**Ways to Call a Function**

Let's review the various ways to call a function that we have encountered:

* Function-style when unbound (fun(arg1, arg2))
  + this is set to the global context (global or window)
* Method-style (obj.method(arg1, arg2))
  + this is set to obj
* Constructor-style (new ClassName(arg1, arg2)).
  + Creates a new, blank object.
  + Sets its \_\_proto\_\_ property to ClassName.prototype.
  + The ClassName function is called with this set to the blank object.
    - Your constructor function sets up the object's key-value pairs (instance variables).
    - Implemented like the initialize method from Ruby
    - The return value is ignored.

Note that callbacks (that is, functions you pass to other functions) are almost always eventually called function style, which, for a function not an arrow function, makes the function's context the same as the context that it is invoked in. Remember, you can use an anonymous function with the const that = this trick or bind to ensure that this is set properly in your callback, but using arrow functions so that this is set to the surrounding scope is preferred in ES6.

**Two last ways to call functions**

Let me regale you with two more ways to call a function: apply and call. You will not use these very commonly, but you will see them in the code of various libraries. They are quite simple, but they take practice.

**Apply**

Function.prototype.apply takes two arguments: an object to bind this to, and an array of arguments to be passed to the method apply is being called on. This is what it looks like:

const obj = {

name: "Earl Watts"

};

// weird function; how is `this` supposed to be set if we don't call

// `greet` method style?

function greet(msg) {

console.log(`${msg}: ${this.name}`);

}

greet.apply(obj, ["Hello"]);

Okay, so what's going on here? Let's start with the first argument that got passed, obj. apply wants to know what object to bind this to. apply simulates calling greet as a method on obj. This is sort of like saying obj.greet("Hello"), except greet isn't an attribute of obj, so we couldn't call it that way exactly.

Note that the second argument to apply is an array of arguments to be passed to the function, greet.

**Call**

Function.prototype.call is really similar to apply, but instead of taking in an array of parameters, it takes them individually. For example:

const obj = {

name: "Earl Watts"

};

function greet(msg1, msg2) {

console.log(`${msg1}: ${this.name}`);

console.log(`${msg2}: ${this.name}`);

}

greet.call(obj, "Hello", "Goodbye");

Why prefer call over apply, and vice versa? In general, call is more convenient when you know ahead of time what arguments you want to pass. apply is more useful when someone is going to give you an array of arguments to use. There is a slight performance cost to using apply because the arguments need to be unpacked. Don't worry much about it, though.