

## Intro to Prototype/JavaScript OO/Ajax

---

### Java-like OO

JavaScript is an OO language, but the OO implementation is different than anything you've likely seen before. However, it is incredibly extensible and the Prototype library has provided an implementation of more conventional OO as you would see in C++ or Java. See this simple example:

```
var WordInfo = Class.create({
  initialize: function(correct, corrections) {
    this.correct = correct;
    this.corrections = corrections;
  }
});
```

This is more of a struct in that it just holds onto data. Things to note:

- `Class.create` is a function call that takes a hash containing instance methods as an argument
- `initialize` is the name of the constructor
- Member variables are created by assigning `this.var` in the constructor

To create and use an instance of the `WordInfo` struct:

```
>>> var word = new WordInfo(false, ['luck', 'lucks', 'lucky']);
>>> word.correct
false
```

The call to `new` tells the interpreter to acquire a new object and bind the value of the `this` variable in the function to that new object.

The `this` variable in JavaScript deserves some discussion.

```
>>> (function() { return this; })();
```

This is a "free function," right? In C++/Java it does not make sense to reference `this` outside of an instance method. Well, in JavaScript it doesn't make much sense either, but it is valid. It actually returns:

```
>>> (function() { return this; })();
Window about:blank
```

You can argue about how much sense this makes, but that's JS for you. However, inside of classes, `this` does pretty much what you would expect.

```

var SimpleClass = Class.create( {
    initialize: function() { },
    f: function() { return this }
});
>>> s = new SimpleClass();
>>> s.f() == s
true

```

But with higher-order functions things become a little strange. Prototype's class creation mechanism takes care of wiring `this` to the correct object for instance methods, but you're on your own for any local functions declared inside of a method.

```

var BiggerClass = Class.create({
    initialize: function() { },
    strange: function() {
        var self = this;
        function f() {
            return self == this;
        }
        return f();
    }
});
>>> s = new BiggerClass();
>>> s.f()
false

```

Inside of the local function `f`, `this` refers to `window`. How do we remedy this? The first option is to forget about the `this` keyword and use it sparingly. After we bind `this` to `self`, thanks to static scoping we can use `self` inside of `f` and it will be the object we expect. This feels a bit hacky, though, right?

So the solution is to use the `bind` function. The `bind` function is added to all functions with Prototype. It allows the programmer to explicitly define what should be considered `this` in the invocation of the function. Think of it as currying away the argument named `this`.

```

var BetterClass = Class.create( {
    initialize: function() { },
    strange: function() {
        var self = this;
        var f = (function() {
            return self == this;
        }).bind(this);
        return f();
    }
});

```

```
>>> s = new BetterClass();
>>> s.f()
true
```

### A Simple Ajax Request

This business about higher-order functions and binding is not purely academic in a dynamic language like JS. In fact, these language features are essential to using Ajax or other event-driven features well.

First let's imagine how you would like an Ajax request to look. Let's imagine we are writing a spellchecker.

```
var SpellChecker = Class.create({
  initialize: function() {
  },

  checkWord: function(word) {
    return this._lookupWord(word);
  }

  // this is pseudo-code and NOT the way to do things!
  _lookupWord: function(word) {
    req = new Ajax.Request('/spellcheck/' + word,
      {
        method: 'get'
      });
    return req.responseText == "correct";
  }
}
```

You call `thisSpellChecker().checkWord(word)` and get a true/false answer back. However, this is NOT the way things work. Why? Let's break down that buzzword: **Asynchronous JavaScript And XML**. Ignore that XML business. What does asynchronous mean here?

Basically it means that every request to the server is handled asynchronously, as opposed to synchronously in this example. In this example, the call to the request just sits there until the server answers and then the code continues. But the server is free to take minutes to provide an answer, and all the time your JavaScript has been frozen. This wouldn't be so bad if JavaScript weren't essentially single-threaded. But it is, and your app would be frozen while waiting for the server.

The solution is to introduce callbacks to process requests asynchronously. In this model after you define a `Request` with a callback function, your code keeps on chugging. After the server has answered, the callback function is called with the result of the request.

So we end up with something more like:

```

var SpellChecker = Class.create({
  initialize: function() {
    this._cache = {};
  },

  checkWord: function(word) {
    this._lookupWord(word);
  }

  // this is pseudo-code and NOT the way to do things!
  _lookupWord: function(word) {
    cb = (function(transport) {
      this._cache[word] = transport.responseText;
      $('resultDiv').innerHTML = transport.responseText;
    }).bind(this) //wtf does this do!
    req = new Ajax.Request('/spellcheck/' + word,
      {
        method: 'get',
        onSuccess: cb
      });
  }
}

```

If this isn't intuitively clear then you should code up an example of an asynchronous request. Notice that binding is actually important here - we couldn't maintain a cache inside of the object without binding the callback to the object.