Function Bind

We'll cover the following

Function Bind#

Instructions#

Implement Function.prototype.bind() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind).

Input: Function, [additional parameters]

Output: Function

Hints#

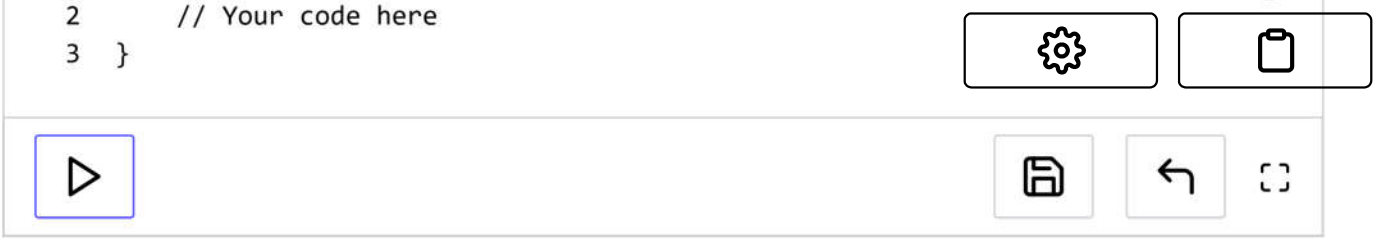
Before attempting this, you may want to read up on:

- Apply, call, and bind (https://codeplanet.io/javascript-apply-vs-call-vs-bind/)
- Scope (https://www.educative.io/collection/page/5679346740101120/5707702298738688/5757334940811264) & Closures (http://javascriptissexy.com/understand-javascript-closures-with-ease/)
- The Rules to `this` (https://www.educative.io/collection/page/5679346740101120/5707702298738688/5676830073815040)
- The Arguments Object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)

We'll call our function `Bind` instead of `bind` so we can leave the original intact.

```
1  Function.prototype.Bind = function() {
```

```
    // Your code here
}
```



We'll provide 3 similar solutions right away in order to drive home the concepts at play.

Solution 1#

ES5#

```
Function.prototype.Bind = function() {
    var fnToBind = this;
    var argsToBind = Array.prototype.slice.call(arguments);
    var thisArg = argsToBind.shift();

    return function() {
        var newArgs = Array.prototype.slice.call(arguments);
        var allArguments = [].concat(argsToBind, newArgs);
        return fnToBind.apply(thisArg, allArguments);
    }
}
```

Solution 2#

Using Rest and Spread#

```
Function.prototype.Bind = function(thisArg, ...args) {
    const self = this;
    return function(...nextArgs) {
        return self.call(thisArg, ...args, ...nextArgs);
    }
};
```

Solution 3#

Adding Arrow Functions#

```
Function.prototype.Bind = function(thisArg, ...args) {
    return (...nextArgs) => this.call(thisArg, ...args, ...nextArgs);
}
```

How it Works#

`Function.prototype.bind` is called on a function, as in `fn.bind()`. It acts on one function and returns another.

That returned function performs the actions of the original, with `this` bound. If additional arguments are provided at the bind time, they will be applied to the returned function when it is invoked. Additional arguments can be passed in as well.

We'll use Solution 3 above to explain all concepts at play. Let's add an example to aid our discussion.

Binding a Function#

```
Function.prototype.Bind = function(thisArg, ...args) {
    return (...nextArgs) => this.call(thisArg, ...args, ...nextArgs);
}

function print(val1, val2) {
    console.log(this.abc, val1, val2);
}

const printBound = print.Bind({ abc: 123 }, 456);
printBound(789); // -> 123 456 789
```


Breaking it Down#

Rest & Spread#

`Function.prototype.bind` is meant to accept an object as its first argument. This is the argument that gets bound to `this`. We call it `thisArg`.

It can also accept more arguments. These arguments will be applied to the original function when it is eventually called, so they can be thought of as being bound to the function as well. Using the rest operator (`...`), these arguments are stored in the array `args`.

Callbacks#

`Function.prototype.bind` returns a function. We must do the same, and it's exactly what we do on line 2. This function will be invoked later.

`this` - dot notation#

As indicated on line 9 above, `Function.prototype.bind` is called using dot notation (`print.Bind()`).

When a function is called using dot notation, inside the function, `this` is equal to the function itself. So in this case, inside `Function.prototype.bind`, `this` is equal to our function `print`.

`this` - arrow functions#

Arrow functions receive their `this` value lexically. This means that they have the same `this` value as their immediate surroundings. Therefore, `this` in the callback on line 2 is equal to the function `print` as well.

`Function.prototype.call`#

`Function.prototype.call` is similar to the `bind` function. It accepts an object as an argument and binds it to `this` inside the function. Rather than returning a function, it returns the result of calling the original function immediately.

We're using it to pass in `thisArg`, which is the object on line 9 above. When the user invokes the callback they received, `boundPrint()`, the callback will call the original function `print` with `thisArg` bound to its `this` value.

Rest and Spread - II#

At the end of the callback, we're spreading out the original arguments provided as well as any new arguments provided. This is how `456` and `789` are correctly provided to the function.

Conclusion#

This is quite a bit of discussion for a function that ends up having one significant line! There are several nuanced concepts packed into this little problem, which is why it's so good to ask during interviews.

JavaScript developers are expected to know how to use `apply`, `call`, and `bind` well. Being asked to write `bind` directly tests the candidate's knowledge.

Even if the candidate knows the function well, the problem is still an excellent test of arguments, callbacks, scope, closures, and the `this` keyword.

Time & Space#

Time and space complexity are not important here. While this question does indeed test problem-solving skills, the emphasis is on deep JavaScript knowledge. In addition, it would be difficult to write an inefficient version of this function. We'd actually have to try to slow it down.

← Back

Memoized Fibonacci

Next →

Versatile Add

Completed

Report an Issue