Chapter 8 JavaScript Objects

John M. Morrison

October 23, 2017

Contents

0 A New Data Structure **1**

1 I demand my equal rights! **5**

2 Case Study: Introducing BallWorld **6**

3 Roughing it in **6**

4 Making Balls Appear in the Canvas **9**

5 Making the Canvas Mouse-Sensitive **11**

6 Creating Application State and Making Balls Appear **12**

7 Inserting The Other Controls **14**

8 What About Colors? **15**

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}
```

You can even attach a function to an object as follows

```
obj.toOrigin = function(){return Math.hypot(this.x, this.y);}
```

Now call this function.

```
obj.toOrigin()
5
```

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 values, 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 and a single function (method)? What if we are using this object frequently, leaving duplicate code everywhere?

Then answer to this problem 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>Class Test</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 created by classes. 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 with a graphics pen. The center of a ball is a point; we will press our `Point` class into service.

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 appearance 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. Since we are going to use our `Point` class, we load it onto our page.

```
<!doctype html>
<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</h2>

<p class="display">
    <canvas id="panel" 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 for the `Point` class; it just contains the class itself

```
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);
    }
}
```

Here is `BallWorld.js`. It just colors the canvas white.

```
function init()
{
    var c = document.getElementById("panel");
    var g = c.getContext("2d");
    g.fillStyle = "white";
    g.fillRect(0, 0, c.width, c.height);
}
```

Since we will be using the `Point` class to support our efforts, modify the `script` tags to look like this. Since the script `BallWorld.js` depends on `Point.js`, `Point.js` must be loaded first.

```
<script type="text/javascript" src="Point.js"></script>
<script type="text/javascript" src="BallWorld.js"></script>
```

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(center, radius, color)
    {
        this.center = center;
        this.radius = radius
        this.color = color;
    }
}
```

Paste the class into a console session and then reproduce this. 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)`. Place this in the file `Ball.js`. After this, modify your `script` tags to look like this.

```
<script type="text/javascript" src="Point.js"></script>
<script type="text/javascript" src="Ball.js"></script>
<script type="text/javascript" src="BallWorld.js"></script>
```

Again order is important, since balls depend on points, and the application depends on both of these.

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

Load your page and get out the console. You can inspect your ball class.

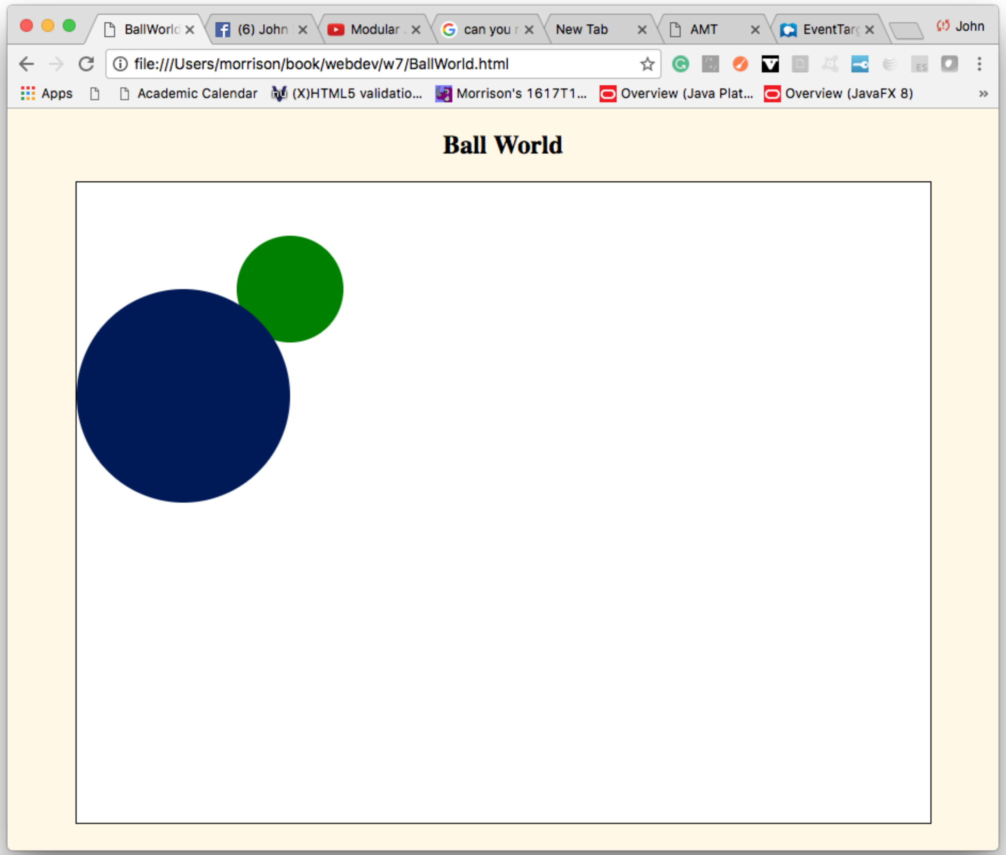
```
> b = new Ball(new Point(100, 100), 50, "blue");
Ball {center: Point, radius: 50, color: "blue"}
> console.log(b)
> console.log(b.center.x)
100
undefined
> console.log(b.center.y)
100
undefined
> console.log(b.radius)
50
undefined
> console.log(b.color)
blue
undefined
```

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("panel");
    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,"#001A57");
    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.

5 Making the Canvas Mouse-Sensitive

Here is where we learn about a new method for element objects, `addEventListener`. This requires two arguments, 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.

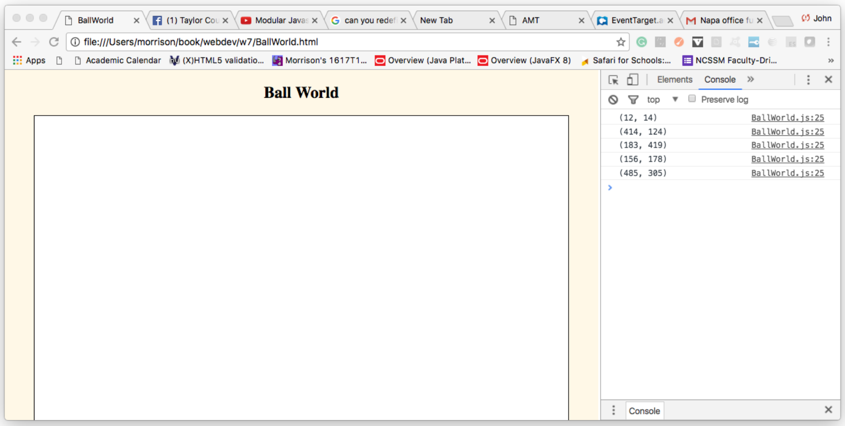
```
    c.addEventListener("click", function(){console.log("clicked");});
```

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.

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

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



```
(12, 14)         BallWorld.js:25
(414, 124)       BallWorld.js:25
(183, 419)       BallWorld.js:25
(156, 178)       BallWorld.js:25
(485, 305)       BallWorld.js:25
```

6 Creating Application State and Making Balls Appear

Let us now give our app some state. Add these variables to the top of the JavaScript file. Our default background will be white, our default color for the next ball will be black and we have created a list in which to store balls as they get created. Recall we can append elements to this list via the array method `push`.

```
var currentColor = "black";
var backgroundColor = "white";
var currentSize = 50;
var balls = [];
```

Each time a new ball is added, we will redraw the screen. To do this, we will add a new method called `refresh`. This method will draw the background, then draw the balls, in order, that reside in the list.

```
function refresh()
{
    var c = document.getElementById("panel");
    var g = c.getContext("2d");
    //color background
    g.fillStyle = backgroundColor;
    g.fillRect(0, 0, c.width, c.height);
    //tell each ball in the list to draw itself.
    for(var k = 0; k < balls.length; k++)
    {
        balls[k].draw(g);
    }
}
```

When the mouse is clicked, the following need to happen

1. We need to get the canvas coördiantes for the click
2. We need to create a new ball centered at the click.
3. We need to place this new ball on the list.
4. We need to refresh our drawing.

Where does this occur? It occurs inside of the function we created to react to mouse clicks.

So, now let us fill in the new code.

```
c.addEventListener("click", function(e)
{
    //get the click spot
    var p = getRelativeCoords(e);
    //create the new ball and push it onto the list.
    balls.push(new Ball(p, currentSize, currentColor));
    //refresh drawing
    refresh();
});
```

Also, modify your `init()` function as follows.

```
function init()
{
    c = document.getElementById("panel");
    var g = c.getContext("2d");
    g.fillStyle = "white";
    g.fillRect(0, 0, c.width, c.height)
    c.addEventListener("click", function(e)
    {
        //get the click spot
        var p = getRelativeCoordinates(e);
        //create the new ball and push it onto the list.
        balls.push(new Ball(p, currentSize, currentColor));
        //refresh drawing
        refresh();
    });
}
```

You can now click and have balls appear on the screen. They will be black on a white background.

7 Inserting The Other Controls

We have re-arrived in familiar territory. Now it is time to create the menus and the slider control. Let us begin by roughing in the necessary HTML. Place this after the `canvas` element.

```
<div id="controlPanel">
    <div class="row">
        <div class="cell">
        </div>
        <div class="cell">
        </div>
        <div class="cell">
        </div>
    </div>
</div>
```

Now open your CSS file. Insert these style directives.

```
#controlPanel
{
    display:table;
}
.row
{
    display:table-row;
}
.cell
{
    display:table-cell;
    padding:1em;
}
```

We will now begin by installing a slider control to control the size of the balls going on the screen. We put a label, a slider, and a paragraph that will display the size chosen by the slider.

```
<div id="controlPanel">
    <div class="row">
        <div class="cell">
```

```
            <p>Ball Radius:</p>
        </div>
        <div class="cell">
            <input type="range" min="0" max="500" value="50" id="sizeSlider">
            </input>
        </div>
        <div class="cell">
            <p id="sizeSpot">50</p>
        </div>
    </div>
</div>
```

Finally, we need to make the slider live. It needs to update the HTML in the size spot and it needs to update the `currentSize` state variable in the application.

```
function updateFromSizeSlider()
{
    var slider = document.getElementById("sizeSlider");
    var spot = document.getElementById("sizeSpot");
    spot.innerHTML = slider.value;
    //remember, properties come back as strings!
    currentSize = parseFloat(slider.value);
}
```

Now give your slider an `onchange` attribute.

```
<input type="range" min="0" max="500"
    value="50" id="sizeSlider" onchange="updateFromSizeSlider();">
</input>
```

Your size slider is now live.

8 What About Colors?

We need to mix colors for two purposes: the color of the next ball and the background color. We could control the colors with sliders and then have two buttons: the next ball button that causes the color we mix to become the color of the next ball, and the background button that changes the background color to the color we have mixed.