Function.prototype.bind()

The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

Try it

JavaScript Demo: Function.bind()

```
const module = {
  x: 42,
  getX: function() {
    return this.x;
  }
};

const unboundGetX = module.getX;
console.log(unboundGetX()); // The function gets invoked at the global scope
// Expected output: undefined

const boundGetX = unboundGetX.bind(module);
console.log(boundGetX());
// Expected output: 42

```

```
const module = {
  x: 42,
```

Run › Reset

Syntax

```
bind(thisArg)
bind(thisArg, arg1)
bind(thisArg, arg1, arg2)
bind(thisArg, arg1, arg2, /* …, */ argN)
```

Parameters

`thisArg`

The value to be passed as the `this` parameter to the target function `func` when the bound function is called. If the function is not in strict mode, `null` and `undefined` will be replaced with the global object, and primitive values will be converted to objects. The value is ignored if the bound function is constructed using the `new` operator.

`arg1, …, argN` Optional

Arguments to prepend to arguments provided to the bound function when invoking `func`.

Return value

A copy of the given function with the specified `this` value, and initial arguments (if provided).

Description

The `bind()` function creates a new *bound function*. Calling the bound function generally results in the execution of the function it wraps, which is also called the *target function*. The bound function will store the parameters passed — which include the value of `this` and the first few arguments — as its internal state. These values are stored in advance, instead of being passed at call time. You can generally see `const boundFn = fn.bind(thisArg, arg1, arg2)` as being equivalent to `const boundFn = (...restArgs) => fn.call(thisArg, arg1, arg2, ...restArgs)` for the effect when it's called (but not when `boundFn` is constructed).

A bound function can be further bound by calling `boundFn.bind(thisArg, /* more args */)`, which creates another bound function `boundFn2`. The newly bound `thisArg` value is ignored, because the target function of `boundFn2`, which is `boundFn`, already has a bound `this`. When `boundFn2` is called, it would call `boundFn`, which in turn calls `fn`. The arguments that `fn` ultimately receives are, in order: the arguments bound by `boundFn`, arguments bound by `boundFn2`, and the arguments received by `boundFn2`.

```
"use strict"; // prevent `this` from being boxed into the wrapper object

function log(...args) {
  console.log(this, ...args);
}
const boundLog = log.bind("this value", 1, 2);
const boundLog2 = boundLog.bind("new this value", 3, 4);
boundLog2(5, 6); // "this value", 1, 2, 3, 4, 5, 6
```

A bound function may also be constructed using the `new` operator if its target function is constructable. Doing so acts as though the target function had instead been constructed. The prepended arguments are provided to the target function as usual, while the provided `this` value is ignored (because construction prepares its own `this`, as seen by the parameters of `Reflect.construct`). If the bound function is directly constructed, `new.target` will be the target function instead. (That is, the bound function is transparent to `new.target`.)

```
class Base {
  constructor(...args) {
    console.log(new.target === Base);
    console.log(args);
  }
}

const BoundBase = Base.bind(null, 1, 2);

new BoundBase(3, 4); // true, [1, 2, 3, 4]
```

However, because a bound function does not have the `prototype` property, it cannot be used as a base class for `extends`.

```
class Derived extends class {}.bind(null) {}
// TypeError: Class extends value does not have valid prototype property undefined
```

When using a bound function as the right-hand side of `instanceof`, `instanceof` would reach for the target function (which is stored internally in the bound function) and read its `prototype` instead.

```
class Base {}
const BoundBase = Base.bind(null, 1, 2);
console.log(new Base() instanceof BoundBase); // true
```

The bound function has the following properties:

`length`

The `length` of the target function minus the number of arguments being bound (not counting the `thisArg` parameter), with 0 being the minimum value.

`name`

The `name` of the target function plus a `"bound "` prefix.

The bound function also inherits the prototype chain of the target function. However, it doesn't have other own properties of the target function (such as static properties if the target function is a class).

Examples

Creating a bound function

The simplest use of `bind()` is to make a function that, no matter how it is called, is called with a particular `this` value.

A common mistake for new JavaScript programmers is to extract a method from an object, then to later call that function and expect it to use the original object as its `this` (e.g., by using the method in callback-based code).

Without special care, however, the original object is usually lost. Creating a bound function from the function, using the original object, neatly solves this problem:

```js
this.x = 9; // 'this' refers to the global object (e.g. 'window') in non-strict mode
const module = {
  x: 81,
  getX() {
    return this.x;
  },
};

console.log(module.getX()); // 81

const retrieveX = module.getX;
console.log(retrieveX()); // 9; the function gets invoked at the global scope

// Create a new function with 'this' bound to module
// New programmers might confuse the
// global variable 'x' with module's property 'x'
const boundGetX = retrieveX.bind(module);
console.log(boundGetX()); // 81
```

Note: If you run this example in strict mode (e.g. in ECMAScript modules, or through the `"use strict"` directive), the global `this` value will be undefined, causing the `retrieveX` call to fail.

If you run this in a Node CommonJS module, the top-scope `this` will be pointing to `module.exports` instead of `globalThis`, regardless of being in strict mode or not. However, in functions, the reference of unbound `this` still follows the rule of "`globalThis` in non-strict, `undefined` in strict". Therefore, in non-strict mode (default), `retrieveX` will return `undefined` because `this.x = 9` is writing to a different object (`module.exports`) from what `getX` is reading from (`globalThis`).

In fact, some built-in "methods" are also getters that return bound functions — one notable example being `Intl.NumberFormat.prototype.format()`, which, when accessed, returns a bound function that you can directly pass as a callback.

Partially applied functions

The next simplest use of `bind()` is to make a function with pre-specified initial arguments.

These arguments (if any) follow the provided `this` value and are then inserted at the start of the arguments passed to the target function, followed by whatever arguments are passed to the bound function at the time it is called.

```
function list(...args) {
  return args;
}

function addArguments(arg1, arg2) {
  return arg1 + arg2;
}

console.log(list(1, 2, 3)); // [1, 2, 3]

console.log(addArguments(1, 2)); // 3

// Create a function with a preset leading argument
const leadingThirtySevenList = list.bind(null, 37);

// Create a function with a preset first argument.
const addThirtySeven = addArguments.bind(null, 37);

console.log(leadingThirtySevenList()); // [37]
console.log(leadingThirtySevenList(1, 2, 3)); // [37, 1, 2, 3]
console.log(addThirtySeven(5)); // 42
console.log(addThirtySeven(5, 10)); // 42
// (the last argument 10 is ignored)
```

With setTimeout()

By default, within `setTimeout()`, the `this` keyword will be set to `globalThis`, which is `window` in browsers. When working with class methods that require `this` to refer to class instances, you may explicitly bind `this` to the callback function, in order to maintain the instance.

```
class LateBloomer {
  constructor() {
    this.petalCount = Math.floor(Math.random() * 12) + 1;
  }
  bloom() {
    // Declare bloom after a delay of 1 second
    setTimeout(this.declare.bind(this), 1000);
  }
  declare() {
    console.log(`I am a beautiful flower with ${this.petalCount} petals!`);
  }
}

const flower = new LateBloomer();
flower.bloom();
// After 1 second, calls 'flower.declare()'
```

You can also use arrow functions for this purpose.

```
class LateBloomer {
  bloom() {
    // Declare bloom after a delay of 1 second
    setTimeout(() => this.declare(), 1000);
  }
}
```

Bound functions used as constructors

Bound functions are automatically suitable for use with the `new` operator to construct new instances created by the target function. When a bound function is used to construct a value, the provided `this` is ignored. However, provided arguments are still prepended to the constructor call.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return `${this.x},${this.y}`;
};

const p = new Point(1, 2);
p.toString();
// '1,2'

// The thisArg's value doesn't matter because it's ignored
const YAxisPoint = Point.bind(null, 0 /*x*/);

const axisPoint = new YAxisPoint(5);
axisPoint.toString(); // '0,5'

axisPoint instanceof Point; // true
axisPoint instanceof YAxisPoint; // true
new YAxisPoint(17, 42) instanceof Point; // true
```

Note that you need not do anything special to create a bound function for use with `new`. `new.target`, `instanceof`, `this` etc. all work as expected, as if the constructor was never bound. The only difference is that it can no longer be used for `extends`.

The corollary is that you need not do anything special to create a bound function to be called plainly, even if you would rather require the bound function to only be called using `new`. If you call it without `new`, the bound `this` is suddenly not ignored.

```
const emptyObj = {};
const YAxisPoint = Point.bind(emptyObj, 0 /*x*/);

// Can still be called as a normal function
// (although usually this is undesirable)
YAxisPoint(13);

// The modifications to `this` is now observable from the outside
console.log(emptyObj); // { x: 0, y: 13 }
```

If you wish to restrict a bound function to only be callable with `new`, or only be callable without `new`, the target function must enforce that restriction, such as by checking `new.target !== undefined` or using a class instead.

Binding classes

Using `bind()` on classes preserves most of the class's semantics, except that all static own properties of the current class are lost. However, because the prototype chain is preserved, you can still access static properties inherited from the parent class.

```
class Base {
  static baseProp = "base";
}

class Derived extends Base {
  static derivedProp = "derived";
}

const BoundDerived = Derived.bind(null);
```

```
console.log(BoundDerived.baseProp); // "base"
console.log(BoundDerived.derivedProp); // undefined
console.log(new BoundDerived() instanceof Derived); // true
```

Transforming methods to utility functions

`bind()` is also helpful in cases where you want to transform a method which requires a specific `this` value to a plain utility function that accepts the previous `this` parameter as a normal parameter. This is similar to how general-purpose utility functions work: instead of calling `array.map(callback)`, you use `map(array, callback)`, which avoids mutating `Array.prototype`, and allows you to use `map` with array-like objects that are not arrays (for example, `arguments`).

Take `Array.prototype.slice()`, for example, which you want to use for converting an array-like object to a real array. You could create a shortcut like this:

```
const slice = Array.prototype.slice;

// ...

slice.call(arguments);
```

Note that you can't save `slice.call` and call it as a plain function, because the `call()` method also reads its `this` value, which is the function it should call. In this case, you can use `bind()` to bind the value of `this` for `call()`. In the following piece of code, `slice()` is a bound version of `Function.prototype.call()`, with the `this` value bound to `Array.prototype.slice()`. This means that additional `call()` calls can be eliminated:

```
// Same as "slice" in the previous example
const unboundSlice = Array.prototype.slice;
const slice = Function.prototype.call.bind(unboundSlice);

// ...

slice(arguments);
```

Specifications

Specification
ECMAScript Language Specification # sec-function.prototype.bind

Browser compatibility

Report problems with this compatibility data on GitHub ■

	■					■						■	
	Chrome ■	Edge ■	Firefox ■	Opera ■	Safari ■	Chrome Android ■	Firefox for Android ■	Opera Android ■	Safari on iOS ■	Samsung Internet ■	WebView Android ■	Deno ■	Node.js ■
`bind`	■ Chrome 7	■ Edge 12	■ Firefox 4	■ Opera 11.6	■ Safari 5.1	■ Chrome 18 Android	■ Firefox 4 for Android	■ Opera 12 Android	■ Safari 6 on iOS	■ Samsung 1.0 Internet	■ WebView 4 Android	■ Deno 1.0	■ Node.js 0.10.0

Tip: you can click/tap on a cell for more information.

■ Full support