here. I came up with the following, but wanted to double check that I had conceptualized it right:

$('button').click(function() {
var randomNum = Math.round(Math.random());
alert(user.data[randomNum].name);
})

Thanks again for the incredible site!!!

-r

Savitha Gowda

April 9, 2014 at 12:07 am / Reply

hi,

Can you please explain following concept: function currying at length in a separate blog post, and you will see how we can easily create very powerful

functions with Currying and Compose, two Functional JavaScript concepts.

which blog i should check?? where have you explained??
i'm waiting for that. i learnt javascript because your blogs only thanks a lot. also please explain me for classical inheritance and prototyple inheritance in javascript.

Mike

April 24, 2014 at 4:11 pm / Reply

Great blog. I'm very new to all of this, so apologies in advance if I'm just being dense, but I had some questions regarding your example of "borrowing" methods using the .bind() function, i.e.:

```
var cars = {
data:[
{name:"Honda Accord", age:14},
{name:"Tesla Model S", age:2}
]
}
cars.showData = user.showData.bind (cars);
cars.showData (); // Honda Accord 14
```

You correctly note that this is bad practice, because the cars object might already have a showData method that we would be accidentally overwriting.

However, I'm not sure this counts as "borrowing." Based on your later GameController example of borrowing, borrowing seems to be about invoking object A's method within the context of object B, whereas this is declaring new methods in Object B.

Thus, to make the above a correct example of borrowing, shouldn't it just end with:

```
user.showData.bind(cars)();
// must add () to the end, because bind() doesn't auto-invoke, unlike apply().
```

NOT:
cars.showData = user.showData.bind (cars);
cars.showData ();

In which case, is there even a problem using bind()() to borrow? Granted it can't take additional arguments like apply/call, so is more limited.

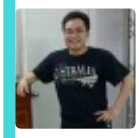
Thanks!

Leslie C

September 8, 2014 at 7:09 pm / Reply

Another thanks for these useful collection of posts.

For "apply" vs "call", I remember "A" is for "apply" and "array" 🙂



Binh Thanh Nguyen

November 3, 2014 at 12:47 pm / Reply

Thanks, nice post

Progr Ammer

November 9, 2014 at 7:22 am / Reply

Is the code for found correct? I don't get why it uses "oThis" both when "this" is already defined and "oThis" is not.

robert

November 18, 2014 at 7:28 am / Reply

Hi,

I am beginner in Javascript, and don't understand followig line:
cars.showData = user.showData.bind (cars);

Why did you do : 'user.showData.bind (cars)', is needed bind in this situation?

cars.showData method is invoked, not in the global context but in Object 'cars' context, so
cars.showData = user.showData; don't do the work?

I apologise in advance for my weak english.

Rick

November 23, 2014 at 8:19 am / Reply

Thank you for the explanation on borrowing methods. I just tried it in one of my scripts and it works great. It saved me from a tedious workaround. Thanks again!

Chris

December 4, 2014 at 8:46 pm / Reply

This explanation is just what I was looking for. There is btw not a single YouTube video which adequately explains this subject (hint hint …).

ShadyStego

January 5, 2015 at 12:26 am / Reply

Thanks for the great post!

I have a question on your illustration:

=====
appController.maxNum = function () {
this.avgScore = Math.max.apply (null, this.scores);
}

appController.maxNum.apply (gameController, gameController.scores);
console.log (gameController.avgScore); // 77
====

I understand how this example works. What I don't get is why this is the example to answer the question "what will happen if the original definition of the method we are borrowing changes.." . I was expecting an example where the original method "avg" would be changed after the first time it was borrowed. The illustration above was using a new method, so I'm a little confused about the relevance to the question.

Thanks in advance!

Melinda

January 29, 2015 at 12:14 pm / Reply

Hi! Wonderful blog post; thank you writing it! I have a question about the example given for currying function with the bind method.

In the function below, the arguments are listed in the order: gender, age, and name. So I assumed when the function was curried and saved to a variable, you would need to list the order of the arguments the same, meaning in var greetAnAdultMale = greet.bind (null, "male", 45); the arguments would be ("male", 45, null).

However, that is not the case in the example when greetAnAdultMale("John Hartlove") is called with a single parameter, it returns the function as "Hello, Mr. John Hartlove." Why is this? How does the JS interpreter know that when the function is called, the argument given corresponds to the name parameter and is not overwriting the first argument (gender)?

```
function greet (gender, age, name) {
// if a male, use Mr., else use Ms.
var salutation = gender === "male" ? "Mr. " : "Ms. ";

if (age > 25) {
return "Hello, " + salutation + name + ".";
}
else {
return "Hey, " + name + ".";
```

```
}
}
```

soundar

February 9, 2015 at 9:15 pm / Reply

after reading your posts i understood all things in js ,thanks alot.

Niko

February 23, 2015 at 8:34 pm / Reply

You are right that the following code creates (or potentially overwrites) a showData method on cars…
cars.showData = user.showData.bind (cars);
cars.showData (); // Honda Accord 14

However, you can easily use bind to borrow the showData method from users without creating it on cars by immediately calling it…
user.showData.bind (cars)();

gheorghe

March 7, 2015 at 10:18 am / Reply

the last example is not so good because the function could be called without the apply method, like so:
….
var args = Array.prototype.slice.apply (arguments[0]);
…..
welcomeStudents (students);

tommy

March 10, 2015 at 2:39 am / Reply

I have learn a lot from your posts.Thank you !

Mike Joyce

March 17, 2015 at 6:17 pm / Reply

Great article. Would be even better if you could explain the fundamental differences between bind, call and apply though.

jaga

March 28, 2015 at 2:40 pm / Reply

Superb post

Harish

April 30, 2015 at 11:23 am / Reply

This is the best site for javascript learning. Objects behaving like an array – Man! that's pure genius. Till yesterday, I thought arguments object can access only length property, but not anymore. 🙂
Thanks Richard. I am indulging this newly acquired knowledge.

Cheers
Harish

Seshu

May 6, 2015 at 10:10 am / Reply

You are Awesome. Thank You so much for the great articles.

Frederik Krautwald

May 27, 2015 at 8:57 am / Reply

Wouldn't it be more accurate to pass `undefined` as `thisArg` to the `bind` method for *partial functions* instead of passing `null`?

Jabir

June 11, 2015 at 6:32 am / Reply

Thanks Richard.

Almost all tutorial on apply and call seemed a little difficult to me. But, this article made everything easy for me to understand.

Esraa

July 20, 2015 at 11:03 pm / Reply

((Math.random () * 2 | 0) + 1) – 1; // random number between 0 and 1
I really wanna to know how does it work , what this operator (|) refer to ?!

snaghwaJung

September 11, 2015 at 3:16 am / Reply

what you worte is awesome and I Learn alot~!
thankyou~!

John

September 26, 2015 at 2:20 am / Reply

This was an immensely helpful post for me 😀 Took the time to read it all in one sitting and I'm very glad I did. The examples were really helpful and things were generally just written in a clear, concise way. Thank you so much for the great article 😀

A lot of the concepts covered here relate to things I've been reading about/practicing otherwise, and this sort of "completed" some of that

knowledge for me. It's a very cool feeling when that moment happens and you suddenly realize how far you've come from where you started.

Pavel

October 2, 2015 at 11:58 am / Reply

Can you please update your article and add one missing bit: call'ing ot apply'ing a bound function.
For example:

[code]
function do_something(a, b, callback){
// ... do something
callback.apply(null, [a, b]);
}
function callback(a, b){
console.log('callback called with:', a, b);
}
do_something(1, 2, callback);
do_something(1, 2, console.log.bind(console)); // bound callback passed
[/code]

Tarak

October 16, 2015 at 4:43 am / Reply

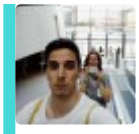
You are Awesome. Thank You so much for the great articles and time.

Jordan

October 19, 2015 at 2:02 pm / Reply

Thank you a lot for this explanation.

Yves



October 26, 2015 at 6:21 pm / Reply

Appreciate! Thanks, great explanation.

Noah

November 1, 2015 at 9:39 pm / Reply

Yes, if you don't need to slice the arguments (meaning, if you want ALL the arguments, instead of just the 2nd and 3rd argument, for example) you can use the code you posted.

Remember, Array.prototype.slice does this:

```
// Our good friend the citrus from fruits example
var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
var citrus = fruits.slice(1, 3);

// citrus contains ['Orange','Lemon']
```

Alligator

November 12, 2015 at 2:45 pm / Reply

What a great article. I really appreciate it and it helped clear up a lot of issues for me!

Anthony Trinh

November 13, 2015 at 2:11 pm / Reply

Hi,

In one of your examples in the Apply/Call section. Is the below a mistake?

avg.call (gameController, gameController.scores);

For this example, since gameController.scores is an array, should we be using apply instead of call?

Thanks,

Anthony Trinh

Deepak Chougule

November 25, 2015 at 9:11 am / Reply

Thanks for awesome article.. 🙂 I have one question.
When we borrow methods using call and apply, doesn't borrowed method added to the object for which we borrowed it?
like in bind, when we borrow method for particular object that method is added in that object.

sbs

January 31, 2016 at 4:04 am / Reply

Nice job!!! thanks!!

Tom

February 10, 2016 at 7:15 am / Reply

Thank you for such a wonderfully simple explanations of js concepts!
I am active js developer with almost 2 years of ex. but I felt that my knowledge is more practical than theoretical one (not so easy to speak about it) kind of. Am preparing for work interview and thought I should sort some things out in my head. Your blog has REALLY helped me to do all that. Thanks a ton Richard!

Ajay

February 12, 2016 at 11:29 am / Reply

Thanks Richard for wonderful article. Now, I could feel better with javascript 🙂

I have a doubt on function currying example. Below is my code snippet.
Question is in comment. Please help clarify.

```
$(document).ready(function() {
var showUser = userController.clickHandler.bind(userController);
showUser(0);

// I get error as explained above. This is fine.
// $('#btn').click(userController.clickHandler);

// Doubt: This works fine for me although i did not use bind to set this. 1 passed as argument is index.
// so, why I am not getting same error as above? How come passing 1 as argument is setting this object.
$('#btn').click(userController.clickHandler(1));
});

var userController = {
userData : [
{firstName: "Ajay", age : "30"},
{firstName: "sonu", age: "30"}
],
clickHandler : function(index) {
$('#id').val(this.userData[index].firstName + ' : ' + this.userData[index].age);
}
};
```

Jesse

March 13, 2016 at 9:33 pm / Reply

I always search for sites to recycle my knowledge and I must say that yours is one of the best I found. Really helped me to remember and even learn new stuffs.

Thank you.

Jesse

March 13, 2016 at 9:34 pm / Reply

I always search for sites to recycle my knowledge and I must say that yours is one of the best I found. Really helped me to remember and even learn new stuffs.

Thank you.

geetai

April 19, 2016 at 2:36 am / Reply

Hi Richard ,

Thanks for wonderful article , I cannot think of learning javascript through any other medium 🙂

I have one doubt in call example given above(illustrating how to set this to point to particular object), we have reset global variable :
// reset the global avgScore
avgScore = "global avgScore";

// To set the "this" value explicitly, so that "this" is bound to the gameController,
// We use the call () method:
avg.call (gameController, gameController.scores);

console.log (window.avgScore);
console.log (gameController.avgScore); // 46.4

But console.log (window.avgScore); this line still shows 46.4 in fiidle

Can you please explain me why so?

Faniry RAMANOHISOA

April 25, 2016 at 8:36 am / Reply

I learnt a lot from your post Richard, thank you very much for being awesome and sharing your knowledge to the world like you do!
Now I can continue learning more about functional programming in Javasri

Mooli

May 6, 2016 at 6:49 am / Reply

thanks again for great article but i still not 100% sure i understand when should i use call() and when .apply() ???

Rodrigo

October 6, 2016 at 11:49 am / Reply

So, I was searching for a good source of javascript articles to reach the next level of development as I think I am already a intermediate/advanced javascript programmer, and then I found out your site. I really loved it, dude.
Well explained articles, with a lot of advanced stuff that I really did not understand at all, now I can.
Thanks and keep doing this great job.

Richard Of Stanley (Author)

November 10, 2013 at 2:43 am / Reply

Thanks for your comment, Leah.

I am surprise that I didn't explicitly explain the difference between the 3 methods. I will take you advice and update the article accordingly, when time permits.

Trackbacks for this post

1. Understand JavaScript's "this" With Ease, and Master It | JavaScript is Sexy
2. Understand JavaScript's "this" With Clarity, and Master It | JavaScript is Sexy
3. 16 JavaScript Concepts JavaScript Professionals Must Know Well | JavaScript is Sexy
4. JavaScript's Apply, Call, and Bind Method...
5. JavaScript's Apply, Call, and Bind Method...
6. Clonando objetos en JavaScript | Bitácora del desarrollador

7. sexy.com | JavaScript is Sexy
8. Javascript: call() and apply() – Reserves of brain
9. Naked JavaScript | Pearltrees
10. Bind, Call, Apply method trong Javascrip. Phần 2 - Call và Apply method - BLOG CYO
11. JavaScript call vs bind vs apply | vasunagpal
12. Bind, Call and Apply in JavaScript - Technical Blogs : Technical Blogs
13. Understanding THIS in JavaScript. | Learning to code in spare time.
14. JavaScript's Apply, Call, and Bind Methods are Essential for JavaScript Professionals | JavaScript is Sexy | Paul de Wouters
15. Preparing Before Hack Reactor Begins | bash $ cat bitchblog
16. Javascript – Bind | anxtech
17. FrontEnd courses list | webappexpress
18. New folder | Pearltrees
19. Yet another Call vs Apply query - javascript
20. javascript - Cannot call method 'render' of undefined - CSS PHP
21. ES6 Arrow Functions: The New Fat & Concise Syntax in JavaScript
22. ES6 Arrow Functions: The New Fat & Concise Syntax in JavaScript | Codango.Com
23. call() a standard method on function objects | WILTK
24. Introduction to "this" | WILTK
25. JavaScript advanced Concepts - @Mohan Dere

Leave a Reply

Name | Email (hidden) | Website

Comment:

Submit Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

 JavaScript is Sexy About Contact Archive