


## Basic JavaScript Part 3 : Prototypes

 December 3rd, 2010

In previous blog posts, I talked about the rich capabilities of [functions](#) and [objects](#) in JavaScript. For this post I want to briefly touch on the concept of prototypes. Having a decent understanding of prototypes in JavaScript is highly recommended as they are a very important part of the language. I have to admit that I'm still trying to fully get my head around the concept of prototypes, but writing this blog post is part of my learning process :-).

As you probably know, JavaScript is not a 'classical' language but a prototypal object language. This means that pretty much everything is an object, including functions. Every function has a property named *prototype*. This property is set to an empty object as soon as the former object gets created. As with every object we can augment it with our own methods.

In a [previous post](#), I showed how to use constructor functions for creating new objects. Have a look at the following simple constructor function.

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

```
var podcast1 = new Podcast('Astronomy cast', 'http:// ...');  
var podcast2 = new Podcast('jQuery podcast', 'http:// ...');
```

This constructor function adds two properties and one method to the objects that we created. Suppose that we developed a magnificent method for downloading the podcast itself. The most obvious place to put this code is in the constructor function as we did with the *toString* method. But we can also add this new method to the prototype of our constructor function.

```
Podcast.prototype.download = function() {  
    console.log("Downloading podcast ...");  
}
```

When a new *Podcast* object is created, this new object will 'inherit' the *download* method from the prototype of the constructor function and becomes available for use. In fact, *Podcast* objects that were created before the new function was added to the prototype of the constructor function also get this new method! Take a look at the following sample code:

```
var podcast1 = new Podcast('Astronomy cast', 'http:// ...');  
console.log(typeof podcast1.download);           // outputs 'undefined'
```

```
Podcast.prototype.download = function() {  
    console.log("Downloading podcast ...");  
}
```

```
var podcast2 = new Podcast('jQuery podcast', , 'http:// ...');
```

```
console.log(typeof podcast1.download);           // outputs 'function'  
console.log(typeof podcast2.download);           // outputs 'function'
```

When the *Podcast* constructor function gets augmented with the *download* function, the previously created object now also exposes the newly added function. I find this a quite fascinating feature.

As already mentioned, we can now simply call the *download* method that we added to the prototype.

```
var podcast = new Podcast('Railscasts', 'http:// ...');  
podcast.download();
```

Even though the *download* method is now available for every object created through

the *Podcast* constructor function, that doesn't mean that this new method is 'owned' by the created *podcast* object itself.

```
var podcast = new Podcast('Railscasts', 'http:// ...');
console.log(podcast.hasOwnProperty('download'));    // outputs 'false'
```

When the *download* method is called, the JavaScript engine first looks at the methods of the *podcast* object which doesn't seem have this method. Next thing, the engine identifies the prototype of the constructor function used for creating the *podcast* object. If the engine can find the method in the prototype object then this method will be called.

Besides the *prototype* property, every object also has a property named *constructor* that contains a reference to the constructor function used for creating the object. The code snippet shown earlier therefore resolves into something like this:

```
var podcast = new Podcast('Railscasts', 'http:// ...');
// outputs 'true'
console.log(podcast.constructor.prototype.hasOwnProperty('download'));
podcast.constructor.prototype.download();
```

As I just mentioned, every object has a *constructor* property. Because the prototype property of the constructor function holds a reference to an object, that means that it also has a constructor which has a prototype of its own, etc ... . The engine goes up the prototype chain searching for a requested method or property until it finds what needs to be called or until it reaches the root prototype, which is *Object.prototype*.

When a *download* method is added to the *Podcast* constructor function, then this method will take precedence over the *download* method of the prototype. This is illustrated by the following code sample:

```
function Podcast(title, url) {
    this.title = title;
    this.url = url;

    this.download = function() {
        console.log('Own download function.');
```

```
    }

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

Podcast.prototype.download = function() {
    console.log("Prototype download function.");
}
```

```
var podcast = new Podcast('Railscasts', 'http:// ...');
podcast.download();    // Outputs 'Own download function.'
```

These are the very basics of prototypes in JavaScript. I really enjoy learning JavaScript as it broadens my perspective on programming languages.

Until next time.