2-1, Lesson 1 - D3 Selections and Blocks

This module will introduce you to D3, the fantastic data visualization JavaScript library created by data journalist Mike Bostock (formerly of the New York Times). Learning D3 can be a challenge due to the unconventional nature of its logic, but you may find it richly rewarding as it opens up a world of web mapping and data graphic possibilities to you.

This unit of the course laboratory has a companion textbook. ***Interactive Data Visualization for the Web*** **(http://chimera.labs.oreilly.com/books/1230000000345/index.html)** by Scott Murray (2013) is available for free as a Web-based e-book at the linked URL. It is an excellent resource filled with easy-to-read-and-comprehend tutorials and examples. Each activity will begin by listing the corresponding book chapter(s) (if any) for your reference. Note that, although free, this book is copyrighted, and the course material does not duplicate its content.

> **Warning: D3 recently underwent a major update and is now on Version 4, which is NOT reflected in Murray's book**. Thus, if you try something out of the book and it doesn't work, locate the error and look up the problem method in the **D3-v4 Documentation (https://github.com/d3/d3/blob/master/API.md)** . You may also run into old examples on the web that use D3 Version 3; updating these to work with Version 4 should be a relatively straightforward matter of looking at which methods are throwing errors in the console and finding the right replacement methods in the v4 docs.

In the first activity of the module, we will investigate selections and blocks, the code structures that provide the backbone of D3 script. The second activity will expand on D3 selections to look at data joins, where the real magic of transforming data into DOM elements happens. The third activity of the module will introduce the concept of generator functions, the engines used by D3 to create eye-popping data graphics. We will use scale generators to position SVG elements on a chart and an axis generator to add annotation to the margins. As a final touch in the third activity, we will look at SVG text generation and D3 text formatting.

When you have finished this module, you should be able to:

- Create a selection and use D3 code blocks to make a basic SVG image
- Dynamically draw SVG elements using a data join
- Use scales to position SVG elements on a chart and annotate the chart with axes and text

Note! Reference chapters in Murray, 2013:

- Lesson 1: Chapter 5
- Lesson 2: Chapters 5 and 6
- Lesson 3: Chapters 7 and 8

1.1 Shaking Hands with D3

D3 (http://d3js.org/) stands for *Data-Driven Documents*. It is a JavaScript toolbox for using *data*—both manipulating it behind the scenes and turning it into stunning visualizations in the browser. To be more precise, D3 works by using data provided by the developer to shape and transform DOM elements (usually SVG elements). A good explanation of the rationale behind D3 is provided by the library's original author, Mike Bostock, in **this keynote talk** (https://vimeo.com/106198518) delivered to the 2014 Free and Open Source Software for Geospatial (FOSS4G) conference.

You can see hundreds of fantastic example visualizations created by D3 developers in the **D3 Gallery** (https://github.com/mbostock/d3/wiki/Gallery). Many of these examples include the code for their creation right on the page, making duplication and experimentation easier. A word of caution, however: you are unlikely to understand what you are looking at until you have gained an understanding of how D3 works. Likewise, the library's **API documentation** (https://github.com/d3/d3/blob/master/API.md), while thorough, can be difficult to parse if you are new to web development. Don't get discouraged. As you become more familiar with coding concepts used by D3, you will become better able to understand the wording of the API documentation and structure of the example code. Although there are a number of links to the documentation throughout this module, you may find Scott Murray's (2013) **textbook** (http://chimera.labs.oreilly.com/books/1230000000345/index.html) a more useful reference for now.

In this module, we will learn the core principles used by the library to build a simple data graphic. Important formatting rules are highlighted by bullet points. Applying these rules will keep your code neater and make it work better. All of these formatting rules and principles are summarized in the blog post **Ten Rules for Coding with D3** (https://northlandia.wordpress.com/2014/10/23/ten-best-practices-for-coding-with-d3/) by Carl Sack (2014).

To begin, you will need to copy your boilerplate web directory and rename the copy *d3-demo*. Then, download the library from the **D3 website** (http://d3js.org/), unzip it, place it in the *lib*folder of your new website, and add a script link to it in *index.html*. Create a *main.js*file for the *d3-demo* site, save it to the *js* folder, and add a second script link to it. Finally, create a *style.css* file, save it to the *css* folder, and link to it in *index.html*. For this demo, you will not need a *data* folder.

> **Practice!** Create a new web directory called *d3-demo*. Add *d3.js* to the *lib* folder, *main.js*to the js folder, and *style.css* to the *css* folder. Add links for each file in the appropriate places in *index.html*. Create a new Git repository for the directory and sync it with GitHub.

1.2 Selections

The core of D3—what allows its methods to interface with the DOM—is the **selection**. A D3 selection is a lot like a jQuery selection, but much more powerful. There are **two methods** (https://github.com/d3/d3-selection/blob/master/README.md#selecting-elements) used to create a selection: `d3.select()` (https://github.com/d3/d3-selection/blob/master/README.md#select)

and `d3.selectAll()` (https://github.com/d3/d3-selection/blob/master/README.md#selectAll) . As you might expect, the difference between the two—at least superficially—lies in how many markup elements are selected at once. `d3.select()` will only grab the *first* element in the DOM that matches the selector (recall that a selector is a string parameter that uses the same syntax as CSS to select an element, e.g., `"tagname"`, `".classname"`, `"#id"`, etc.). Subsequent methods chained to the selection will only affect that element. Conversely, `d3.selectAll()` grabs *all* markup elements in the DOM that match the selector and applies any subsequent methods to all of the selected elements.

We will come to see the reason for this distinction over the course of this activity and the next. For now, we will begin our demo script by using `d3.select()` to select the HTML `<body>` and D3's `.append()` method to add a new `<svg>` element, which will eventually hold our data graphic. First, let's make the selection in our *main.js* file (Example 1.1):

Example 1.1: selecting the `<body>` in *main.js*

JavaScript

```
 //execute script when window is loaded
window.onload = function(){

    var container = d3.select("body") //get the <body> element from the DOM

};
```

This selects the HTML `<body>` element from the DOM and returns it to the variable `container`. Notice that there is *no semicolon after the* `.select()` *method*. This is intentional, as we will be chaining more methods to it momentarily. D3 utilizes method chaining in a way that's similar to jQuery, but to an even greater extent.

At this stage, if you were to issue the statement `console.log(container)`, you would see a nested (2-dimensional) array with the `body` as the only element (Figure 1.1):

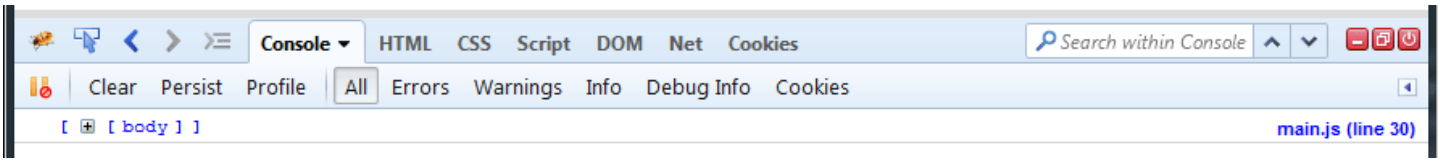

Figure 1.1: the D3 body selection

Having first created the `<body>` selection, we can alter the selection by applying **operators** to it. In the context of D3, operators are methods that work on selections. All of the operators used on a given selection are typically chained together. The formatting convention is to place each operator on its own line, indented one tab width (four spaces) from the initial line of the method chain. Because this tends to result in code that appears squarish or rectangular, a multi-line chain of D3 operators is referred to as a **code block** or block.

> **Rule!** Place each operator applied to a selection on its own line, indented one tab (four spaces) from the first line of the code block.

To add the `<svg>` container, we will use D3's `.append()` operator, creating our first block (Example 1.2):

Example 1.2: appending the `<svg>` to the `<body>` in *main.js*

JavaScript

```
//Example 1.1 line 3...container block
 var container = d3.select("body") //get the <body> element from the DOM
     .append("svg") //put a new svg in the body
```

If you now reload your *d3-demo* website and use the Inspector, you should be able to see the new, still-dimensionless SVG in the DOM (Figure 1.2):

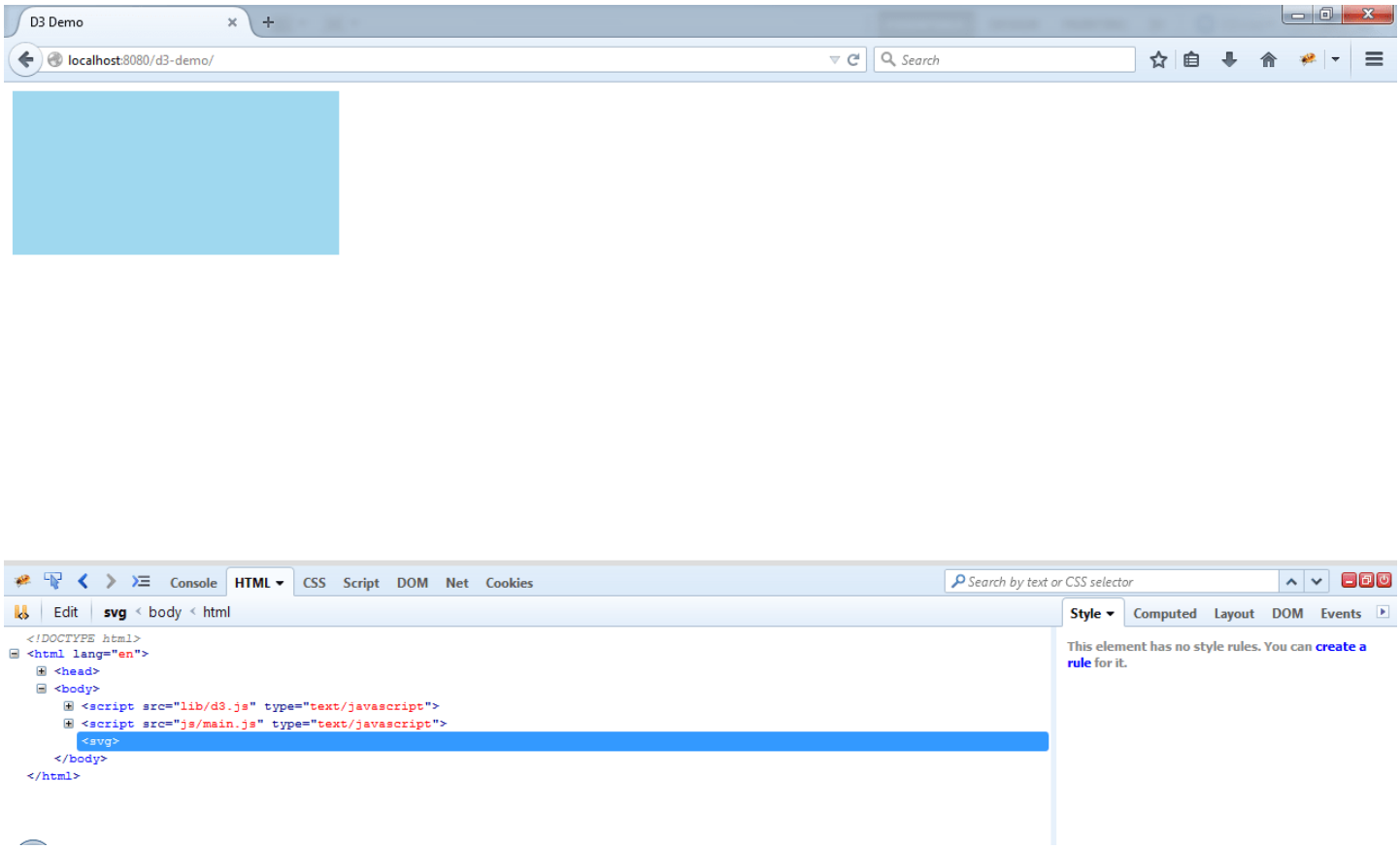

Figure 1.2: an SVG created using D3

Note that both jQuery and D3 have `.append()` methods. In this case, we know that the `.append()` method we are using belongs to D3 because the block starts with `d3`. Recall from previous Module how JavaScript object prototypes work: D3's `.append()` is a method of the `d3` object, just as jQuery's `.append()` is a method of the `jQuery` object (and its `$` alias). In any script that uses chain syntax or blocks (such as D3, jQuery, and Leaflet), the methods you can use in the chain depend on the library object referenced at the beginning of the chain (e.g., `d3`, `$`, or `L`). You can always figure out what library is being used by reading backwards up the chain or block to its beginning. If the beginning of the chain or block is a variable, you need to look at how that variable was created to discover which library is being used.

> **Rule!** In any method chain or block, only chain together methods belonging to the library referenced at the start of the chain.

The two libraries' `.append()` methods may look the same, but in reality there are some key differences. First, there is a slight difference in syntax: jQuery requires either an opening HTML tag (e.g., `.append("<svg>")`) or a full HTML string to append the element, whereas D3 requires a string with only the element name (e.g., `.append("svg")`). Whereas it is common to pass a full HTML string to jQuery's `.append()` method (e.g., `.append("<svg width='100' height='100'>")`), this simply won't work in D3. The second difference is that a jQuery `<body>` selection (i.e., `$("body")`) would continue to return the `<body>` element even after the `.append()` method is chained to it, whereas D3's `.append()` operator *changes the selection.*

What does this mean? To demonstrate, if you now add the statement `console.log(container)` below the selection, you should see this (Figure 1.3):

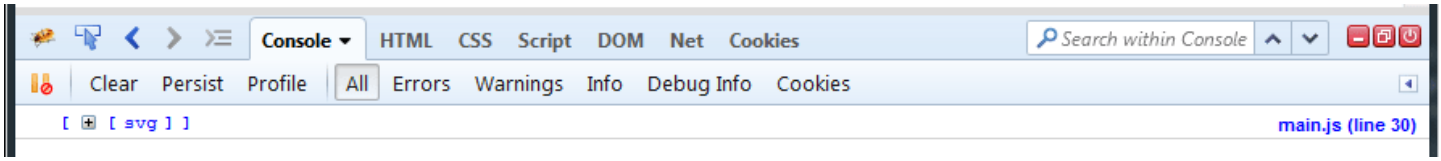

Figure 1.3: the selection now returns the new SVG

Compare Figure 1.3 to Figure 1.1. The selection has changed from holding the `<body>` element to holding the newly-created `<svg>` element.

1.3 Operands

Now the variable to which the block is assigned matches the selected element, or **operand**: the container `<svg>`. The operand is the element that each subsequent operator will operate on. To make the purpose of each block clear, it is a good idea to assign each block to a variable based on the operand that is returned when the end of the block is reached. This variable serves as the **block name**. Remember that it is very important to *only* place a semicolon at the *end* of a block, and not on each line, as a semicolon tells the interpreter that it has reached the end of a statement and will therefore break your method chain. Another way to think of the operand of a block is as whatever is returned when the interpreter reaches the semicolon.

> **Rule!** Only place a semicolon after the last line of a block. If your code results in errors, look for a wayward semicolon.

> **Rule!** Give each block a name by assigning it to a variable named for the operand it holds.

Now that the `<svg>` element is our operand, we can add operators to the block that manipulate that element. Recall from previous Module that every SVG requires `width` and `height` attributes to display. These values can be stored in separate variables that are passed as parameters to the operators. We can use D3's `.attr()` [(https://github.com/d3/d3-selection/blob/master/README.md#selection_attr)](https://github.com/d3/d3-selection/blob/master/README.md#selection_attr) operator to assign any attributes to markup elements (Example 1.3):

Example 1.3: Adding attributes to the `<svg>` element in *main.js*

JavaScript

```
    //SVG dimension variables
    var w = 900, h = 500;

     //Example 1.2 line 1...container block
    var container = d3.select("body") //get the <body> element from the DOM
        .append("svg") //put a new svg in the body
        .attr("width", w) //assign the width
        .attr("height", h) //assign the height
        .attr("class", "container") //always assign a class (as the block name) for styling and future select
ion
```

In addition to the dimensions, it is good practice to add a class name to each newly created element in the block so that it can be easily selected and manipulated by CSS or future D3 script (Example 1.3 line 9). Making the element's class name identical to the block name can help avoid confusion later in the script.

> **Rule!** Assign each newly created element a class name identical to the name of the block.

The last thing we will do to our container `<svg>` is to add an inline style, coloring the background so we can see the container on the page. Note that you could also do this in a CSS stylesheet by applying the style to the `container` class. To add a higher-priority inline style, we can simply use D3's `.style()` (https://github.com/d3/d3-selection/blob/master/README.md#selection_style) operator (the same way we would use jQuery's `.css()` method). We will use a semicolon to close out the block (Example 1.4):

Example 1.4: adding an inline style to the container in *main.js*

JavaScript

```
    //Example 1.3 line 4...container block
    var container = d3.select("body") //get the <body> element from the DOM
        .append("svg") //put a new svg in the body
        .attr("width", w) //assign the width
        .attr("height", h) //assign the height
        .attr("class", "container") //assign a class name
        .style("background-color", "rgba(0,0,0,0.2)"); //only put a semicolon at the end of the block!
```

Now you should see the SVG container on the page as well as using the Inspector (Figure 1.4):

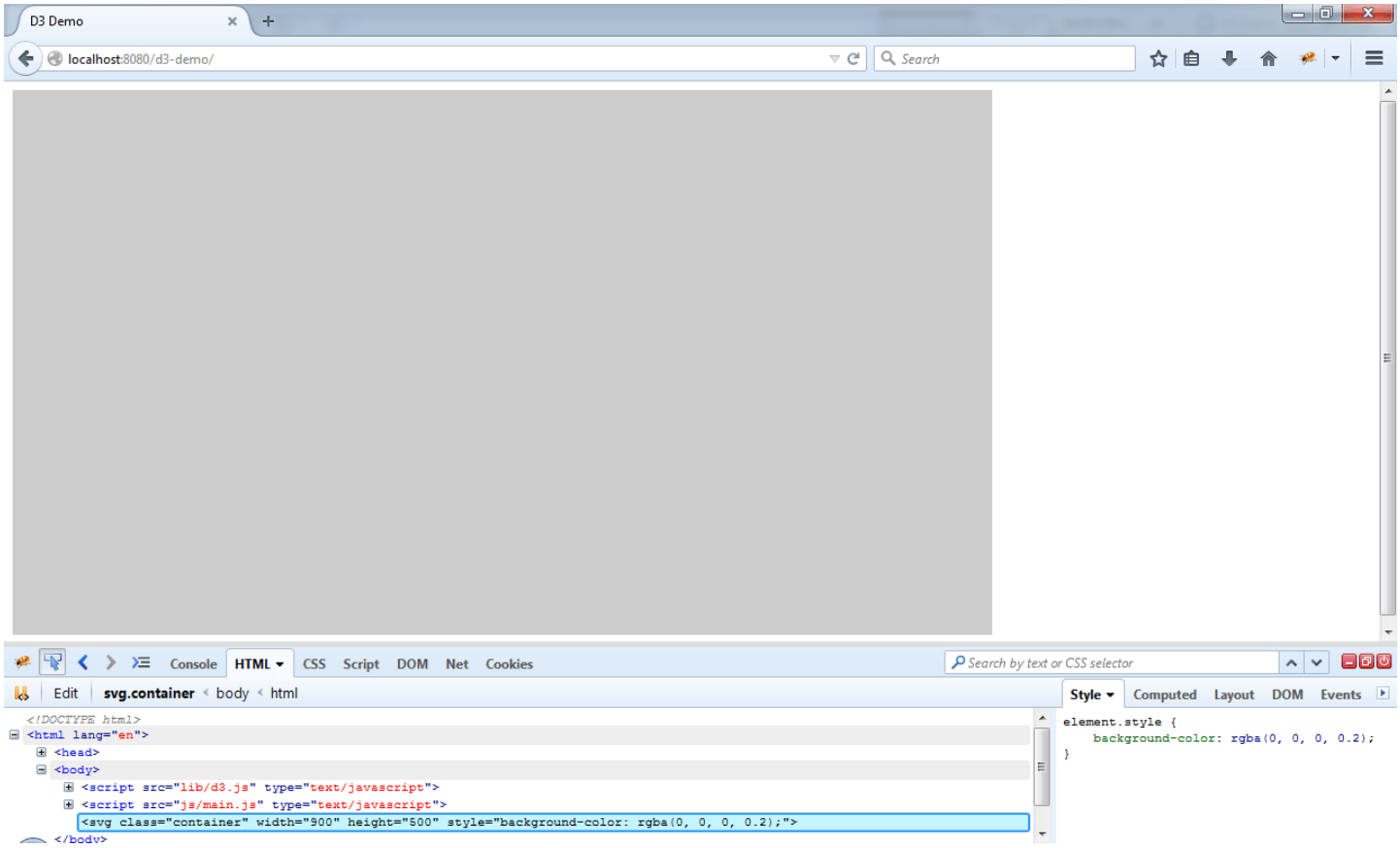

Figure 1.4: the SVG container on the page and in the DOM

Now that we have an SVG container, it's time to put something in it. Let's say, for example, that we want to add an inner rectangle to frame our graphics. We could just continue to add on to the *container* block, appending a new rectangle and adding operators to style it (Example 1.5):

Example 1.5: a block with too many operands in *main.js*

JavaScript

```
//Example 1.4 line 1...container block
var container = d3.select("body") //get the <body> element from the DOM
    .append("svg") //put a new svg in the body
    .attr("width", w) //assign the width
    .attr("height", h) //assign the height
    .attr("class", "container") //assign a class name
    .style("background-color", "rgba(0,0,0,0.2)") //svg background color
    .append("rect") //add a <rect> element
    .attr("width", 800) //rectangle width
    .attr("height", 400) //rectangle height

// <rect> is now the operand of the container block
```

The problem with Example 1.5 is that appending the `<rect>` element changes the operand again. Thus, what's now returned to the `container` variable is the `<rect>` element, not the `<svg>`. This means that *only* the `<rect>` element can be added to the `<svg>`; there is no longer a way to append other elements to the container unless you create a completely new selection. While this is possible to do, it is

much more convenient to "save" the existing `<svg>` selection in the `container` variable for multiple uses. This simply involves breaking the block and creating a second block for the inner rectangle (Example 1.6):

Example 1.6: correctly formatted blocks with only one change of operand each in *main.js*

JavaScript

```
//Example 1.5 line 1...container block
var container = d3.select("body") //get the <body> element from the DOM
    .append("svg") //put a new svg in the body
    .attr("width", w) //assign the width
    .attr("height", h) //assign the height
    .attr("class", "container") //assign a class name
    .style("background-color", "rgba(0,0,0,0.2)"); //svg background color

//innerRect block
var innerRect = container.append("rect") //put a new rect in the svg
    .attr(width, 800) //rectangle width
    .attr(height, 400) //rectangle height
```

Notice that the new `innerRect` block starts by accessing the `container` variable—which holds the `<svg>` as its operand—and appending the `<rect>` element to it. The `container` variable preserves its operand while the `<rect>` element becomes the operand of `innerRect`.

We can expand this principle into another general rule of thumb:

> **Rule!** Create only one new element per block.

In Example 1.7, the `container` block creates our `<svg>` and the `innerRect` block creates our `<rect>`. If we want to append something else new to either element, we will start a new block and name it for the new element we want to append.

1.4 Datum

So far, D3 selections and blocks may seem pretty straightforward—in fact, very similar to a version of jQuery with extended method chaining syntax. However, where D3 departs from this model is a special property of its selections: the **datum**.

In a selection created with `d3.select()` (or their children, such as `innerRect`), the **.datum()** (https://github.com/d3/d3-selection/blob/master/README.md#selection_datum) operator is used to **bind** a data value to the selection. The `.datum()` method takes a *single data value* (literally, a **datum** (http://dictionary.reference.com/browse/datum)) as a parameter and attaches it to the selection. Here's what this looks like (Example 1.7):

Example 1.7: binding a datum to the `innerRect` selection in *main.js*

JavaScript

```
//Example 1.6 line 9...innerRect block
var innerRect = container.append("rect") //put a new rect in the svg
    .datum(400)
    .attr(width, 800) //rectangle width
    .attr(height, 400) //rectangle height

console.log(innerRect);
```

In the Console, if you examine the inner array of the `innerRect` selection, you will see that there is a property called `__data__` attached to the `<rect>` element in the DOM. This property holds the datum. Figure 1.5 shows our new rectangle with a default black fill and the datum that is bound to it in the DOM:

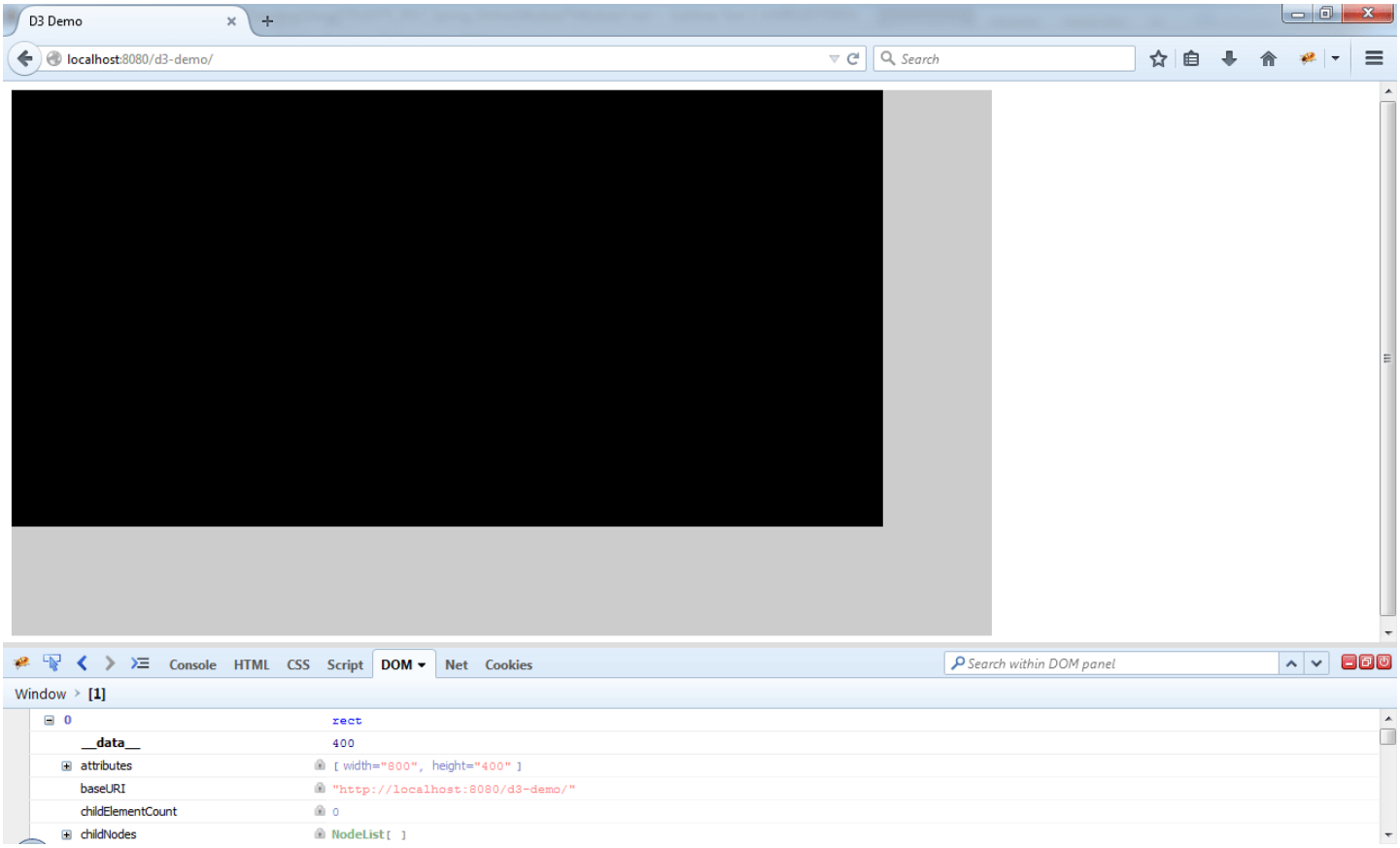

Figure 1.5: a rectangle and its datum

Now comes the fun part: actually *using* the datum. Any D3 operator method that requires a value as one of its parameters can make use of a datum or data that is bound to the block. This is done through an anonymous function that returns the datum (Example 1.8):

Example 1.8: using a datum in *main.js*

JavaScript

```
//Example 1.7 line 1...innerRect block
var innerRect = container.append("rect") //put a new rect in the svg
    .datum(400) //a single value is a datum
    .attr(width, function(d){ //rectangle width
        return d * 2; //400 * 2 = 800
```

```
    })
    .attr(height, function(d){ //rectangle height
        return d; //400
    })
```

On line 3 of Example 1.8, we bind the datum `400` to the `innerRect` block using `.datum(400)`. That makes `400` available as the parameter of any anonymous function used by an operator in the block. On lines 4 and 7, we name this parameter `d`. We could also name it `cheese` or `gobadgers`; either way it would contain the value `400`. Returning `d` in each function, or some derivative (e.g., `d * 2`), sends that value to the operator. Since the returned values in Example 1.10 match the hard-coded values in Example 1.9, you should not observe any visible changes to the rectangle in your browser yet.

To complete our rectangle, we will assign a few more `<rect>` (https://developer.mozilla.org/en-US/docs/Web/SVG/Element/rect) attributes to give the element a class name, position it, and style it differently from the default black fill (Example 1.9):

Example 1.9: adding rectangle attributes and style in *main.js*

JavaScript

```
//Example 1.8 line 1...innerRect block
var innerRect = container.append("rect")
    .datum(400) //a single value is a DATUM
    .attr("width", function(d){ //rectangle width
        return d * 2; //400 * 2 = 800
    })
    .attr("height", function(d){ //rectangle height
        return d; //400
    })
    .attr("class", "innerRect") //class name
    .attr("x", 50) //position from left on the x (horizontal) axis
    .attr("y", 50) //position from top on the y (vertical) axis
    .style("fill", "#FFFFFF"); //fill color
```

Now if you reload your browser window, the rectangle should be white and centered inside the SVG (Figure 1.6):

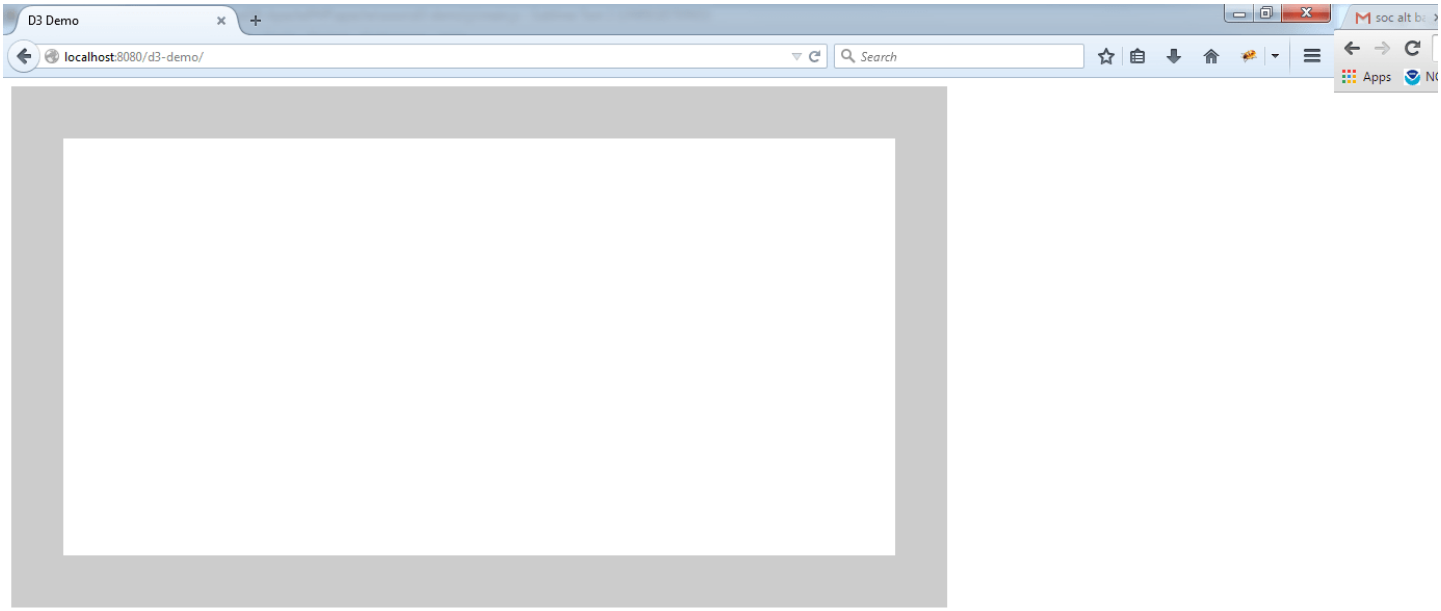

Figure 1.6: The finished inner rectangle

> **Practice!** Create a visible SVG graphic with an inner rectangle using properly formatted D3 code blocks in the *main.js* script of your *d3-demo* site.

> **Self-Check:**
>
> **1. When should you begin a new code block instead of adding operators onto an existing one?**
>
> **a.** when creating a new selection
> **b**. when adding a *second* `.append()` operator to the block
> **c.** when the operator you are adding changes the operand so that it no longer matches the block name
> **d.** all of the above
>
> **2. True/False: Each line in a code block should include a semicolon at the end of the line to properly separate the operators.**