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 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.

*// 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 Call or Apply​*

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 window object:​*

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​

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 must list the parameters individually to pass them to the call () method.

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 methods are the essential tools to setting and using *this* effectively and properly.

Call() Method

The **call()** method calls a function with a given this value and arguments provided individually.

A different this object can be assigned when calling an existing function. this refers to the current object, the calling object. With call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

function.call(thisArg, arg1, arg2, ...)

## Parameters

**thisArg**

The value of this provided for the call to a function. Note that this may not be the actual value seen by the method: if the method is a function in non-strictmode, null and undefined will be replaced with the global object and primitive values will be converted to objects.

**arg1, arg2, ...**

Arguments for the function.

## Return value

The result of calling the function with the specified **this** value and arguments

Examples

# Using call to chain constructors for an object

You can use call to chain constructors for an object, like Java. In the following example, the constructor for the Product object is defined with two parameters, name and price. Two other functions Food and Toy invoke Product passing this and name and price. Product initializes the properties name and price, both specialized functions define the category.

function Product(name, price) {

this.name = name;

this.price = price;

}

function Food(name, price) {

Product.call(this, name, price);

this.category = 'food';

}

function Toy(name, price) {

Product.call(this, name, price);

this.category = 'toy';

}

var cheese = new Food('feta', 5);

var fun = new Toy('robot', 40);

# Using call to invoke an anonymous function

In this purely constructed example, we create an anonymous function and use call to invoke it on every object in an array. The main purpose of the anonymous function here is to add a print function to every object, which is able to print the right index of the object in the array. Passing the object as this value was not strictly necessary, but is done for explanatory purpose.

var animals = [

{ species: 'Lion', name: 'King' },

{ species: 'Whale', name: 'Fail' }

];

for (var i = 0; i < animals.length; i++) {

(function(i) {

this.print = function() {

console.log('#' + i + ' ' + this.species + ': ' + this.name);

}

this.print();

}).call(animals[i], i);

}

# Using call to invoke a function and specifying the context for 'this'

In below example, when we will call greet the value of this will be bound to object i.

function greet() {

var reply = [this.person, 'Is An Awesome', this.role].join(' ');

console.log(reply);

}

var i = {

person: 'Douglas Crockford',

role: 'Javascript Developer'

};

greet.call(i);

*// Douglas Crockford Is An Awesome Javascript Developer*

Note:

While the syntax of this function is almost identical to that of call(), the fundamental difference is that call() accepts an **argument list**, while apply() accepts a **single array of arguments**.

apply() Method

The apply() method calls a function with a given this value, and arguments provided as an array (or an array-like object).

func.apply(thisArg, [argsArray])

You can assign a different this object when calling an existing function. this refers to the current object, the calling object. With apply, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

apply is very similar to call(), except for the type of arguments it supports. You use an arguments array instead of a list of arguments (parameters). With apply, you can also use an array literal, for example, *func*.apply(this, ['eat', 'bananas']), or an Array object, for example, func.apply(this, new Array('eat', 'bananas')).

You can also use arguments for the argsArray parameter. arguments is a local variable of a function. It can be used for all unspecified arguments of the called object. Thus, you do not have to know the arguments of the called object when you use the apply method. You can use arguments to pass all the arguments to the called object. The called object is then responsible for handling the arguments.

# Parameters

thisArg

The value of this provided for the call to func. Note that this may not be the actual value seen by the method: if the method is a function in non-strict mode code, null and undefined will be replaced with the global object, and primitive values will be boxed.

argsArray

An array-like object, specifying the arguments with which func should be called, or [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) if no arguments should be provided to the function.

Examples

# Using apply to chain constructors

You can use apply to chain [constructors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) for an object, similar to Java. In the following example we will create a global [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) method called construct, which will enable you to use an array-like object with a constructor instead of an arguments list.

Function.prototype.construct = function(aArgs) {

var oNew = Object.create(this.prototype);

this.apply(oNew, aArgs);

return oNew;

};

# Using apply and built-in functions

Clever usage of apply allows you to use built-ins functions for some tasks, that otherwise probably would have been written by looping over the array values. As an example here we are going to use Math.max/Math.min, to find out the maximum/minimum value in an array.

// min/max number in an array

var numbers = [5, 6, 2, 3, 7];

// using Math.min/Math.max apply

var max = Math.max.apply(null, numbers);

// This about equal to Math.max(numbers[0], ...)

// or Math.max(5, 6, ...)

var min = Math.min.apply(null, numbers);

// vs. simple loop based algorithm

max = -Infinity, min = +Infinity;

for (var i = 0; i < numbers.length; i++) {

if (numbers[i] > max) {

max = numbers[i];

}

if (numbers[i] < min) {

min = numbers[i];

}

}

But beware: in using apply this way, you run the risk of exceeding the JavaScript engine's argument length limit. The consequences of applying a function with too many arguments (think more than tens of thousands of arguments) vary across engines (JavaScriptCore has hard-coded [argument limit of 65536](https://bugs.webkit.org/show_bug.cgi?id=80797)), because the limit (indeed even the nature of any excessively-large-stack behavior) is unspecified. Some engines will throw an exception. More perniciously, others will arbitrarily limit the number of arguments actually passed to the applied function. To illustrate this latter case: if such an engine had a limit of four arguments (actual limits are of course significantly higher), it would be as if the arguments 5, 6, 2, 3 had been passed to apply in the examples above, rather than the full array.

If your value array might grow into the tens of thousands, use a hybrid strategy: apply your function to chunks of the array at a time:

function minOfArray(arr) {

var min = Infinity;

var QUANTUM = 32768;

for (var i = 0, len = arr.length; i < len; i += QUANTUM) {

var submin = Math.min.apply(null,

arr.slice(i, Math.min(i+QUANTUM, len)));

min = Math.min(submin, min);

}

return min;

}

var min = minOfArray([5, 6, 2, 3, 7]);