Function Properties and Methods

The fact that functions are first-class objects means they can have properties and methods themselves. For example, all functions have a `length` property that returns the number of parameters the function has.

Let’s use the `square()` function that we wrote in chapter 4 as an example:

Copy

```
function square(x) {     return x*x;}
```

If we query the `length` property, we can see that it accepts one parameter:

Copy

```
square.length<< 1
```

If a function doesn’t refer to an object as `this` in its body, it can still be called using the `call()` method, but you need provide `null` as its first argument. For example, we could call the `square()` function using the `call()` method, like so:

Copy

```
square.call(null, 4)<< 16
```

The `apply()` method works in the same way, except the arguments of the function are provided as an array, even if there is only one argument:

Copy

```
square.apply(null, [4])<< 16
```

This can be useful if the data you’re using as an argument is already in the form of an array, although it's not really needed in ES6, as the spread operator can be used to split an array of values into separate parameters.

These are two powerful methods, as they allow generalized functions to be written that are not tied to specific objects by being methods of that object. This gives flexibility over how the functions can be used.

Custom Properties

There is nothing to stop you adding your own properties to functions in the same way that you can add properties to any object in JavaScript. For example, you could add a `description` property to a function that describes what it does:

Copy

```
square.description = 'Squares a number that is provided as an
argument'<< 'Squares a number that is provided as an argument'
```

Memoization

A useful feature of this is that it provides result caching, or memoization.

If a function takes some time to compute a return value, we can save the result in a `cache` property. Then if the same argument is used again later, we can return the value from the cache, rather than having to compute the result again. For example, say squaring a number was an expensive computational operation that took a long time. We could rewrite the `square()` function so it saved each result in a `cache` object that is a property of the function:

Copy

```
function square(x){    square.cache = square.cache || {};    if
(!square.cache[x]) {        square.cache[x] = x*x;    }    return
square.cache[x]}
```