Basic JavaScript Part 2 : Objects

November 12th, 2010

In a previous blog post, I showed some of the rich capabilities of functions in JavaScript. For this post I want to have a brief look at objects in JavaScript. Let's start with a basic example:

```
var podcast = {
    title: 'Astronomy Cast',
    description: 'A fact-based journey through the galaxy.',
    link: 'http://www.astronomycast.com'
};
```

When you've already seen some kind of data in JSON format, this might look familiar. In JavaScript, this is called an *object literal*. The example above shows a variable named *podcast*that contains an object with three properties, named *title*, *description* and *link*. As you might expect, it's now possible to access the data of these properties using the regular dot notation.

```
console.log(podcast.title);
console.log(podcast.description);
console.log(podcast.link);
```

Hopefully there are no big surprises here. But what I find to be more interesting about objects in JavaScript are their similarities with arrays. Take a look at the following array definition:

```
var astronomyPodcast = ['Astronomy Cast', 'A fact based ...', 'http://www.astro ...'];
```

When I want to access the description in the array, I simply go by using the correct index:

```
console.log(astronomy[1]);    // outputs the description
```

No big deal so far. The nice thing about objects in JavaScript is that we can use the same notation but only with a small difference:

```
console.log(podcast['description']);
```

The sole difference here is that we are using the name of the property instead of a numeric index. Objects actually behave the same as arrays in JavaScript except that for objects you get to choose a nice and friendly name for its properties. Lets look at another example that loops over every item in the array:

```
for(var i in astronomy) {
    console.log(astronomy[i]);
}
```

This outputs every value contained by the array. The variable i contains the current index in the array. We can do the same for objects:

```
for(var i in podcast) {
    console.log(podcast[i]);
}
```

The variable *i* doesn't contain a numeric value in this case, but the name of the current property. So in short, a JavaScript object is very similar to an array with the sole difference that you have to define the keys yourself. Just a quick side note here: try to avoid for-in loops with arrays and use a regular for loop instead.

Methods

A method on an object is simply a property that contains a function.

```
var podcast = {
    title: 'Astronomy Cast',
    description: 'A fact-based journey through the galaxy.',
    link: 'http://www.astronomycast.com',

    toString: function() {
       return 'Title: ' + this.title;
    }
```

```
};
```

Again, calling a method can be done using the dot notation or the square bracket notation as we saw earlier:

```
podcast.toString();   OR
podcast['toString']();
```

I have to admit that this last notation looks a bit weird and I don't think it's a common practice.

Constructors

Instead of using object literals, the most common way I've seen so far for creating an object in JavaScript is by using a so called *constructor function*. A constructor function is just a regular function but with a slightly different naming convention. The name of a constructor function is generally defined using Pascal casing as opposed to the usual Camel casing (like the *toString*method as shown earlier).

```
function Podcast() {
    this.title = 'Astronomy Cast';
    this.description = 'A fact-based journey through the galaxy.';
    this.link = 'http://www.astronomycast.com';

    this.toString = function() {
       return 'Title: ' + this.title;
    }
}
```

In order to create a Podcast object, we simply use the *new* operator:

```
var podcast = new Podcast();
podcast.toString();
```

When a constructor function is invoked with the *new* operator, an object is always returned. By default, *this* points to the created object. The properties and methods are added to the object (referenced by *this*) after which the new object is returned. Because a constructor function is only a convention, you can also pass arguments to it just as with regular functions.

When an object is created using a constructor function, there's also a property named*constructor* that is set with a reference to the constructor function that was used for creating the respective object.

Encapsulation

In the example we've been using so far, it's fairly easy to replace property values and method implementations of a *Podcast* object.

```
var podcast = new Podcast();
podcast.title = 'The Simpsons';
podcast.toString = function() {
    return 'Doh';
}
```

Suppose we want our *Podcast* objects to be immutable. Unfortunately, JavaScript doesn't have a notation for private, protected, public or internal methods like C# does. So, if we want to hide the values of the properties on this object, we have to refactor them to regular variables and make use of closures (these are explained in the previous post).

```
function Podcast() {
    var _title = 'Astronomy Cast';
    var _description = 'A fact-based journey through the galaxy.';
    var _link = 'http://www.astronomycast.com';

    this.getTitle = function() {
        return _title;
    }

    this.getDescription = function() {
        return _description;
    }
```

```
    this.getLink = function() {
        return _link;
    }

    this.toString: function() {
       return 'Title: ' + _title;
    }
}
```

The 'public' methods have access to the 'private' variables while the latter are not exposed to the externals of the object. These public methods that have access to the private members are also called *privileged methods*.

```
var podcast = new Podcast();
console.log(podcast.getTitle());
console.log(podcast._title);    // undefined
```

You probably want to watch our for these privileged methods returning a private member that holds either an object or an array. Because these are passed by reference, the client code can still change the private member. To prevent this from happening, you might want to consider returning a copy of the object or array.

We can have private functions as well using the same approach as for private variables:

```
function Podcast() {
    ...

    // Private function
    function download() {
        ...
    }

    function reliesOnDownload() {

        ...

        download();

        ...

    }

    ...
}
```

Suppose we want to make the *download* method publicly available. But on the other hand, we also have some other methods in our *Podcast* object (like the *reliesOnDownload method*) that make use of the *download* method, relying on its robust functionality. So making this method publicly available can jeopardize the correct working of these other methods if some client decides to replace our *download* method with its own buggy implementation or even worse, deletes it completely. We can't have that, of course.

We mentioned earlier that constructor functions implicitly return *this*. But we can return our own custom object as well and we can use this to solve our problem of elevating private methods.

```
function Podcast() {
    var _title = 'Astronomy Cast';
    var _description = 'A fact-based journey through the galaxy.';
    var _link = 'http://www.astronomycast.com';

    function getTitle() {
        return _title;
    }
```

```
    function getDescription() {
        return _description;
    }

    function getLink() {
        return _link;
    }

    function toString() {
        // Now we can safely rely on the getTitle() accessor as well.
        return 'Title: ' + getTitle();
    }

    function download() {
        // Some highly resilient implementation ...
    }

    function reliesOnDownload() {
       // Relies on our own implementation of the download() method
       download();

       ...   }

    return {
        getTitle: getTitle,
        getDescription: getDescription,
        getLink: getLink,
        toString: toString,
        download: download
    }
}
```

The net result here is that we safely exposed our private methods to any outside code. Clients can still replace the download method on the custom object that we explicitly returned from the constructor function, but at the very least we can safely rely on our own implementation.

Closing

There you go, just some quick trivia on objects in JavaScript. I cannot emphasize enough how powerful JavaScript really is and that learning about this great programming language is a tremendous experience.