JUL 22 2009

4 Ways Functions Mess With this

In JavaScript, the `this` keyword can be a tricky thing. It's trickiness comes from how functions behave differently depending on how you call them. What? You can call them differently? Yep! There's 4 major invocation patterns for a function, so let's see how each works, and how they handle `this`.

1. Method Invocation
2. Function Invocation
3. Constructor Invocation
4. Apply Invocation

METHOD INVOCATION

The first patten of using functions will look the most familiar. When you come from classical programming languages (C#, Java), you're likely used to classes being the only building blocks of a program. Nothing can exist without a class. You only have a function that is a method of a class. And it's super simple what `this` relates to.

Well, that is the same way method invocation works in Javascript. If you define an Object, and make one of its properties a function, then

when you execute that method, `this` will be the Object that owns it.

```
var Obj = {
    something: 'a string',
    changeSomething: function() {
        //this == Obj
        return this.something.toUpperCase();
    }
};
Obj.changeSomething(); //returns 'A STRING'
```

Anytime we call `changeSomething` of `Obj`, like above, we know `this` will be `Obj`, so we don't have to repeat the name of the object constantly. This is also the same if we were to clone the object. `this` would reference the object that owns the method.

FUNCTION INVOCATION

The second pattern, might look a little stranger, and seem to behave strange also, but once you realize a few things about JavaScript, you'll see there is logic behind it (*yes, really!*). The next pattern is commonly called the Function invocation, and this is basically where we define a function all by its lonesome self. This is possible because in JavaScript, functions are first-class objects. So they can be passed around, stored, all the good stuff. But let's assume we know all this.

When we declare a function not associated with an object, then invoking that function will have `this` bound to the global object (in most cases, `window`). Basically because, every function must be bound to an object. If it's bound to an object by these other patterns,

then if defaults to the top level object. This seems understandable enough when declaring global functions.

```
var logWindow = function() {
    //this == window
    console.log(this);
};
//both result in the same
function logWindow() {
    //this == window
    console.log(this);
}
logWindow() // logs the window object in Firebug
```

However, this pattern applies to anywhere you define such a function, not just at the global level. This is what causes people to have big problems. Say we have a function of an object, and invoke the method. We know this will be bound to the owning object. But JavaScript allows us to declare functions inside of functions. We might do this to create event handlers, callbacks, or simple utility functions we use multiple times in the method. Well, inside **those** functions, `this` is no longer bound to the owning object. Indeed, inside those scopes, `this` is actually bound to the global object.

```
//oops
var Thing = {
    name: 'I was owned by Thing',
    woops: function() {
        //this == Thing
        var concat = function(first,second) {
            //this == window
            this.name = first %2B second;
```

```
        };
        concat(this.name, ', but I just changed window!');
    }
};
Thing.woops(); //sets window.name = 'I was owned by Thing,
```

Here we can see the method woops defines an inner function (that is overly simple, yes) to help accomplish its goal. It passes in 2 strings to the function, and access `this`, which is what we hoped, `Thing`. But then the inner function, `concat`, is told to re-assign a property of `this`. Well, that function has `this` bound to `window`, so we just started blowing away properties on `window`. Just lovely.

The work-arounds for this problem aren't all the difficult. Most involve simply storing `this` in a different variable before you get to the scope of the inner function, like `self`, and then inside, access `self` instead of `this`. Here's that same example, using this fix.

```
//self
var Thing = {
    name: 'I was owned by Thing',
    woops: function() {
        //this == Thing
        var self = this;
        var concat = function(first,second) {
            //this == window
            //self == Thing
            self.name = first %2B second;
        };
        concat(this.name, ', and still am. Woot!');
    }
```

CONSTRUCTOR INVOCATION

The third way, using a function as a constructor, breaks all the rules of the past 2 patterns. When you use the `new` operator with a function, while that function is executing, `this` is bound to a brand new object. That object gets assigned the prototype of the function as the prototype of the object. And any uses of `this` inside that constructor function will modify your new object during creation.

This pattern is fairly intuitive if you're used to classical languages. We can create a function that is expected to be used to create new objects. This function will run commands on the new object that is created, and then return the new object.

```
var Dog = function(name) {
    //this == brand new object ({});
    this.name = name;
    this.age = (Math.random() * 5) + 1;
};
var myDog = new Dog('Spike');
//myDog.name == 'Spike'
//myDog.age == 2
var yourDog = new Dog('Spot');
//yourDog.name == 'Spot'
//yourDog.age == 4
```

When `myDog` called a `new Dog`, `this` is applied to the object being assigned to `myDog`. Later, when we assigned `yourDog` to a new Dog, `this` then is applied to the object being assigned to `yourDog`. Capiche?

One nicety that I've used, is that this function can be anywhere. It can be an inner function, it can be a global function, or it can be a method of an object. When you employ the constructor pattern, it changes the way that function works.

A word of caution though, if you create a function to be a constructor, use it, but accidentally forget the `new` keyword, you have completely screwed the way the function will be bound. In other classical languages, you would have recieved a compiler error or something equivalent. In JavaScript, you'll just have *clobbered the object that owned the function*, be it a specified object, or the window.

```
var Dog = function(name) {
    this.name = name;
    this.age = (Math.random() * 5) %2B 1;
};
var notADog = Dog('Lassie');//notADog is undefined
//window.name == 'Lassie';
//window.age == 1;
```

In this instance, we clobbered `window`, and we didn't even get `window` returned to `notADog`, because our `Dog` function doesn't normally return a value. It only returns when used in the Constructor pattern.

I personally don't have any problem with this, since any time I've ever wanted a new object from a function, I automatically write `new`. **It's ingrained**. However, to help curb the problem, its good practice to name constructor functions with a capital letter, and all other functions with a lower case letter. That might help you spot when a `new` is missing.

If you habitually do this and clobber objects you're not supposed to, you can add a little part to each of your constructor methods to insure the function has been invoked with `new`.

```
//Constructor checker
var Dog = function(name) {
    if(!(this instanceof Dog)) {
        throw new Error('You forgot "new", dummy!');
    }
    this.name = name;
};
var notADog = Dog('Bart');// throws an Error at you
```

APPLY INVOCATION

Lastly, there is a pattern to specifically control what `this` should be in a function, no matter what it would be by default. Like I said earlier, functions are objects, which means they can have methods. All functions have a method called `apply`, which when used will invoke the owning object (the function) with a specific object bound as `this`.

```
var addMessages = function(mess1, mess2, mess3) {
    //normally, this == window
    this.message = mess1 %2B mess2 %2B mess3;
};
var test = {};
var args = ['test ', 'will be', ' this'];
addMessage.apply(test,args);
```

While `addMessage`, being a global function, would normally be bound to `window`, if we use its method `apply`, we can pass an object to be bound to. The other argument for `apply` is an array of arguments for the function.

There's another way to do almost the same thing, using the `call` method. The only difference I can tell, is that instead of taking an array of arguments, you feed them in individually.

```
addMessage.call(test, 'test ', 'will be', ' this');
```

You can use these methods on any function. **Any**. For instance, on methods or prototypes. You might find someone use `apply` of an `Array` method on an `arguments` array, or element collection, since they aren't actual arrays, they don't have the methods on their `prototype`. But by using `apply`, you can use those methods on them anyways.

THAT'S A WRAP

Functions in JavaScript are a lot of fun. But before you've used JavaScript too much, these things can really get you. Or even if you have been using JavaScript before, you might not have really known why some things happen the way they do. Knowing these has certainly helped me to design better JavaScript.