

Function.prototype.apply()

Jump to: [Syntax](#) [Description](#) [Examples](#) [Specifications](#) [Browser compatibility](#) [See also](#)

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

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**.

JavaScript Demo: Function.apply()

```
1 var numbers = [5, 6, 2, 3, 7];
2
3 var max = Math.max.apply(null, numbers);
4
5 console.log(max);
6 // expected output: 7
7
8 var min = Math.min.apply(null, numbers);
9
10 console.log(min);
11 // expected output: 2
12
```


Syntax

```
function.apply(thisArg, [argsArray])
```

Parameters

thisArg

Optional. 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

Optional. An array-like object, specifying the arguments with which *func* should be called, or `null` or `undefined` if no arguments should be provided to the function. Starting with ECMAScript 5 these arguments can be a generic array-like object instead of an array. See below for browser compatibility information.

Return value

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

Description

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.

Since ECMAScript 5th Edition you can also use any kind of object which is array-like, so in practice this means it's going to have a property `length` and integer properties in the range `(0..length-1)`. As an example you can now use a `NodeList` or a custom object like `{ 'length': 2, '0': 'eat', '1': 'bananas' }`.

❏ Most browsers, including Chrome 14 and Internet Explorer 9, still do not accept array-like objects and will throw an exception.

Examples

Using `apply` to append an array to another

We can use `push` to append an element to an array. And, because `push` accepts a variable number of arguments, we can also push multiple elements at once. But, if we pass an array to `push`, it will actually add that array as a single element, instead of adding the elements individually, so we end up with an array inside an array. What if that is not what we want? `concat` does have the behaviour we want in this case, but it does not actually append to the existing array but creates and returns a new array. But we wanted to append to our existing array... So what now? Write a loop? Surely not?

`apply` to the rescue!

```
1 | var array = ['a', 'b'];
2 | var elements = [0, 1, 2];
3 | array.push.apply(array, elements);
4 | console.info(array); // ["a", "b", 0, 1, 2]
```

Using `apply` and built-in functions

Clever usage of `apply` allows you to use built-in 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.

```
1 | // min/max number in an array
2 | var numbers = [5, 6, 2, 3, 7];
3 |
4 | // using Math.min/Math.max apply
5 | var max = Math.max.apply(null, numbers);
6 | // This about equal to Math.max(numbers[0], ...)
7 | // or Math.max(5, 6, ...)
8 |
9 | var min = Math.min.apply(null, numbers);
10 |
11 | // vs. simple loop based algorithm
12 | max = -Infinity, min = +Infinity;
13 |
```

```

14 for (var i = 0; i < numbers.length; i++) {
15     if (numbers[i] > max) {
16         max = numbers[i];
17     }
18     if (numbers[i] < min) {
19         min = numbers[i];
20     }
21 }

```

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](#)), 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:

```

1 function minOfArray(arr) {
2     var min = Infinity;
3     var QUANTUM = 32768;
4
5     for (var i = 0, len = arr.length; i < len; i += QUANTUM) {
6         var submin = Math.min.apply(null,
7                                   arr.slice(i, Math.min(i+QUANTUM, len)));
8         min = Math.min(submin, min);
9     }
10
11     return min;
12 }
13
14 var min = minOfArray([5, 6, 2, 3, 7]);

```

Using `apply` to chain constructors

You can use `apply` to chain constructors for an object, similar to Java. In the following example we will create a global Function method called `construct`, which will enable you to use an array-like object with a constructor instead of an arguments list.

```

1 Function.prototype.construct = function(aArgs) {
2     var oNew = Object.create(this.prototype);
3     this.apply(oNew, aArgs);
4     return oNew;
5 };

```

Note: The `Object.create()` method used above is relatively new. For alternative methods, please consider one of the following approaches:

Using `Object.__proto__`:

```

1 Function.prototype.construct = function (aArgs) {
2     var oNew = {};
3     oNew.__proto__ = this.prototype;
4     this.apply(oNew, aArgs);
5     return oNew;

```

Example usage:

```


1 function MyConstructor() {
2     for (var nProp = 0; nProp < arguments.length; nProp++) {
3         this['property' + nProp] = arguments[nProp];
4     }
5 }
6
7 var myArray = [4, 'Hello world!', false];

```









```

8 | var myInstance = MyConstructor.construct(myArray);
9 |
10 | console.log(myInstance.property1);           // logs 'Hello world!'
11 | console.log(myInstance instanceof MyConstructor); // logs 'true'
12 | console.log(myInstance.constructor);         // logs 'MyConstructor'

```

 **Note:** This non-native `Function.construct` method will not work with some native constructors; like `Date`, for example. In these cases you have to use the `Function.prototype.bind` method. For example, imagine having an array like the following, to be used with `Date` constructor: `[2012, 11, 4]`; in this case you have to write something like: `new (Function.prototype.bind.apply(Date, [null].concat([2012, 11, 4])))()`. This is not the best way to do things, and probably not to be used in any production environment.

Specifications

Specification	Status	Comment
 ECMAScript 3rd Edition (ECMA-262)	 ST Standard	Initial definition. Implemented in JavaScript 1.3.
 ECMAScript 5.1 (ECMA-262) The definition of 'Function.prototype.apply' in that specification.	 ST Standard	
 ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Function.prototype.apply' in that specification.	 ST Standard	
 ECMAScript Latest Draft (ECMA-262) The definition of 'Function.prototype.apply' in that specification.	 D Draft	

Browser compatibility

New compatibility tables are in beta

<div></div>						<div></div>						<div></div>		
<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>
Basic support														
Yes	Yes	1	Yes	Yes	Yes	Yes	Yes	Yes	4	Yes	Yes	Yes	Yes	Yes
ES 5.1: generic array-like object as arguments														
Yes	?	4	?	?	?	?	Yes	?	4	?	?	Yes	?	?

..

Full support

..

Compatibility unknown

See also

- arguments object
- `Function.prototype.bind()`
- `Function.prototype.call()`
- Functions and function scope
- `Reflect.apply()`