JavaScript Objects

John M. Morrison

January 17, 2017

Contents

0 A New Data Structure

JavaScript has a data structure called an *object*; these allow you to package data (properties) and functions (methods) into a single package of code. JavaScript provides a variety of ways to create objects; we will look at the simplest ones first.

One way to create objects is to stick-build them right at the scene. Let us demonstrate with a little console session. Line numbers have been added for convenience here

```
1       > obj = {}
2       Object {}
3       > obj.x = 3
4       3
5       > obj.y = 4
6       4
```

```
7         > obj
8         Object {x: 3, y: 4}
```

On line 1, we create an empty object. The console echoes back on line 2 that we, in fact, have an empty object. On lines 3 and 5, we give properties `x` and `y` to our object. On line 8, we see that we now have an object with property `x` set to 3 and `y` set to 4.

You can also create an object with an object literal in this manner.

```
other = {x:3, y:4};
Object {x: 3, y: 4}
```

Here is something to puzzle over.

```
other == obj
false
```

This is so because `==` tests for equality of identity of objects, not equality of value. The two objects hold the same value, but they are separate copies, so they are not the same object.

Notice that, if we want to create another object like this one, that we must repeat the entire stick-building process or we must create another object literal to do the job. What if our object is more complex than just being a container for two numbers? Can we make our object "smart" and endow it with behavior as well as properties? What if we are using this object frequently, leaving duplicate code everywhere?

Then answer to this is ECMA6 classes. Let us see what a class for points in the plane might look like. When we study the Canvas object, we will see that a point class will be very useful. Let us begin by creating an HTML file so we can open our JavaScript code in the console.

```
<!DOCTYPE HTML>
<html>
   <head>
       <title></title>
       <meta charset='utf-8'/>
       <script type="text/javascript"  src="objects.js"></script>
   </head>
   <body>
       <h2>Class Test</h2>
    </body>
</html>
```

We now introduce two new keywords, `class`, and `constructor`. A class is a blueprint for the creation of objects. So, let us begin by creating a very simple class for points.

```
class Point
{
    constructor(x,y)
    {
        this.x = x;
        this.y = y;
    }
}
```

We now inspect this in the console.

```
> var p = new Point(3,4);
undefined
> p
Point {x: 3, y: 4}
> typeof p
"object"
```

So, we can stamp out as many objects of this new type as we like by making the call `new Point(a,b)`. When we do this, the special method `constructor` is called; this oversees the birth of our point objects. They are born with the `x` and `y` we specify.

```
> p.x
3
> p.y
4
```

Our code becomes more readable. We are not just creating an ad hoc object; we are making a point.

We can also attach methods to objects. Let us make a nice string representation for a point. We will also make a method for calculating the distance to another point, and give default values to `x` and `y`.

```
class Point
{
    constructor(x = 0,y = 0)
    {
        this.x = x;
        this.y = y;
```

```
    }
    toString()
    {
        return "(" + this.x + ", " +  this.y + ")";
    }
    distanceTo(p)
    {
        return Math.hypot(this.x - p.x, this.y - p.y);
    }
}
```

Let us now take this for a test drive. You can now see how we attached methods to our point objects.

```
> var p = new Point(3,4);
undefined
> var q = new Point();
undefined
> p
Point {x: 3, y: 4}
> q
Point {x: 0, y: 0}
> p.toString()
"(3, 4)"
> q.toString()
"(0, 0)"
> p.distanceTo(q)
5
```

1 I demand my equal rights!

We have noticed that == for objects is just equality of identity. What if we want to test to see if two points are equal? We can attach a method for that. We introduce a new keyword `instanceof`.

We now add this method to our class.

```
    equals(that)
    {
        if( !(that instanceof Point))
        {
            return false;
        }
        return (this.x == that.x) && (this.y == that.y);
    }
```

The predicate in the `if` statement is checking to see if the object we are comparing to is a `Point`. If not, we return `false` and bail. This is called the `species test`. We reject equality if the object we are comparing to is of a different species. Once the species test is over, we then know the object we are comparing to is a point and we carry out the comparison. Let us now test-drive this.

```
> var p = new Point(3,4)
undefined
> var q = new Point(3,4);
undefined
> p.equals(q)
true
> var r = new Point();
undefined
> p.equals(r)
false
> p.equals("caterwaul")
false
> p.equals(56)
false
```

Notice that the last two tests show the species test at work. Comparing a point to a number or a string returns `false`.

2 Case Study: Introducing BallWorld

What we are going to create here is an app with a canvas on it that will be mouse sensitive. When clicked, a colored ball will appear that is centered at the point of the click. We will create controls to allow the user to choose the ball's color and its size.

Let us think about the balls. It might be handy if a ball knew its center, radius, and color. Then we could teach the ball how to draw itself.

Each time the user clicks, we will add the new ball to an array, then update the canvas by painting the background and all of the balls in the array.

Along the way, you will see interesting things done with CSS and HTML to create a polished appeareance for our application.

Our application will need to be aware of the background color and the color and size of the next ball to be created. When the user clicks in the canvas, the screen will update. We will avail ourselves of the `table` and `table-cell` CSS values to make things display nicely.

3 Roughing it in

Let us create the three files that will drive this application. Here is the start for the HTML.

```
<!doctype html>
<!--Author: Morrison-->

<html>
<head>
<title>BallWorld</title>
<link rel="stylesheet" href="BallWorld.css"/>
<script type="text/javascript" src="BallWorld.js">
</script>
</head>
<body onload="init();">
<h2>Ball World in</h2>

<p class="display">
<canvas id="ballPark" height="600" width="800"></canvas>
</p>
</body>
</html>
```

Now for the CSS; it does a few minimal things to make the app look decent.

```
/*Author: Morrison*/
h1, h2, .display
{
    text-align:center;
}
canvas
{
    border:solid 1px black;
}
body
{
    background-color:#FFF8E7;
}
```

Here is our JavaScript file. It just colors the canvas white.

```
function init()
{
    var c = document.getElementById("ballPark");
```

```
    var g = c.getContext("2d");
    g.fillStyle = "white";
    g.fillRect(0, 0, c.width, c.height);
}
```

We will now turn to making a class for the balls that are to appear on the screen. Recall that a ball needs to know its center, radius, and color.

```
class Ball
{
    constructor(x, y, radius, color)
    {
        this.x = x;
        this.y = y;
        this.radius = radius
        this.color=color;
    }
}
```

Paste the class into a console session and then reproduce this.

```
> b = new Ball(100,100, 50, "blue");
Ball {x: 100, y: 100, radius: 50, color: "blue"}
> b.x
100
> b.y
100
> b.radius
50
> b.color
"blue"
```

What happens if you use a hex code instead of a named color?

Now let us endow our ball objects with the ability to draw themselves. Here `g` is a graphics context. If `b` is a ball, we call this using `b.draw(g)`, where `g` is a graphics context. Place this at the top of `BallWorld.js`.

```
class Ball
{
    constructor(x, y, radius, color)
    {
        this.x = x;
        this.y = y;
        this.radius = radius
        this.color=color;
```

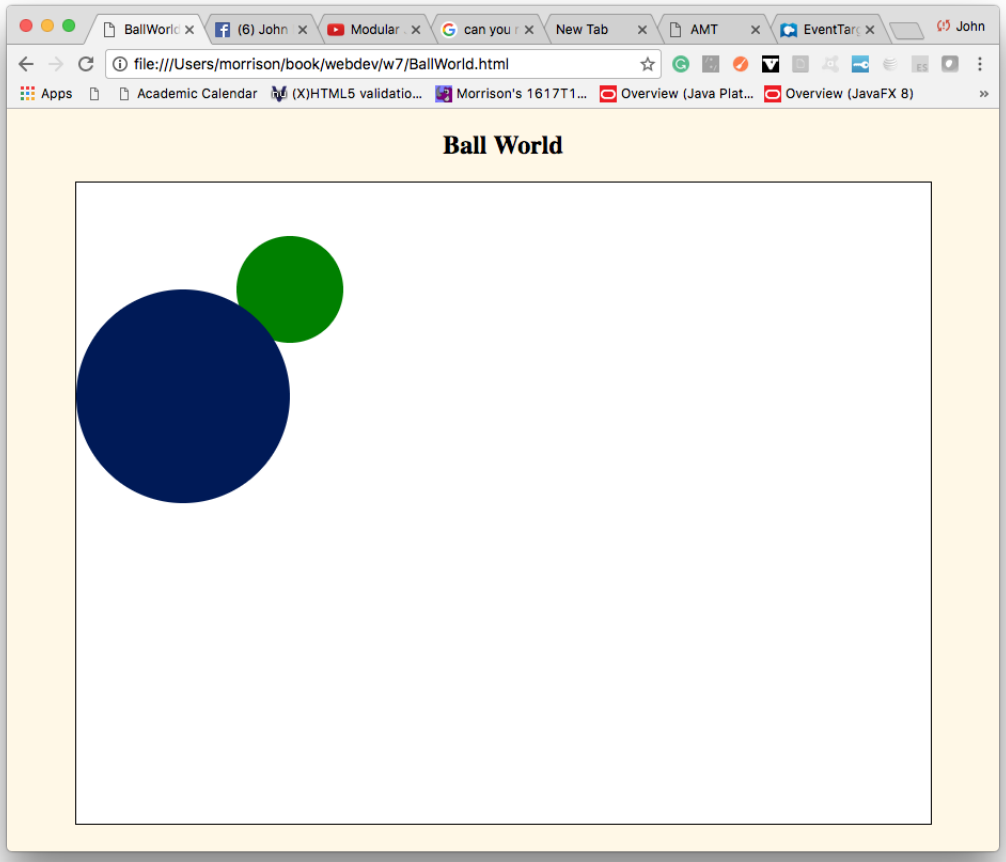
```
    }
    draw(g)
    {
        g.fillStyle = this.color;
        g.beginPath()
        g.arc(this.x, this.y, this.radius, 0, 2*Math.PI);
        g.fill();
    }
}
```

4 Making Balls Appear in the Canvas

Modify the `init()` method, creating a couple of balls. Then have the balls draw themselves.

```
function init()
{
    var c = document.getElementById("ballPark");
    var g = c.getContext("2d");
    g.fillStyle = "white";
    g.fillRect(0, 0, c.width, c.height);
    var b = new Ball(200,100,50,"green");
    b.draw(g);
    var c = new Ball(200,100,50,"green");
    c.draw(g);
}
```

Here is the result. Note the appearance of green and dook bloo balls on the canvas.



That's a start, but what we *really* want is for a click of the mouse to trigger the creation of a ball in the canvas.

Programming Exercises

1. Make a class for squares. Gauge a square's size on its size length. Give its center via properties `x` and `y`.
2. Draw several squares of various sizes on the canvas.
3. Can you do the same thing for an upward-facing equilateral triangle.

Here is where we learn about a new method for element objects, `addEventListener`. This requires two methods, one is a string describing the event's type and the other is a function which gets called whenever the event occurs.

We will attach this event listener to the canvas as follows.

```
c.addEventListener("click", function(){console.log("clicked");});
```

Insert this line into the `init` method. Then open the HTML file in the browser and open the developer tools. Look in the console; each time you click on the canvas, the word "clicked" will appear there. Voila! Our canvas recognizes mouse clicks.

Now we need to get he coördiates of a mouse click. Here is the correct tool. On some ancient browsers, `e.offsetX` is not defined; in those cases the or operator as used avails itself of the older `e.layerX`. These coördinates are returned in an object literal. Now let us put this to use.

```
/*
*   preconditon: e is a mouse event
*   postcondition: returns an object containing the coordinates
*   where the mouse occurred. These coordiantes are relative coordinates
*   in the element where the event occurred.
*/
function getRelativeCoords(e)
{
    return { x: e.offsetX || e.layerX, y: e.offsetY || e.layerY };
}
```

You will see reports on your click events in the console like so.