Forget everything you know about object-oriented programming. Instead, I want you to think about race cars. Yes – race cars.

Recently I was watching the 24 Hours of Le Mans –a popular racing event in France. The fastest cars in the race are the Le Mans Prototypes. Although these cars are built by car manufacturers like Audi and Peugeot, they are not cars you'll see on the streets and highways of your home town. They are built exclusively for high-speed endurance racing.

Manufacturers put an enormous amount of money into researching, engineering, and building these prototype cars, and the engineers are always trying to find an edge. They experiment with alloys, biofuels, brake technologies, tire compounds, and safety features. Over time, some of these experiments are refined and make their way into mainstream production cars. Some of the technology in your car made its first appearance in a racing prototype.

You could say mainstream cars *inherit* technology from racing *prototypes*.

And now we've set the mood to talk about prototypes and inheritance in JavaScript. It's not like the classical inheritance you might have learned about in C++, Java, or C#, but it's just as powerful and arguably more flexible.

Of Objects and Classes

JavaScript is full of objects, and I mean objects in the classical sense. An object marries state and behavior into a single entity. For example, an array in JavaScript is an object with values and also methods like push, reverse, and pop.

```
1. var myArray = [1, 2];
2. myArray.push(3);
3. myArray.reverse();
4. myArray.pop();
5. var length = myArray.length;
```

The question is – where does a method like push come from? The static languages we mentioned earlier use class definitions to define the structure of an object, but JavaScript is a classless language and there is no Array "class" defining the methods for every array object. Because JavaScript is dynamic, we can arbitrarily place methods into an object on an as-needed basis. For example, the following code defines an object to represent a point in two dimensional space, and includes an add method.

```
1. var point = {
2.     x : 10,
3.     y : 5,
4.     add: function(otherPoint) {
5.         this.x += otherPoint.x;
6.         this.y += otherPoint.y;
7.     }
8. };
```

This approach doesn't scale, however. We want to make sure *every* point object has an add method. We also want *all* point objects to share a single implementation of the add function instead of placing an add function into every point object. This is where prototypes come into play.

Of Prototypes

Every object in JavaScript holds a hidden piece of state – a reference to another object known as the object's prototype. The array we created earlier references a prototype object, and so does the point object we created. I say the prototype reference is a hidden, but there are implementations of ECMAScript (JavaScript's formal name) that allow us to grab this prototype reference using an object's __proto__ property (Google Chrome, for example). Conceptually we can think of the object - prototype relationship like the diagram in figure 1.

Figure 1

Moving forward as developers, we will be able to use the Object.getPrototypeOf function instead of the __proto__ property to inspect an object's prototype reference. At the time of writing, we can use Object.getPrototypeOf in Google Chrome, Firefox, and IE9. More browsers will implement this feature in the future, as it is now part of the ECMAScript standard. We can use the following code to prove that the myArray and point objects we've created truly reference two different prototype objects.

```
1. Object.getPrototypeOf(point) != Object.getPrototypeOf(myArray);
```

For the rest of this article I'll use __proto__ and Object.getPrototypeOf interchangeably, primarily because __proto__ is easier to look at in diagrams and sentences. Just keep in mind it isn't standard, and **Object.getPrototypeOf** is the recommended technique to uncover an object's prototype.

What Makes Prototypes Special?

We've yet to answer the question: where does an array's push method come from? It turns out the answer is in the prototype object for myArray. Figure 2 is a screenshot of the script debugger in Chrome. We've invoked Object.getPrototypeOf for myArray to inspect its prototype object.

Figure 2

Notice the prototype object for myArray includes a number of functions, including the push, pop, and reverse methods we invoked in our opening code sample. So the prototype object is the one holding the push method, but how does this method get invoked through myArray?

```
1. myArray.push(3);
```

The first step in understanding how this works is to recognize that prototypes are not special. Prototypes are just objects. We can give them methods, give them properties, and treat them as equal to any other JavaScript object. However, to paraphrase a pig in George Orwell's novel *Animal Farm* – all objects are equal, but some objects are more equal than others.

Prototype objects in JavaScript *are* special because of the following rule. When we tell JavaScript we want to invoke the push method on an object, or read the x property on an object, the runtime first looks in the object itself. If the runtime doesn't find what we want, it follows the __proto__ reference and looks for the member in the object's prototype. When we invoke push on myArray, JavaScript doesn't find push in the myArray object, but it does find push in myArray's prototype object and JavaScript invokes this method (see figure 3).

Figure 3

The behavior I've described means any method or property in an object's prototype is essentially inherited by the object itself. We do not need classes to implement inheritance in JavaScript. Just like my car inherits technology from a prototype race car, a JavaScript object inherits features from a prototype object.

Figure 3 also shows us that each array object can also maintain its own state and members. If we ask myArray for the value of its length property, JavaScript will find the value of the length property in myArray and never look at the prototype. We can use this feature to "override" methods by placing a method like push into the object itself. Doing so would effectively hide the push implementation in the prototype.

Sharing Prototypes

The true magic of prototypes in JavaScript is how multiple objects can maintain references to the same prototype object. For example, if we create two arrays:

```
1. var myArray = [1, 2];
2. var yourArray = [4, 5, 6];
```

Then both arrays will share the same prototype object, and following code evaluates to true:

```
1. Object.getPrototypeOf(myArray) === Object.getPrototypeOf(yourArray);
```

If we invoke push on either array object, JavaScript will find the push method in the common prototype.

Figure 4

Prototype objects in JavaScript give us inheritance, and they allow us to share method implementations, too. Prototypes also chain. In other words, since a prototype object is just an object, then a prototype object can maintain a reference to another prototype object.

You can see this if you revisit figure 2 and see how the prototype's __proto__ property is a non-null value pointing to yet another prototype. When JavaScript goes looking for a member, like the push method, it will follow these prototype references and inspect every object until it finds the member or reaches the end of the chain. Chaining opens up flexible avenues for inheritance and sharing.

The next question you might ask is: how do I set prototype references for my custom objects? For example, with the point object we were working with earlier – how can we put the add method into a prototype object and inherit the method from multiple point objects? Before we can answer this question, we'll need to look at functions.

Of Functions

Functions in JavaScript are objects, too. There are several important ramifications to this statement, and we won't be able to cover them all in this article, but the ability to assign a function to a variable and the ability to pass a function as a parameter to another function are both foundational paradigms in modern JavaScript programming.

What we need to focus on is the fact that functions are objects, and because functions are objects then functions themselves can have methods, properties, and reference a prototype object. Let's discuss the implications of the following code.

```
1. // this will return true:
2. typeof (Array) === "function"
3.
4. // and so will this:
5. Object.getPrototypeOf(Array) === Object.getPrototypeOf(function () { })
6.
7. // and this, too:
8. Array.prototype != null
```

The first line of code proves that Array is a function in JavaScript. Later we will see how we can invoke the Array function to create a new array object. The next line of code proves that the Array object uses the same prototype as any other function object – just like we saw how all array objects share the same prototype. The last line of code proves the Array function has a prototype property, and this prototype property points to a valid object. This prototype property is highly significant.

Every function object in JavaScript has a prototype property. **Do not** confuse this prototype property with the __proto__ property – they do not serve the same purpose or point to the same object.

```
1. // this returns true
2. Object.getPrototypeOf(Array) != Array.prototype
```

Array.__proto__ gives us the *prototype for Array* – think of this as the object the Array function inherits from.

Array.protoype, on the other hand, is the *prototype object for all arrays*. That is, it's the prototype object for array objects like myArray, and it contains the methods all arrays will inherit. We can write some code to prove this fact.

```
1. // true
2. Array.prototype == Object.getPrototypeOf(myArray)
3.
4. // also true
5. Array.prototype == Object.getPrototypeOf(yourArray);
```

We can also redraw our previous diagram with this new knowledge.

Figure 5

Given what we know, imagine we want to create a new object and make the new object behave like an array. One approach would be to use the following code.

```
1. // create a new, empty object
2. var o = {};
3.
4. // inherit from the same prototype as an array object
5. o.__proto__ = Array.prototype;
6.
7. // now we can invoke any of the array methods ...
8. o.push(3);
```

Although this code is interesting and useful, the problem is that not every JavaScript environment supports a writeable __proto__ property for objects. Fortunately, JavaScript does have a built-in and standard mechanism to create a new object and set the new object's __proto__ reference in a single operation - the "new" operator.

```
1. var o = new Array();
2. o.push(3);
```

The new operator in JavaScript has three essential tasks. First, it creates a new empty object. Next, it sets the new object's __proto__ property to match the prototype property of the function being invoked. Finally, the operator invokes the function and passes the new object as the "this" reference. If we were to expand out those last two lines of code, the script would look like this.

```
1. var o = {};
2. o.__proto__ = Array.prototype;
3. Array.call(o);
4. o.push(3);
```

The call method of a function allows you to invoke a function and specify the object to use as the "this" reference inside the function. Of course, the author of the function has to implement a function expecting to be used in this fashion. When the author creates such a function, we call it a constructor function.

Constructor Functions

A constructor function is just a regular JavaScript function object with two distinguishing characteristics.

1. The first letter of a constructor function is capitalized by convention (making it easy to identify constructor functions).
2. A constructor function expects to be used in conjunction with the new operator to construct objects.

Array is one example of a constructor function. The Array function expects you to use it with a new operator, and its first letter is a capital letter. JavaScript includes the Array function as a built in object, but anyone can author a constructor function. In fact, we've finally reached a point where we can write a constructor function for the point object we created earlier.

```
1.  var Point = function (x, y) {
2.      this.x = x;
3.      this.y = y;
4.      this.add = function (otherPoint) {
5.          this.x += otherPoint.x;
6.          this.y += otherPoint.y;
7.      }
8.  }
9.
10. var p1 = new Point(3, 4);
11. var p2 = new Point(8, 6);
12. p1.add(p2);
```

In the above code we are using the new operator and the Point function to construct points with x and y properties, and an add method. In memory you can think of the end result like figure 6.

Figure 6

The problem now is we still have an add method inside of each point. Applying what we know about prototypes and inheritance, we'd prefer to have the add method in Point.prototype instead of each point. To achieve inheritance of the add method, all we need to do is modify the Point.prototype object.

```
1.  var Point = function (x, y) {
2.      this.x = x;
3.      this.y = y;
4.  }
5.
6.  Point.prototype.add = function (otherPoint) {
7.      this.x += otherPoint.x;
8.      this.y += otherPoint.y;
```

```
}

var p1 = new Point(3, 4);
var p2 = new Point(8, 6);
p1.add(p2);
```

Voila! We've just applied prototypal inheritance in JavaScript!

Figure 7

Summary

I hope this article has helped you to demystify the concept of prototypes in JavaScript. We've seen how prototypes allow objects to inherit functionality from other objects, and we looked at one approach to building objects using the new operator and a constructor function. What we've covered is only a start to the power and flexibility of prototype objects. I encourage you to explore and learn new information about prototypes and the JavaScript language.

Also – drive safely. You never know what new technology those other cars on the road might inherit from their prototypes.