The Difference Between Call and Apply in Javascript

One very common thing that trips me up when writing Javascript is knowing when to use `call` and when to use `apply`. If you're wondering what these methods are, or don't know how scope works in JavaScript, then it might make sense to read the Javascript Guide first.

Let's look at some ways we might want to use them:

```
var person1 = {name: 'Marvin', age: 42, size: '2xM'};
var person2 = {name: 'Zaphod', age: 42000000000, size: '1xS'};

var sayHello = function(){
    alert('Hello, ' + this.name);
};

var sayGoodbye = function(){
    alert('Goodbye, ' + this.name);
};
```

Now if you've read the guide, this example will look really familiar. You'd already know that writing the following code:

```
sayHello();
sayGoodbye();
```

will give errors (if you're lucky), or just unexpected results (if you aren't). This is because both functions rely on their scope for the `this.name` data, and calling them without explicit scope will just run them in the scope of the current window.

So how do we scope them? Try this:

```
sayHello.call(person1);
sayGoodbye.call(person2);

sayHello.apply(person1);
sayGoodbye.apply(person2);
```

All four of these lines do exactly the same thing. The run `sayHello` or `sayGoodbye` in the scope of either `person1` or `person2`.

Both `call` and `apply` perform very similar functions: they execute a function in the context, or scope, of the first argument that you pass to them. Also, they're both functions that can only be called on other functions. You're not going to able to run `person1.call()`, nor does it make any sense to do so.

The difference is when you want to *seed* this call with a set of arguments. Say you want to make a `say()` method that's a little more dynamic:

```
var say = function(greeting){
    alert(greeting + ', ' + this.name);
};
```

```
say.call(person1, 'Hello');
say.call(person2, 'Goodbye');
```

So that's `call` for you. It runs the function in the context of the first argument, and subsequent arguments are passed in to the function to work with. So how does it work with more than one argument?

```
var update = function(name, age, size){
    this.name = name;
    this.age = age;
    this.size = size;
};

update.call(person1, 'Slarty', 200, '1xM');
```

No big deal. They're simply passed to the function if it takes more than one parameter.

The limitations of `call` quickly become apparent when you want to write code that doesn't (or shouldn't) know the number of arguments that the functions need… like a dispatcher.

```
var dispatch = function(person, method, args){
    method.apply(person, args);
};

dispatch(person1, say, ['Hello']);
dispatch(person2, update, ['Slarty', 200, '1xM']);
```

So that's where `apply` comes in - the second argument needs to be an array, which is unpacked into arguments that are passed to the called function.

So that's the difference between `call` and `apply`. Both can be called on functions, which they run in the context of the first argument. In `call` the subsequent arguments are passed in to the function as they are, while `apply` expects the second argument to be an array that it unpacks as arguments for the called function.

If you enjoyed this article, do tweet it or share the link with others.

You could also look at other articles with the same tag: javascript

Brought to you by the folks behind Runway7, a collection of useful APIs that make web development faster and more convenient.