Lecture 09

D3 Visualization

Zoran B. Djordjević

Why Visualize

- You have collected all the data and build all the Hadoop and Spark processes analyzing those data.
- Unfortunately, your audience does not care about the configuration issues and integration of impossible API-s. You audience wants graphs. Power Points will do for most of presentations, but eventually they will ask for web pages with interactive graphics.
- If you work with R use `ggplot2`, an excellent graphing package and you will be able to generate excellent presentations.
- If you need to built interactive Web graphs you need something else.
- It appears that `D3.js` is one of the most popular Java Script API for interactive graphics.
- Big Data community is using `D3.js` enthusiastically.
- This presentation is based on the book "Interactive Data Visualization" by Scott Murray, O'Reilly, March 2013.
- An excellent resource is a free book "D3 Tips and Tricks" by Malcolm Maclean. Download it from: https://leanpub.com/D3-Tips-and-Tricks

Objectives

- Fundamentals
- Setup
- Adding elements
- Chaining methods
- Binding data
- Using your data
- Drawing divs
- The power of data()
- An SVG primer
- Drawing SVGs
- Types of data
- Making a bar chart
- Making a scatterplot
- Scales
- Axes

Fundamentals

- There are things you need to learn or know since D3 is not alone.
  - You need to be familiar with or learn a few standard Web technologies: HTML, the DOM, JavaScrip and CSS
  - You have to have a little programming experience.
  - Have even heard of jQuery or written some JavaScript before
  - You are not easily scared by unknown acronyms like CSV, SVG, JSON, DNY, UGT, ….

HTML

- Hypertext Markup Language is used to structure content for web browsers. The simplest HTML page looks like this:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <h1>Page Title</h1>
        <p>This is an interesting paragraph.</p>
    </body>
</html>
```

DOM

- The Document Object Model refers to the hierarchical structure of HTML.
- Each bracketed tag is an *element*, and we refer to elements' relative relationships to each other in human terms: parent, child, sibling, ancestor, and descendant.
- In the HTML above, `<body>` is the parent element to both of its children, `<h1>` and `<p>` (which are siblings to each other).
- All elements on the page are descendants of `<html>`.
- Web browsers parse the DOM in order to make sense of page content.
- D3 can access and dynamically modify DOM properties, including properties of all nodes.
- D3 can dynamically modify the DOM tree of your HTML page and add new nodes at the runtime.

CSS

- Cascading Style Sheets are used to style the visual presentations of HTML pages. A simple CSS style sheet looks like this:

```
body {
    background-color: white;
    color: black;
}
```

- CSS styles consist of *selectors* and *rules*. Selectors identify specific elements to which styles will be applied:

```
h1          /* Selects level 1 headings                */
p           /* Selects paragraphs                      */
.caption    /* Selects elements with class "caption" */
#subnav     /* Selects element with ID "subnav"       */
```

- Rules are properties that, cumulatively, form the styles:

```
color: pink;
background-color: yellow;
margin: 10px;
padding: 25px;
```

- We connect selectors and rules using curly brackets:

```
p {
    font-size: 12px;
    line-height: 14px;
    color: black;
}
```

CSS

- D3 uses CSS-style selectors to identify elements on which to operate, so it’s important to understand how to use them.
- CSS rules can be included directly within the head of a document, like:

```
<head>
 <style type="text/css">
     p {
         font-family: sans-serif;
         color: lime;
     }
 </style>
</head>
```

- or saved in an external file with a .css suffix, and then referenced in the document’s head:

```
<head>
    <link rel="stylesheet" href="style.css">
</head>
```

JavaScript

- JavaScript is a dynamic scripting language that can instruct the browser to make changes to a page after it has already loaded.
- A note: JavaScript has become a very popular server-side language????
- Scripts can be included directly in HTML, between two script tags

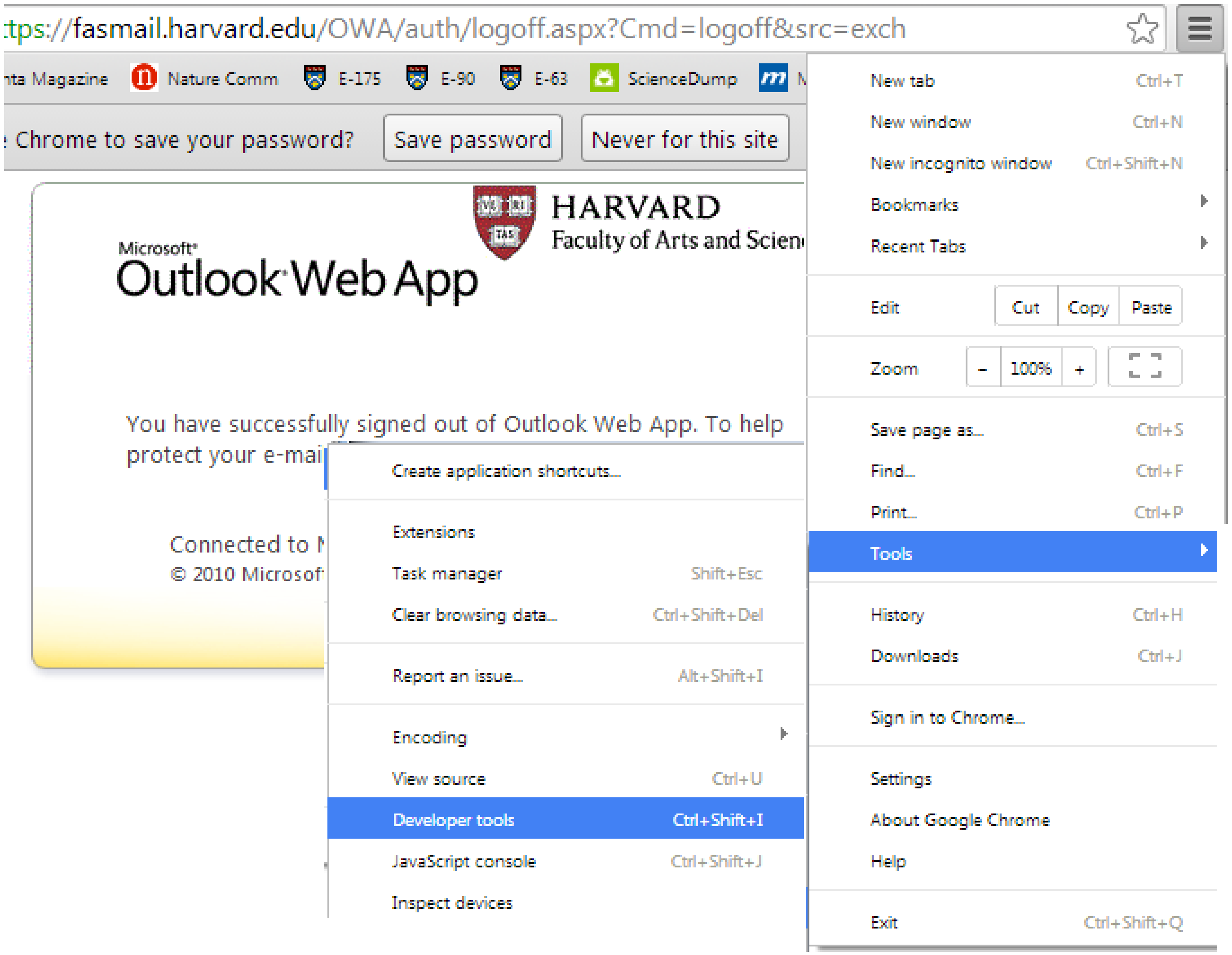
```
<body>
    <script type="text/javascript">
        alert("Hello, world!");
    </script>
</body>
```

- or stored in a separate file, and then referenced somewhere in HTML (in the `<head>`):

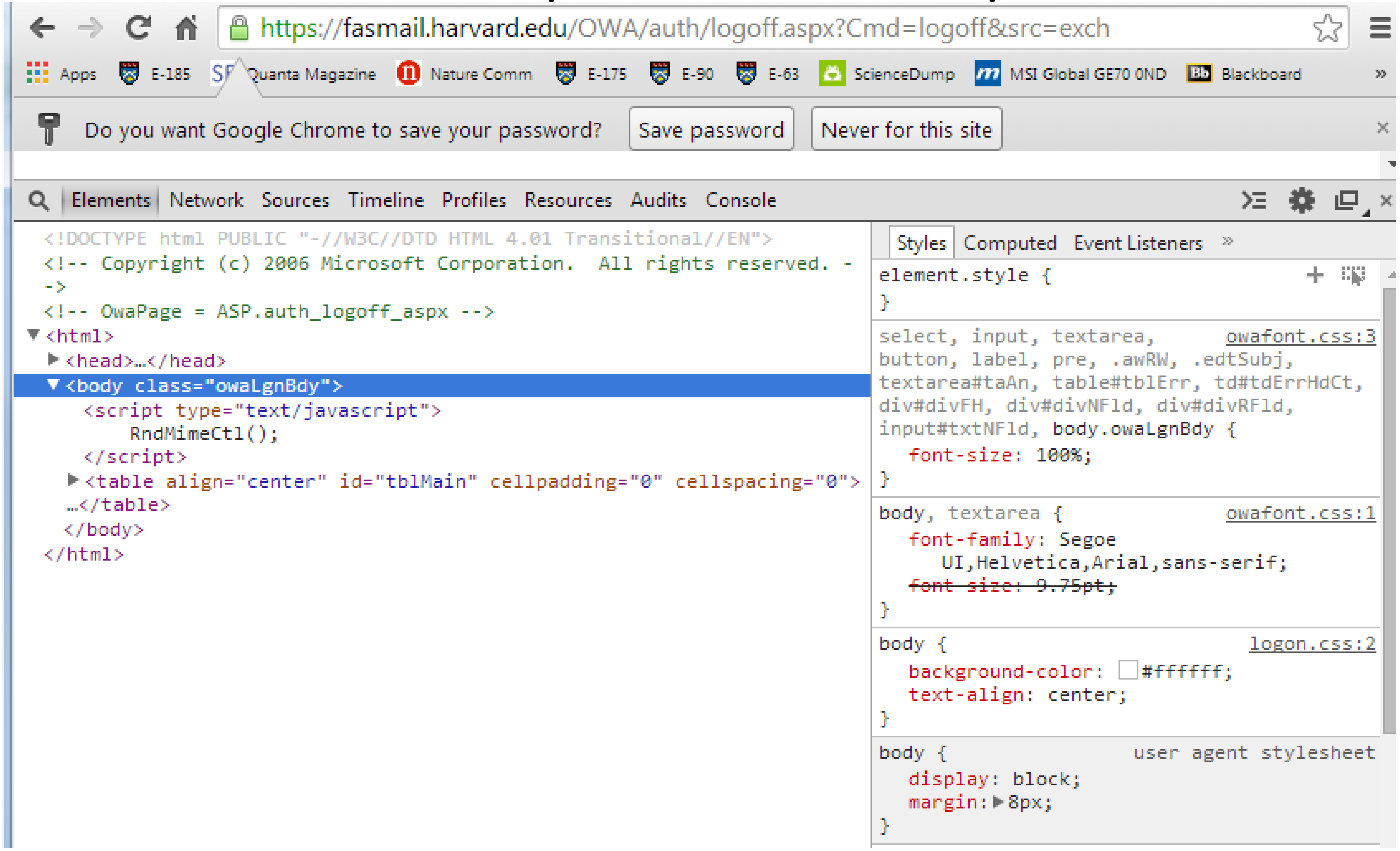
```
<head>
    <title>Page Title</title>
    <script type="text/javascript" src="myscript.js"></script>
</head>
```

Know your Browser's Developer Tools

- Newer versions of MS IE, Mozila, Chrome, Safari all have similar developer tools and JavaScript Console



Chrome Developer Tool, JavaScript Console



```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- Copyright (c) 2006 Microsoft Corporation.  All rights reserved. -
->
<!-- OwaPage = ASP.auth_logoff_aspx -->
▼<html>
 ▶<head>…</head>
 ▼<body class="owaLgnBdy">
    <script type="text/javascript">
        RndMimeCtl();
    </script>
   ▶<table align="center" id="tblMain" cellpadding="0" cellspacing="0">
   …</table>
  </body>
</html>
```

```
element.style {
}
select, input, textarea,                      owafont.css:3
button, label, pre, .awRW, .edtSubj,
textarea#taAn, table#tblErr, td#tdErrHdCt,
div#divFH, div#divNFld, div#divRFld,
input#txtNFld, body.owaLgnBdy {
   font-size: 100%;
}
body, textarea {                              owafont.css:1
   font-family: Segoe
      UI,Helvetica,Arial,sans-serif;
   ~~font-size: 9.75pt;~~
}
body {                                         logon.css:2
   background-color: □#ffffff;
   text-align: center;
}
body {                             user agent stylesheet
   display: block;
   margin:▶8px;
}
```

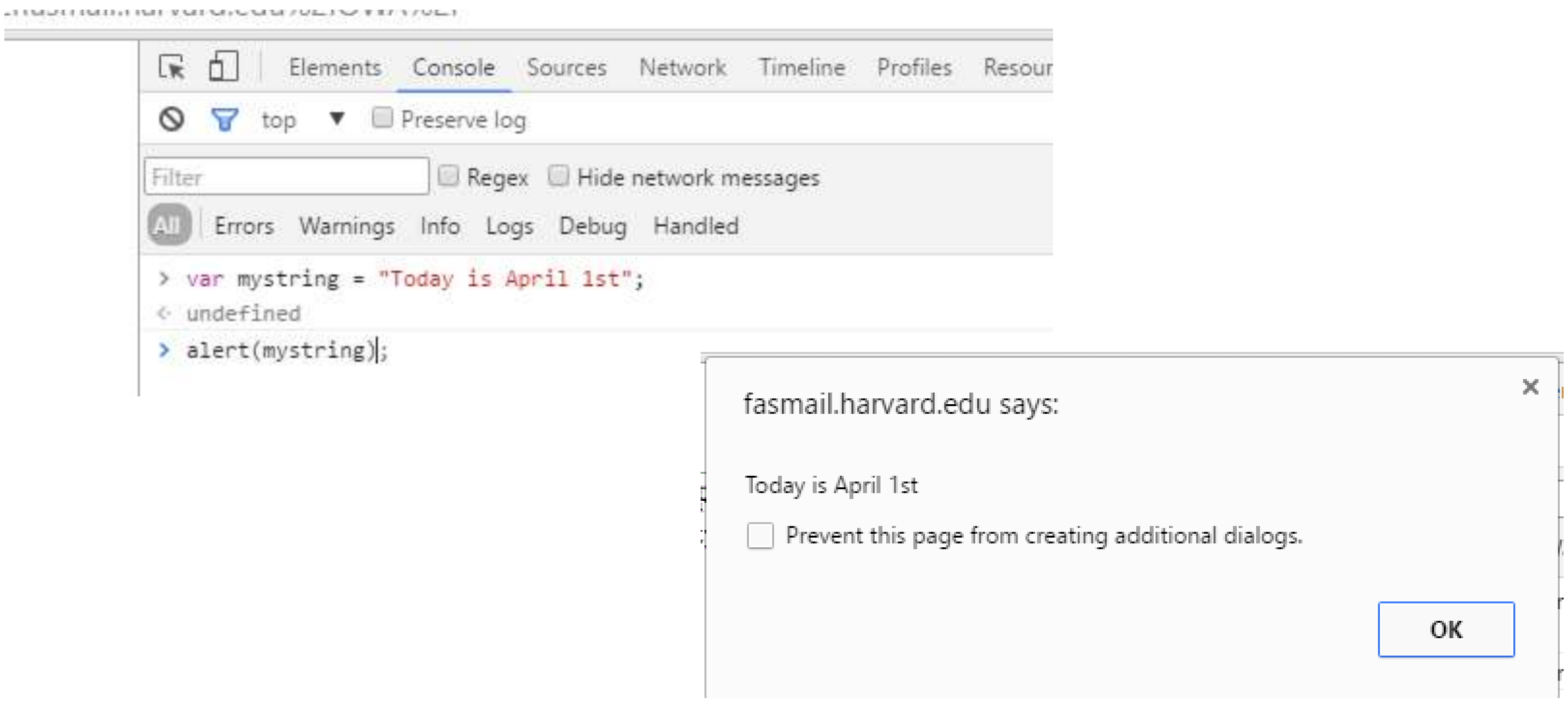
- While "View Source" shows you the original HTML source of the page, developer tool shows the *current state of the DOM*. This is useful.
- D3 code will modify DOM elements dynamically and we could see changes.

Chrome’s JavaScript Console

- You can type code in JavaScript Console. For example, at the command prompt ">", type the following:

```
> var mystring = "Today is April 1st"
> alert(mystring)
```

- The console will initiate an alert display:



```
> var mystring = "Today is April 1st";
<· undefined
> alert(mystring);
```

fasmail.harvard.edu says:

Today is April 1st

Prevent this page from creating additional dialogs.

SVG

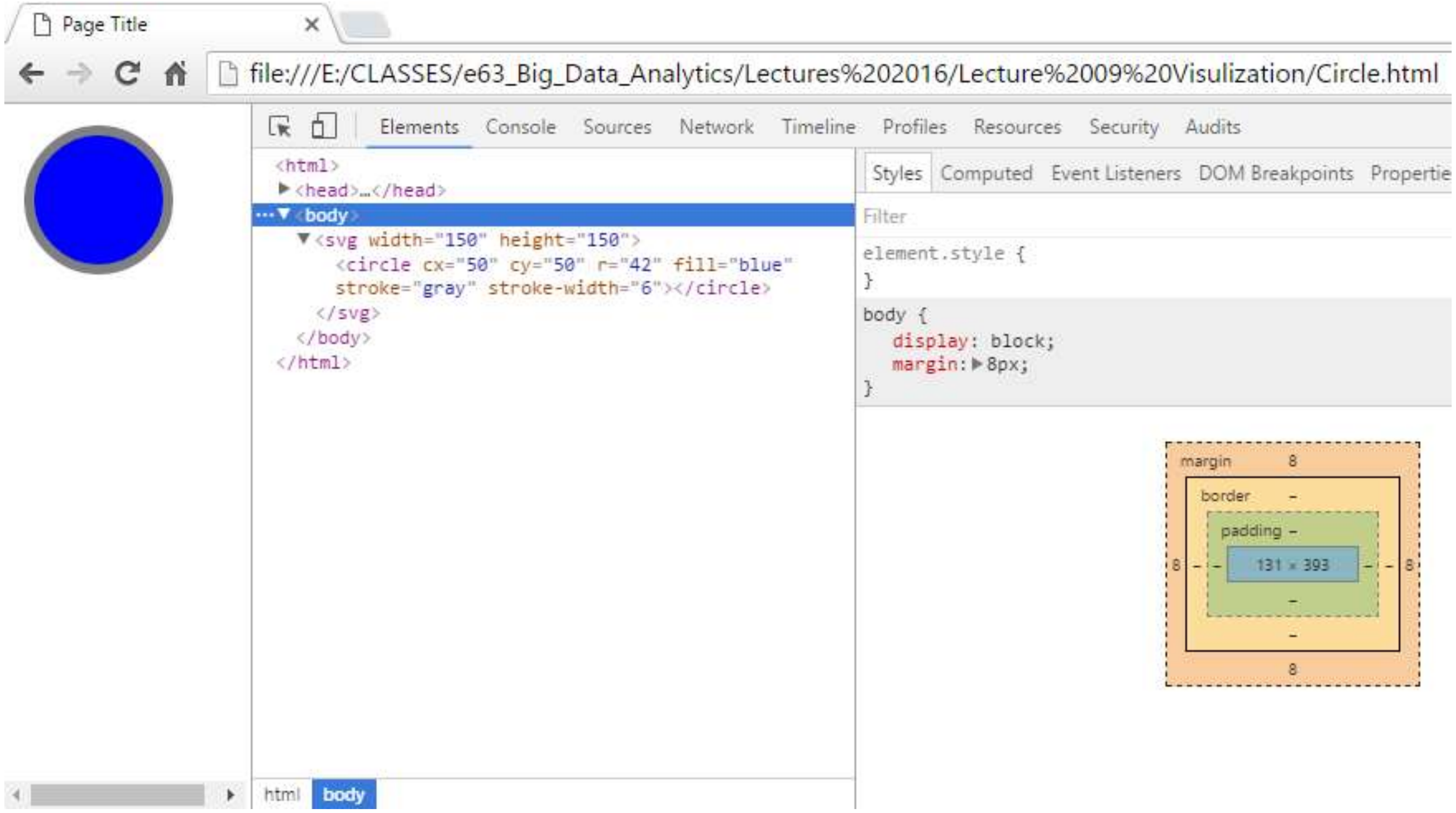
- Scalable Vector Graphics (SVG) provides a range of visual opportunities that are not possible with regular HTML elements.
- SVG is a text-based image format. You can specify what an SVG image should look like by writing simple markup code, similar to HTML tags.
- SVG code can be included directly within any HTML document. Web browsers have supported the SVG format for years (except for Internet Explorer).
- Here's a little circle

```
....
<body>
  <svg width="150" height="150">
     <circle cx="50" cy="50" r="52"
         fill="blue" stroke="gray" stroke-width="6"/>
  </svg>
</body>
. . . .
```

- D3 is at its best when rendering visuals as Scalable Vector Graphics.
- You're not required to use SVG with D3, though.

Circle.html

- When displayed with Chrome's developer tools open, Circle.html produces the following output.

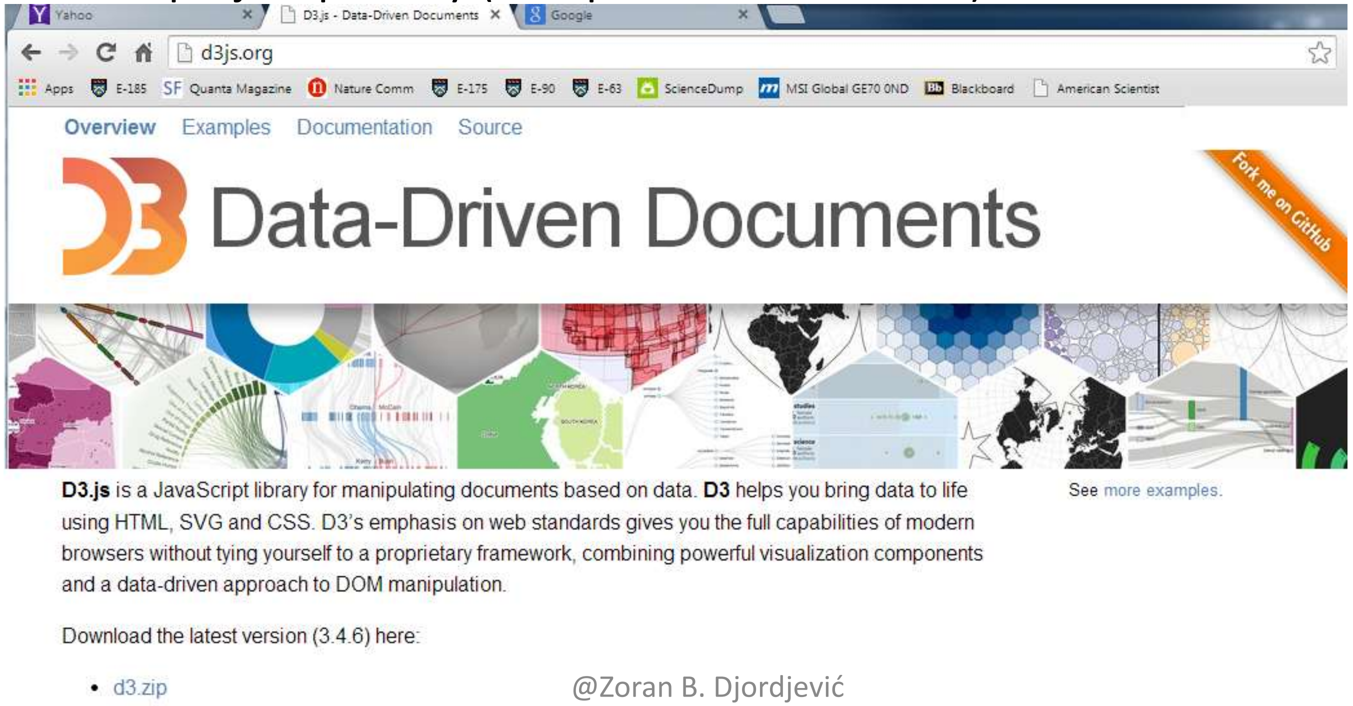


```
<html>
▶<head>…</head>
▼<body>
  ▼<svg width="150" height="150">
      <circle cx="50" cy="50" r="42" fill="blue"
      stroke="gray" stroke-width="6"></circle>
    </svg>
  </body>
</html>
```

```
element.style {
}
body {
   display: block;
   margin:▶8px;
}
```

Setup

- Start by creating a new folder for your project. Within that folder, create a sub-folder called d3. Then, from site: `http://d3js.org` download the latest version of d3.js into that sub-folder, and decompress the ZIP file.
- The current version of D3 is 3.5.5.
- D3 is also provided in a "minified" version, d3.v3.min.js, from which whitespace has been removed for smaller file size and faster load time.
- The functionality is the same. Typically, you use the regular version while developing (for friendlier debugging), and switch to the minified version once you launch the project publicly (for optimized load times).


Yahoo
D3.js - Data-Driven Documents
Google
d3js.org
Apps E-185 Quanta Magazine Nature Comm E-175 E-90 E-63 ScienceDump MSI Global GE70 0ND Blackboard American Scientist
Overview Examples Documentation Source
Data-Driven Documents
Fork me on GitHub

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

See more examples.

Download the latest version (3.4.6) here:

- d3.zip

An advantage of D3

- Modifying documents using the W3C DOM API is tedious: the method names are verbose, and the imperative approach requires manual iteration and bookkeeping of temporary state. For example, to change the text color of paragraph elements:

```
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
  var paragraph = paragraphs.item(i);
  paragraph.style.setProperty("color", "white", null);
}
```

- D3 employs a declarative approach, operating on arbitrary sets of nodes called selections. For example, you can rewrite the above loop as:

```
d3.selectAll("p").style("color", "white");
```

- Yet, you can still manipulate individual nodes as needed:

```
d3.select("body").style("background-color", "black");
```

There are other frameworks

- Look at sites like:
  - http://www.gapminder.org/tools/bubbles#_
  - http://vizhub.healthdata.org/le

Referencing D3 and Viewing your pages

- Create a simple HTML page within your project folder named `index.html`. Your folder structure should look something like this:

```
project/
      00_empty_page_template.html
      d3/
        d3.js
        d3.v3.min.js (optional)
```

- Populate file `00_empty_page_template.html` with the following code

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>D3 Test</title>
        <script type="text/javascript" src="d3/d3.js"/>
    </head>
    <body>
        <script type="text/javascript">
            // Your beautiful D3 code will go here
        </script>
    </body>
</html>
```

- Optionally, replace `<script …  src ="d3/d3.js"/>` with call to `d3.v3.min.js` library using

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

Viewing your code, Web Server

- You do not have to have a Web server and could open your files using Directory Explorer and the browser. Some browsers would not open local files as a security precaution. Any Web Server will serve.
- For example, you can download and install Python 3.X.X. Place `%PYTHON_HOME%`, e.g. `c:\Python34`, in your `%PATH%` and then start a local server in your project directory

```
$cd ..\project
$ python -m SimpleHTTPServer 8888 &
```

- This will activate a web server on port 8888.
- In your browser, you could type

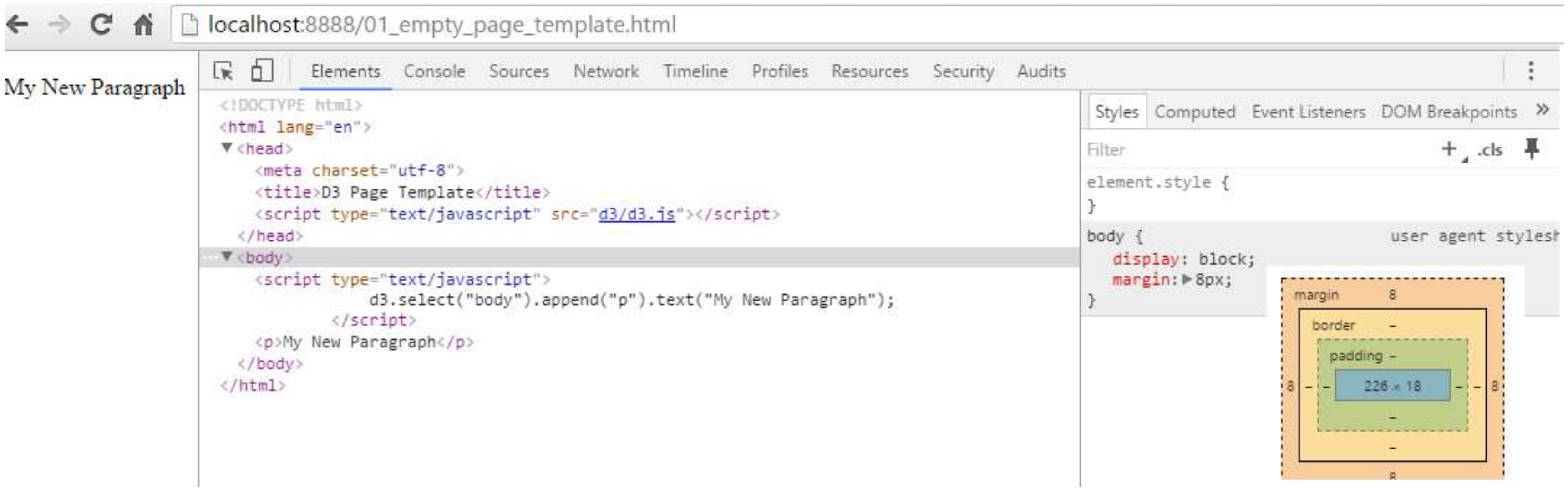
```
http://localhost:8888/00_empty_page_template.html
```

- and the requested page will render.
- There is nothing on it???

Adding a DOM Element

- One of the first steps in use of D3 is creation of a new DOM element.
- Typically, this will be an SVG object for rendering a data visualization, but we'll start simple, and just create a `<p>` paragraph element.
- We start with the `00_empty_page_template.html` and remove the comment around the script tags with:

  ```
  d3.select("body").append("p").text("My New Paragraph!");
  ```
- Save and refresh! There is text in the formerly empty browser window, and a new `<p>My New Paragraph</p>` element is visible in the web inspector.
- Element `<p>…</p>` was added by D3.

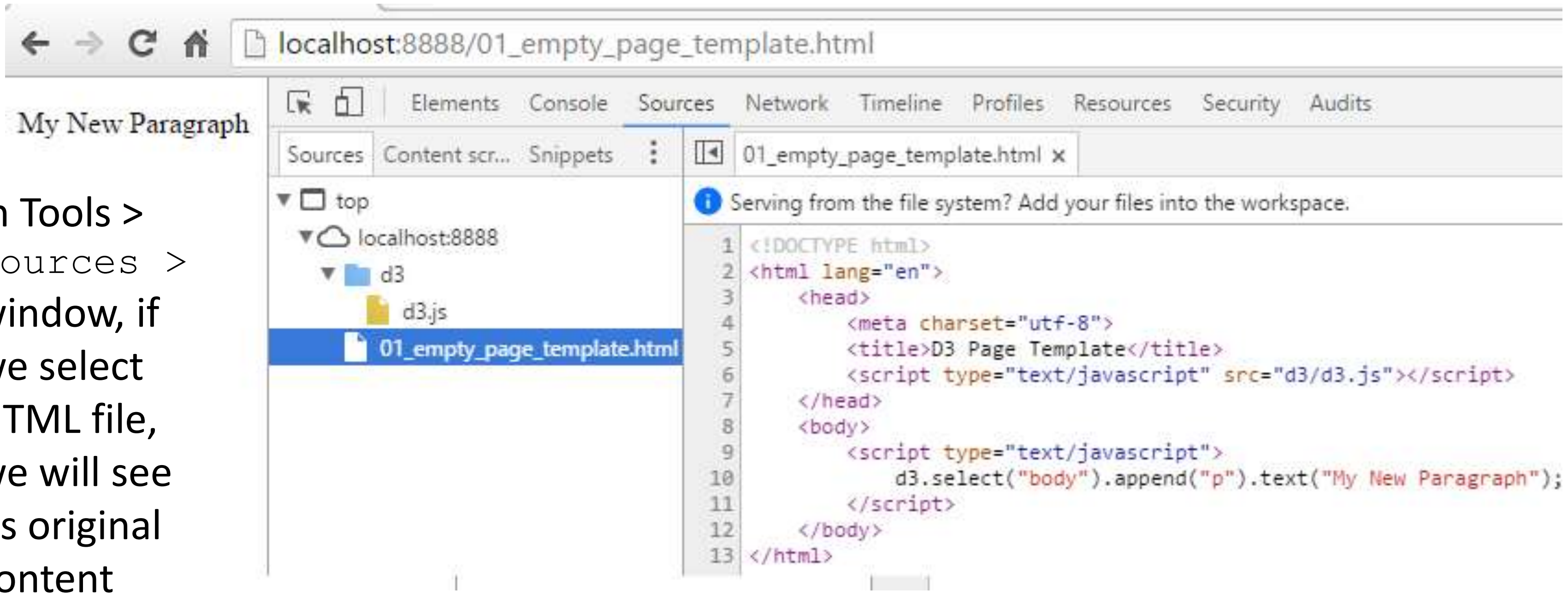


```
<!DOCTYPE html>
<html lang="en">
▼<head>
    <meta charset="utf-8">
    <title>D3 Page Template</title>
    <script type="text/javascript" src="d3/d3.js"></script>
  </head>
▼<body>
    <script type="text/javascript">
              d3.select("body").append("p").text("My New Paragraph");
          </script>
    <p>My New Paragraph</p>
  </body>
</html>
```

```
element.style {
}
body {                              user agent stylesh
  display: block;
  margin: ▶8px;
}
```

New DOM Element is added

- As seen in Elements view, there is a new `<p>` element that was generated on-the-fly! We will use a similar technique to dynamically generate tens or hundreds of elements, each one corresponding to a piece of a data set.
- In a sequence, we:
  - Invoked D3's select method, which selects a single element from the DOM using CSS selector syntax. (We selected the `body`.)
  - Created a new `p` element and appended it to the end of our selection, meaning just *before* the closing `</body>` tag in this case.
  - Set the text content of that new, empty paragraph to `"New paragraph!"`



Serving from the file system? Add your files into the workspace.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>D3 Page Template</title>
        <script type="text/javascript" src="d3/d3.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            d3.select("body").append("p").text("My New Paragraph");
        </script>
    </body>
</html>
```

In Tools > `Sources >` window, if we select HTML file, we will see its original content

Method Chaining

- D3 employs *method chaining syntax*, which you may be familiar from OO languages and jQuery. By "chaining" methods together with periods, you can perform several actions in a single line of code.
- *Functions* and *Methods* are synonyms for the same concept: a chunk of code that accepts an argument as input, performs some action, and returns some data as output.
- Our first line of D3 code read as:

```
d3.select("body").append("p").text("New paragraph!");
```

- JavaScript, like HTML, doesn’t care about whitespace and line breaks, so you can put each method on its own line for legibility:

```
d3.select("body")
    .append("p")
    .text("New paragraph!");
```

- Everyone has their own coding style. Use indents, line breaks, and whitespace (tabs or spaces) where ever they work for you.

One Call at a Time

- d3 — References the D3 object, so we can access its methods.
- `.select("body")` — Give `select()` a CSS selector as input, and it will return a reference to the first element in the DOM that matches. (Use `selectAll()` when you need more than one element.) We want the `body`, so a reference to `body` is handed off to the next method in the chain.
- `.append("p")` — `append()` creates whatever new DOM element you specify and appends it to the end (but *just inside*) of whatever selection it is acting on. In our case, we want to create a new `<p>` within the body. We specified `"p"` as the input argument, but this method also sees the reference to body that was passed down the chain from the `select()` method. Finally, `append()`, in turn, hands down a reference to the new element it just created.
- `.text("New paragraph!")` — `text()` takes a string and inserts it between the opening and closing tags of the current selection. Since the previous method passed down a reference to new `p`, this code just inserts the new text between `<p>` and `</p>`. (If there were an existing content, it would be overwritten.)
- `;` — Semicolon indicates the end of this line of code.

Method Hand Off

- Many, but not all, D3 methods return a selection (or, really, reference to a selection), which enables this handy technique of method chaining.
- Typically, a method returns a reference to the element that it just acted upon, but not always.
- When chaining methods, order matters. The output type of one method has to match the input type expected by the next method in the chain.
- If adjacent inputs and outputs are mismatched, the hand-off will not work and errors will be generated.
- To find out what each function expects and returns, D3 API reference is your friend. Please go to:

  https://github.com/mbostock/d3/wiki/API-Reference
- It contains detailed information on each method, including whether or not it returns a selection.

Going Chainless

- Our sample code could be rewritten without chain syntax as:

```
var body = d3.select("body");
var p = body.append("p");
p.text("New paragraph!");
```

- There will be times you need to break the chain, such as when you are calling so many functions that it doesn't make sense to string them all together. Sometimes, you want to organize your code in a way that makes more sense to you or makes the code more readable.

Binding Data

- With D3, we *bind* our data input values to elements in the DOM.
- Binding is like "attaching" or associating data to specific elements, so that later you can reference those values to apply mapping rules.
- Without the binding step, we have a bunch of data-less, un-mappable DOM elements.
- In a bind process we use D3's `selection.data()` method to bind data to DOM elements.
- There are two things we need in place first, before we can bind data:
  - The data
  - A selection of DOM elements

Data

- D3 is smart about handling different kinds of data, so it will accept practically any array of numbers, strings, or objects (themselves containing other arrays or key/value pairs).
- D3 can handle JSON (and GeoJSON) gracefully, and even has a built-in method to help you load in CSV files.
- We will start with a sample data set:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

- First, you need to decide what to select. That is, what elements will your data be associated with?
- We will keep it simple and make a new paragraph for each value in the data set. Something like this should be helpful

```
d3.select("body").selectAll("p")
```

- The paragraphs we want to select *don't exist yet*. How can we select elements that don't yet exist?
- The answer lies with `enter(),` a truly magical method.
- Our final code for this example, reads

```
d3.select("body").selectAll("p")
     .data(dataset)
     .enter()
     .append("p")
     .text("New paragraph!");
```

04_data_values_00.html

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="utf-8">
        <title>D3: Referencing values after the data join</title>
         <script type="text/javascript" src="d3.v3.js"></script>
   </head>
       <body>
         <script type="text/javascript">
             var dataset = [ 5, 10, 15, 20, 25 ];
             d3.select("body").selectAll("p")
                              .data(dataset)
                              .enter()
                              .append("p")
                              .text("New paragraph" );
                        //.text(function(d) { return d; });
        </script>
      </body>
</html>
```

- Browser shows:

```
New Paragraph
New Paragraph
New Paragraph
New Paragraph
New Paragraph
```

Results

- Here is what is happening.

`d3.select("body")` — Finds the body in the DOM and hands a reference off to the next step in the chain.

`.selectAll("p")` — Selects all paragraphs in the DOM. Since none exist yet, this returns an empty selection. This empty selection is representing the paragraphs that *will soon exist*.

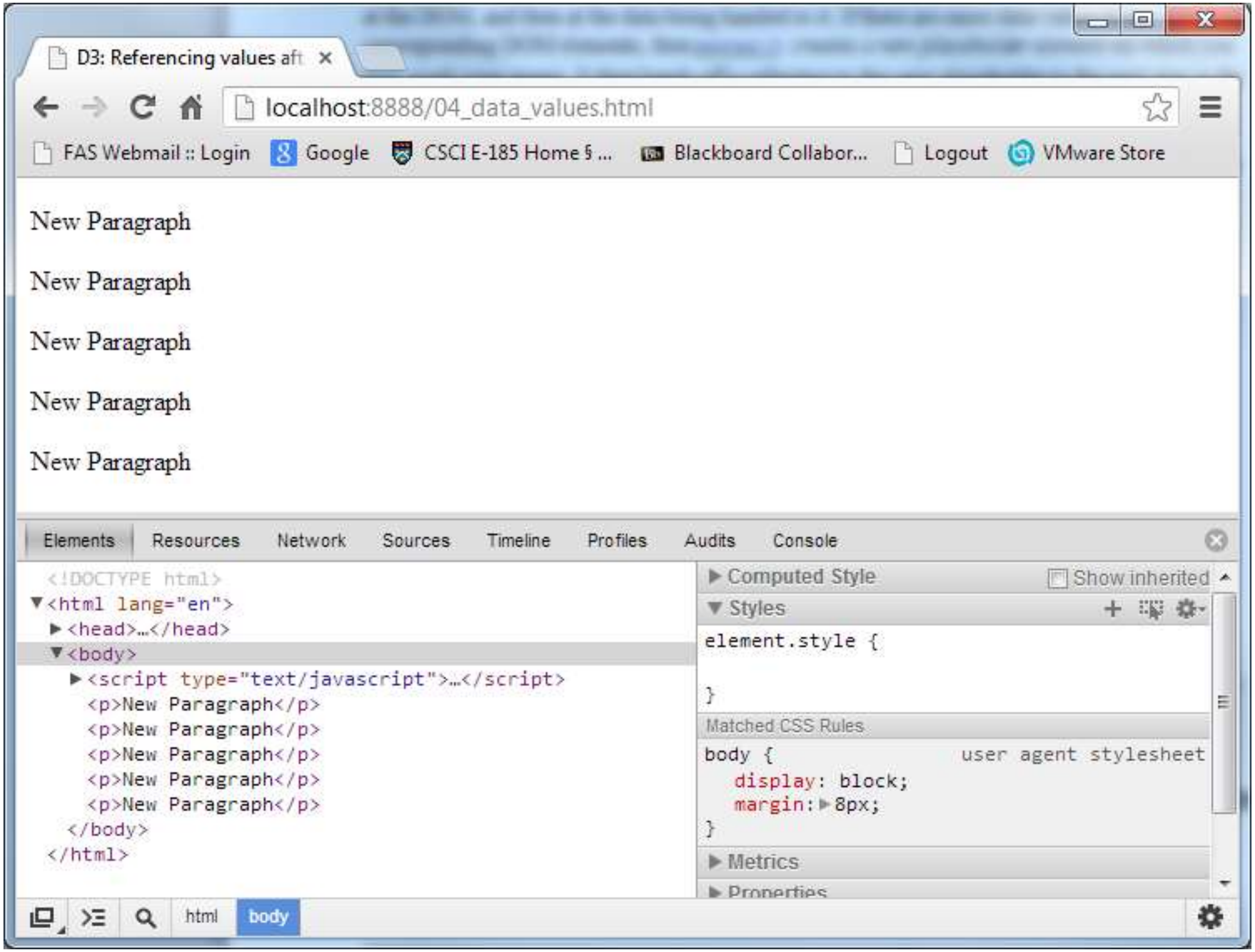
`.data(dataset)` — Counts and parses our data values. There are five values in our data set, so everything past this point is executed five times, once for each value.

`.enter()` — To create new, data-bound elements, you must use `enter()`. This method looks at the DOM, and then at the data being handed to it. If there are more data values than corresponding DOM elements, then `enter()` *creates a new placeholder element* . It then hands off a reference to this new placeholder to the next step in the chain.

`.append("p")` — Takes the placeholder selection created by `enter()` and inserts a `p` element into the DOM for every element in the dataset. Subsequently, it hands off a reference to the element it just created to the next step in the chain.

`.text("New paragraph!")` — Takes the reference to the newly created `p` and inserts a text value.

Where is the Data?



New Paragraph

New Paragraph

New Paragraph

New Paragraph

New Paragraph

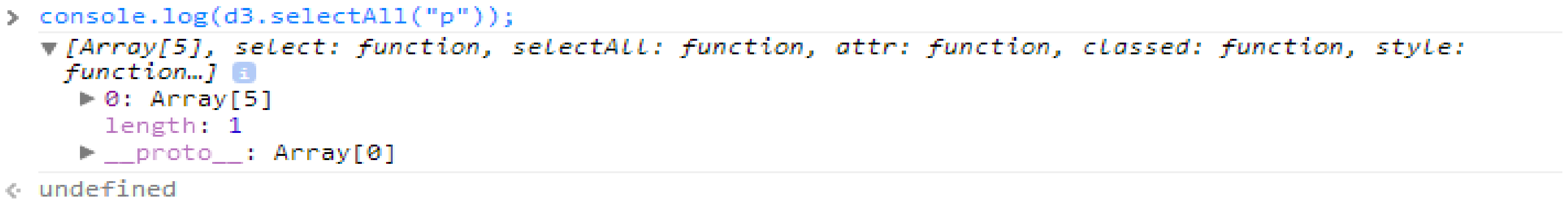
```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
     <p>New Paragraph</p>
     <p>New Paragraph</p>
     <p>New Paragraph</p>
     <p>New Paragraph</p>
     <p>New Paragraph</p>
   </body>
 </html>
```

```
element.style {

}
body {                    user agent stylesheet
   display: block;
   margin:▶8px;
}
```

JavaScript Console

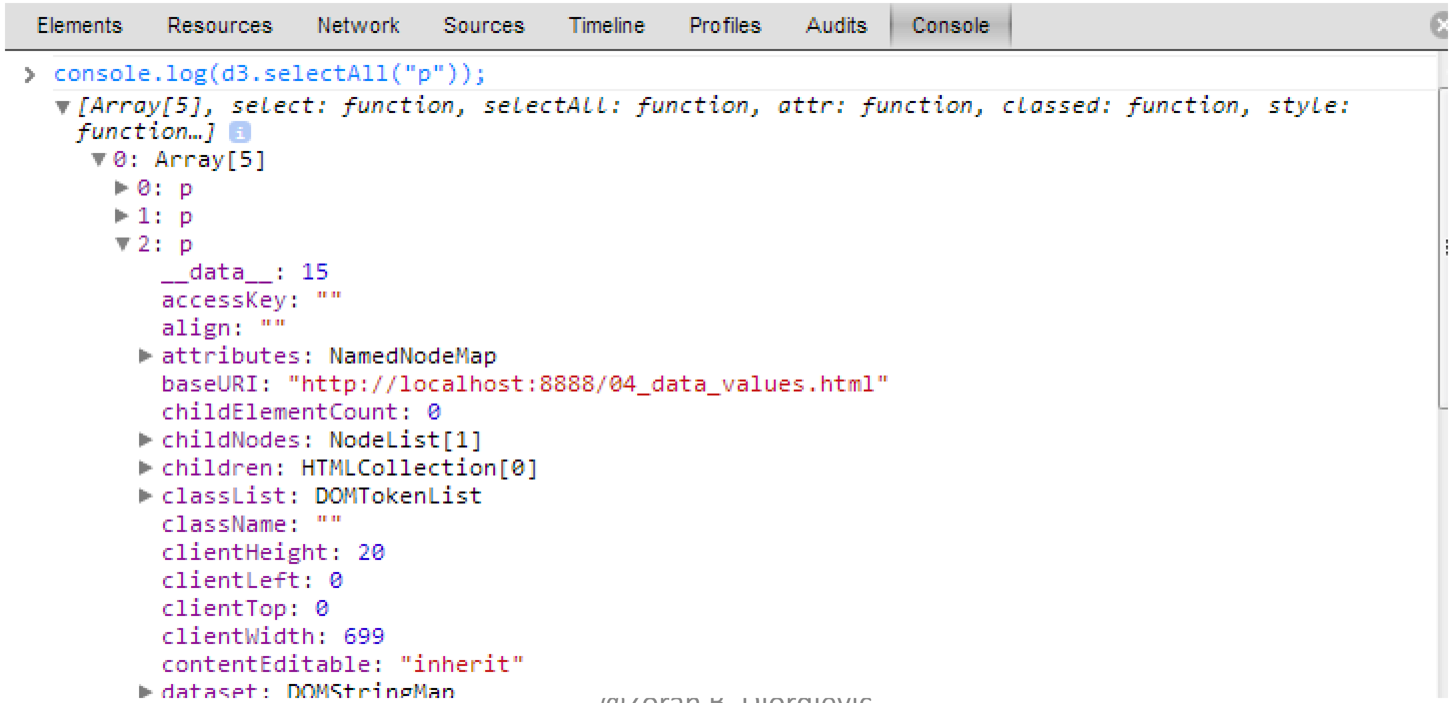
- Click on *Console*, type in the following JavaScript/D3 code, and hit enter:

`console.log(d3.selectAll("p"))` Operation produces an `Array[5]`



```
> console.log(d3.selectAll("p"));
▼ [Array[5], select: function, selectAll: function, attr: function, classed: function, style:
  function…]
  ▶ 0: Array[5]
    length: 1
  ▶ __proto__: Array[0]
< undefined
```

- If we click on `Array[5]` we see 5 elements. If we expand one of them, we do see data. Data are contained in `_data_:15` element.



```
> console.log(d3.selectAll("p"));
▼ [Array[5], select: function, selectAll: function, attr: function, classed: function, style:
  function…]
  ▼ 0: Array[5]
    ▶ 0: p
    ▶ 1: p
    ▼ 2: p
        __data__: 15
        accessKey: ""
        align: ""
      ▶ attributes: NamedNodeMap
        baseURI: "http://localhost:8888/04_data_values.html"
        childElementCount: 0
      ▶ childNodes: NodeList[1]
      ▶ children: HTMLCollection[0]
      ▶ classList: DOMTokenList
        className: ""
        clientHeight: 20
        clientLeft: 0
        clientTop: 0
        clientWidth: 699
        contentEditable: "inherit"
      ▶ dataset: DOMStringMap
```

Data

- Our third data value, the number 15, is showing up under the first paragraph's`__data__` attribute.
- Click into the other paragraph elements, and you'll see they also contain `__data__` values: `10, 15, 20`, and `25`, just as we specified.
- When D3 binds data to an element, that data doesn't exist in the DOM, but it does exist in memory as a `__data__` attribute of that element.
- JavaScript console is where you can go to confirm whether or not your data was bound as expected.

Using Data

- To use our dataset we change the code. We replace the line

```
.text("New paragraph!")
```

- with

```
.text(function(d) { return d; });    << anonymous function
```

- Browser output is changed to display the dataset:

```
5
10
15
20
25
```

- Data is used to populate the contents of each paragraph, all thanks to the magic of the `data()` method. When chaining methods together, after you call `data()`, you always create an anonymous function that accepts reference `d` as input. The `data()` ensures that `d` is set to the corresponding value in the original data set, given the current element at hand.
- The value of "the current element" changes over time as D3 loops through each element. For example, when looping through the third time, our code creates the third `p` tag, and `d` will correspond to the third value in data set (`dataset[2]`). So the third paragraph gets text content of "`15`".

User Defined Functions

- The basic structure of a user defined function is:

```
function(input_value) {
    //Calculate something here
    return output_value;
}
```

- The function we used above was very simple

```
function(d) {
    return d;
}
```

- and it is wrapped within D3's `text()` function, so whatever our function returns is handed off to `text()` .

```
.text(function(d) {
    return d;
});
```

- We can customize these functions however we want. For example, we could add some extra text, which produces more elaborate browser display, like:

```
.text(function(d) {
    return "I can count up to " + d;
});
```

Data Needs to be "held"

- Why do we have to write `function(d)..` rather than just pass variable `d` on its own.
- For example, this will not work:

```
.text("I can count up to " + d);
```

- Without wrapping `d` in an anonymous function, `d` has no value. Apparently, `d` needs to be held (or handed out) by a function:

```
.text(function(d) {  //
return "I can count up to " + d;
});
```

- The reason for this syntax is that `.text(), attr(),` and many other D3 methods take a function as an argument. For example, `text()` can take either a simple static string of text as an argument:

```
.text("someString")
```

- …*or* the result of a function:

```
.text(someFunction())
```

- …*or* an anonymous function itself can be the argument, such as when you write:

```
.text(function(d) {
    return d;
})
```

- Above, we are defining an anonymous function. If D3 sees a function there, it will *call* that function, while handing off the current datum `d` as the function's argument. Without the function in place, D3 can't know to whom it should hand off the argument `d`.

D3 `attr()` and `style()` methods

- `attr()` and `style()`, are D3 methods which allow us to set HTML attributes and CSS properties on selections, respectively.
- For example, if we add one more line to our recent code

```
.style("color", "red");
```

- All the text will turn red.
- We could be more specific and use a custom function to make the text red only if the current data exceeds a certain threshold. So we could revise that last line to use a function:

```
.style("color", function(d) {
    if (d > 15) {   //Threshold of 15
        return "red";
    } else {
        return "black";
    }
});
```

- The first three lines are black, but once `d` exceeds `15`, the text turns red.
- In what follows next we will use `attr()` and `style()` to manipulate `div`-s, generating a bar chart!

Method `attr()`

- Method `attr()` is attached to a selection (result of action of select("body"), for example.
- Method is invoked with argument `name` and optionally argument `value`: `selection.attr(name[, value])`
- If *`value`* is specified, sets the attribute with the specified name to the specified value on all selected elements. If *`value`* is a constant, then all elements are given the same attribute value; otherwise, if *`value`* is a function, then the function is evaluated for each selected element (in order), being passed the current data `d` and the current index `i`, with the this context as the current DOM element. The function's return value is then used to set each element's attribute. A `null` value will remove the specified attribute.
- If *`value`* is not specified, `attr(name)` returns the value of the specified attribute for the first non-null element in the selection. This is generally useful only if you know that the selection contains exactly one element.
- The specified *`name`* may have a namespace prefix, such as `xlink:href`, to specify an "`href`" attribute in the XLink namespace. By default, D3 supports `svg, xhtml, xlink, xml`, and `xmlns` namespaces.

HTML Tag `<div>`

- The `<div>` tag defines a division or a section in an HTML document.
- The `<div>` tag is used to group block-elements to format them with CSS.
- The `<div>` element is very often used together with CSS, to layout a web page. By default, browsers always place a line break before and after the <div> element. However, this can be changed with CSS.
- <div> could use inline style

```
<div style="color:#0000FF">
  <h3>This is a heading</h3>
  <p>This is a paragraph.</p>
</div>
```

- Or, `<div>` could use classes defined in the style sheet

```
<style>
.center
{  margin:auto;  width:70%;  background-color:#b0e0e6; }
</style>
. . . .
<div class="center">
         <p>Cats play with mice.</p>
</div>
```

Drawing with Data

- Bar charts are essentially just rectangles, and an HTML `<div>` is the easiest way to draw a rectangle. This `div` could work well as a data bar:

```
<div style="display: inline-block;
            width: 20px;
            height: 75px;
            background-color: teal;"></div>
```

- The width and height of the above `<div>` element are set with a CSS style.
- Each bar in our chart will share the same display properties (except for height), so we could define those shared styles in a CSS class called `bar`:

```
div.bar {
    display: inline-block;
    width: 20px;
    height: 75px;   /* We'll override this later */
    background-color: teal;
}
```

- Each `div` needs to be assigned the `bar` class, so that new CSS rules apply.
- If we were writing the HTML code by hand, we would write:

```
<div class="bar"></div>
```

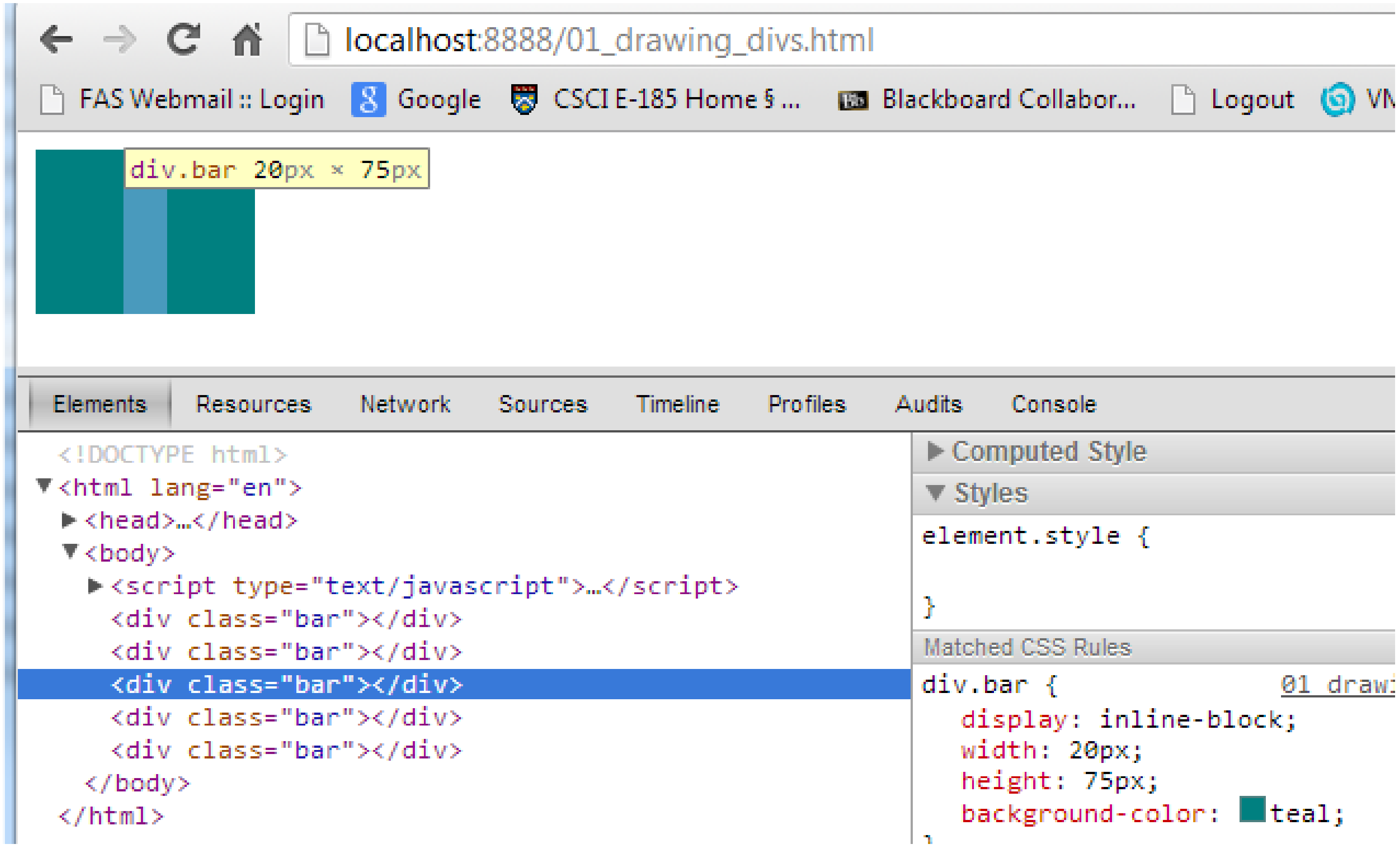
- In D3, to add a class to an element, we use the `selection.attr()` method.
- It's important to understand the difference between `attr()` and its close cousin, `style()`.

01_drawing_divs.html

```
<!DOCTYPE html>
<html lang="en">
         <head>
            <meta charset="utf-8">
            <title>D3: Drawing divs with data</title>
            <script type="text/javascript" src="../d3/d3.js"></script>
               <style type="text/css">
                              div.bar {
                                       display: inline-block;
                                       width: 20px;
                                       height: 75px;
                                       background-color: teal;
                              }
              </style>
         </head>
         <body>
                  <script type="text/javascript">
                              var dataset = [ 5, 10, 15, 20, 25 ];
                              d3.select("body").selectAll("div")
                                       .data(dataset)
                                       .enter()
                                       .append("div")
                                       .attr("class", "bar");
                  </script>
         </body>
</html>
```

Browser View

- Though we see a single rectangle, with the help of developer tools we could see that we do have 5 of them, one after another, all of the same height.



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
      <div class="bar"></div>
      <div class="bar"></div>
      <div class="bar"></div>
      <div class="bar"></div>
      <div class="bar"></div>
    </body>
</html>
```

```
element.style {

}
```

```
div.bar {
    display: inline-block;
    width: 20px;
    height: 75px;
    background-color: teal;
}
```

Setting Attributes

- D3 method `attr()` is used to set an HTML attribute and its value on an element.
- An HTML attribute is any property/value pair that you could include between an element's <>brackets. For example, these HTML elements

```
<p class="caption">
<select id="country">
<img src="logo.png" width="100px" alt="Logo" />
```

- contain a total of five attributes (and corresponding values), all of which could be set with attr():

```
class   |   "caption"
id      |   "country"
src     |   "logo.png"
width   |   "100px"
alt     |   "Logo"
```

- To give our `div`-s a class of bar, we used:

```
.attr("class", "bar")
```

`class`-es

- Note that an element's `class` is an HTML attribute.
- The `class` attribute, is used to reference a CSS style rule.
- This may cause some confusion because there is a difference between setting a `class` (from which styles are inferred) and applying a `style` directly to an element. We can do both with D3.
- Although you should use whatever approach makes the most sense to you, it is recommended to use `classes` for properties that are shared by multiple elements, and apply `style` rules directly only on isolated tags.
- There is also another D3 method, `classed(),` which can be used to quickly apply or remove classes from elements. Call to attr()

```
.attr("class", "bar")
```

- could be rewritten as:

```
.classed("bar", true)
```

- and it would have applied class `bar` to all `div-s`

Setting Styles

- The `style()` method is used to apply a CSS property and value directly to an HTML element. This is the equivalent of including CSS rules within a style attribute right in your HTML, as in: `<div style="height: 75px;"></div>`
- In a bar chart, the height of each bar must be a function of the corresponding data value. We can add this to the end of our D3 code:

```
.style("height", function(d) {
    var barHeight = d * 5; //Scale up by 5 to make plot prettier
    return barHeight + "px";
});
```

02_drawing_divs_height.html

```
<!DOCTYPE html>
<html lang="en">
          <head>
             <meta charset="utf-8"> <title>D3: Setting div heights from data</title>
             <script type="text/javascript" src="d3/d3.js"></script>
             <style type="text/css">
                    div.bar {
                              display: inline-block; width: 20px;
                              height: 75px;        /* Gets over-riden below */
                              background-color: teal;
                    }
             </style>
          </head>
          <body>
                    <script type="text/javascript">
                              var dataset = [ 5, 10, 15, 20, 25 ];
                              d3.select("body").selectAll("div")
                                        .data(dataset)
                                        .enter()
                                        .append("div")
                                        .attr("class", "bar")
                                        .style("height", function(d) {
                                                  return 5*d + "px";  });

                    </script>
          </body>
</html>
```

Setting Styles

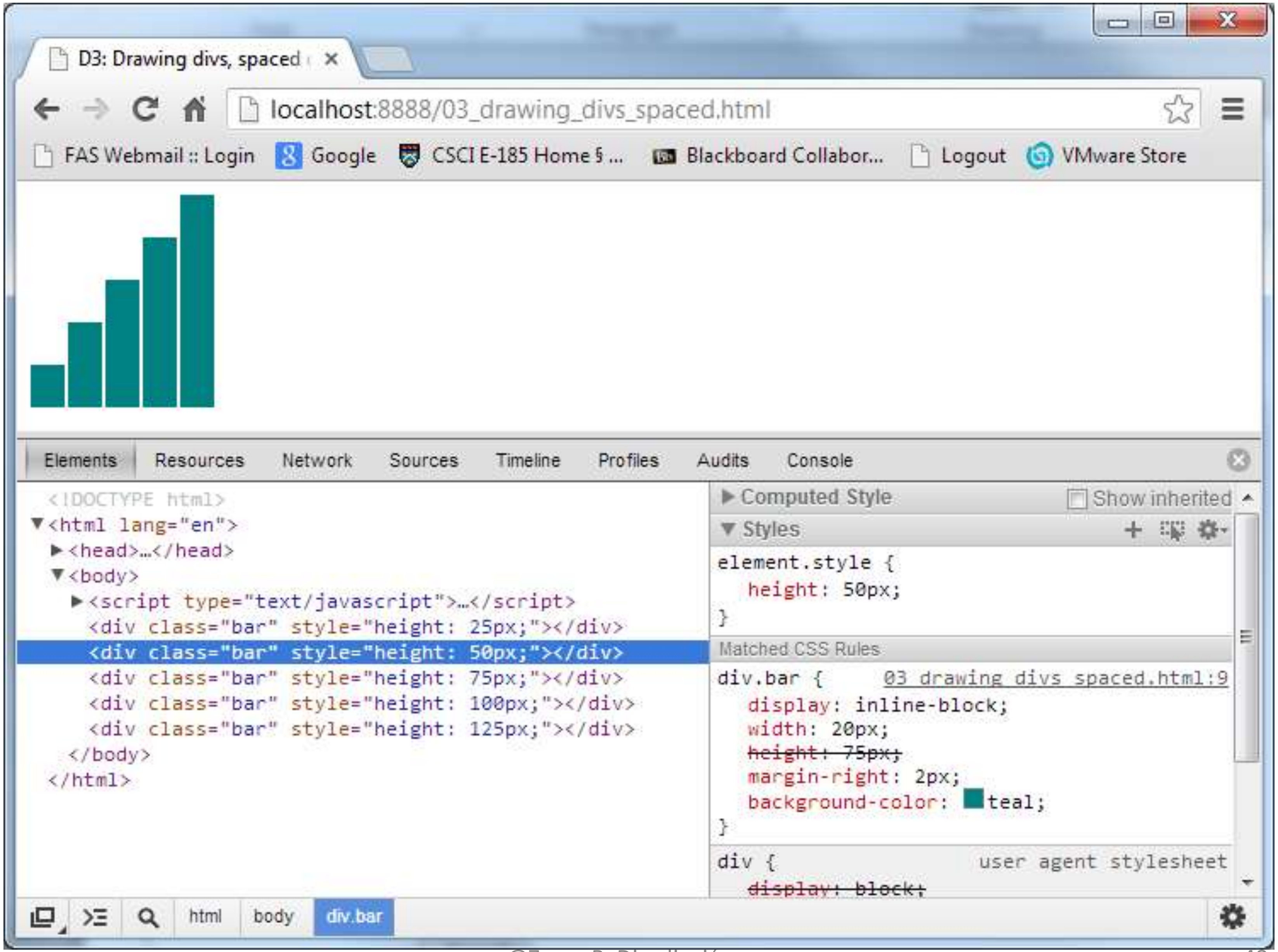
- To add some space to the right of each bar, to space things out, we add property `margin-right: 2px;` to the `div.bar` class

```
<style type="text/css">
        div.bar {
                display: inline-block;
                width: 20px;
                height: 75px;   /* Gets over-riden by file below */
                margin-right: 2px;
                background-color: teal;
                }
</style>
```

03_drawing_divs_spaced.html

```
<!DOCTYPE html>
<html lang="en">
   <head>
          <meta charset="utf-8">
          <title>D3: Drawing divs, spaced out</title>
          <script type="text/javascript" src="d3/d3.v3.js"></script>
          <style type="text/css">
                    div.bar {
                      display: inline-block;  width: 20px;
                      height: 75px;     /* Gets overriden by D3-assigned height  */
                      margin-right: 2px;   background-color: teal;
                    }
          </style>
  </head>
  <body>
       <script type="text/javascript">
             var dataset = [ 5, 10, 15, 20, 25 ];
                  d3.select("body").selectAll("div").data(dataset)
                                   .enter().append("div").attr("class", "bar")
                                   .style("height", function(d) {
                                       var barHeight = d * 5;
                                       return barHeight + "px";
                                       });
                    </script>
          </body>
</html>
```

Spaced Bar Graph



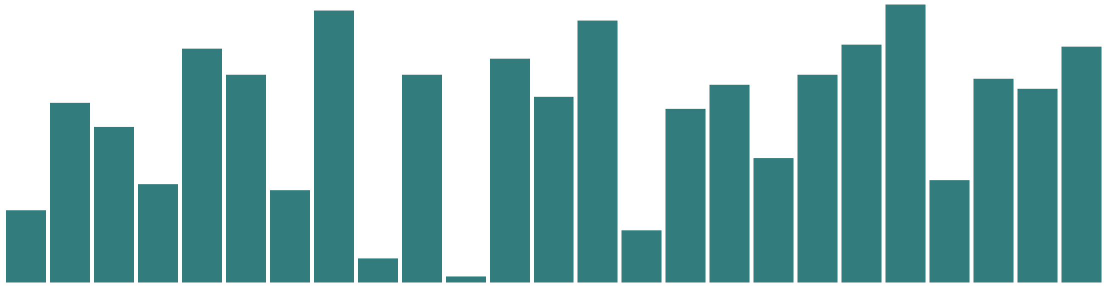
```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
     <div class="bar" style="height: 25px;"></div>
     <div class="bar" style="height: 50px;"></div>
     <div class="bar" style="height: 75px;"></div>
     <div class="bar" style="height: 100px;"></div>
     <div class="bar" style="height: 125px;"></div>
   </body>
</html>
```

```
element.style {
   height: 50px;
}
```

```
div.bar {       03 drawing divs spaced.html:9
   display: inline-block;
   width: 20px;
   ~~height: 75px;~~
   margin-right: 2px;
   background-color: 🟩teal;
}
div {             user agent stylesheet
   ~~display: block;~~
```

Random Data

- Sometimes, for testing purposes, we need to generate random data values,



```
var dataset = [];                            //Initialize empty array
for (var i = 0; i < 25; i++) {               //Loop 25 times
    var newNumber = Math.random() * 30;      //New random number (0-30)
    dataset.push(newNumber);                 //Add new number to array
}
```

- This code:
  - Creates an empty array called dataset.
  - Initiates a for loop, which is executed 25 times.
  - Each time, it generates a new random number with a value between zero and 30.
  - That new number is appended to the dataset array. (push() is an array method that appends a new value to the end of an array.)
- If we open up the JavaScript console and enterconsole.log(dataset). You should see the full array of 25 randomized data values.

```
> console.log(dataset);
```

SVG Primer

- D3 is most useful when used to generate and manipulate graphs with SVGs.
- Drawing with `<div>`-s and other native HTML elements is possible, but a bit clunky and subject to the usual inconsistencies across different browsers.
- Using SVG is more reliable, visually consistent, and faster.
- Scalable Vector Graphics is a text-based image format. Each SVG image is defined using markup code similar to HTML.
- SVG code can be included directly within any HTML document. Every web browser supports SVG *except* Internet Explorer versions 8 and older.
- SVG is XML-based. Elements that don't have a closing tag must self-close. For example:
- `<element></element>   <!-- Uses closing tag -->`
- `<element/>            <!-- Self-closing tag -->`
- Before you can draw anything, you must create an SVG element.
- An SVG element is a canvas on which your visuals are rendered. At a minimum, it's good to specify width and height values. If you don't specify these, the SVG will take up as much room as it can within its enclosing element.

```
<svg width="500" height="50">
</svg>
```

- SVG generated by that static code is an empty rectangle.

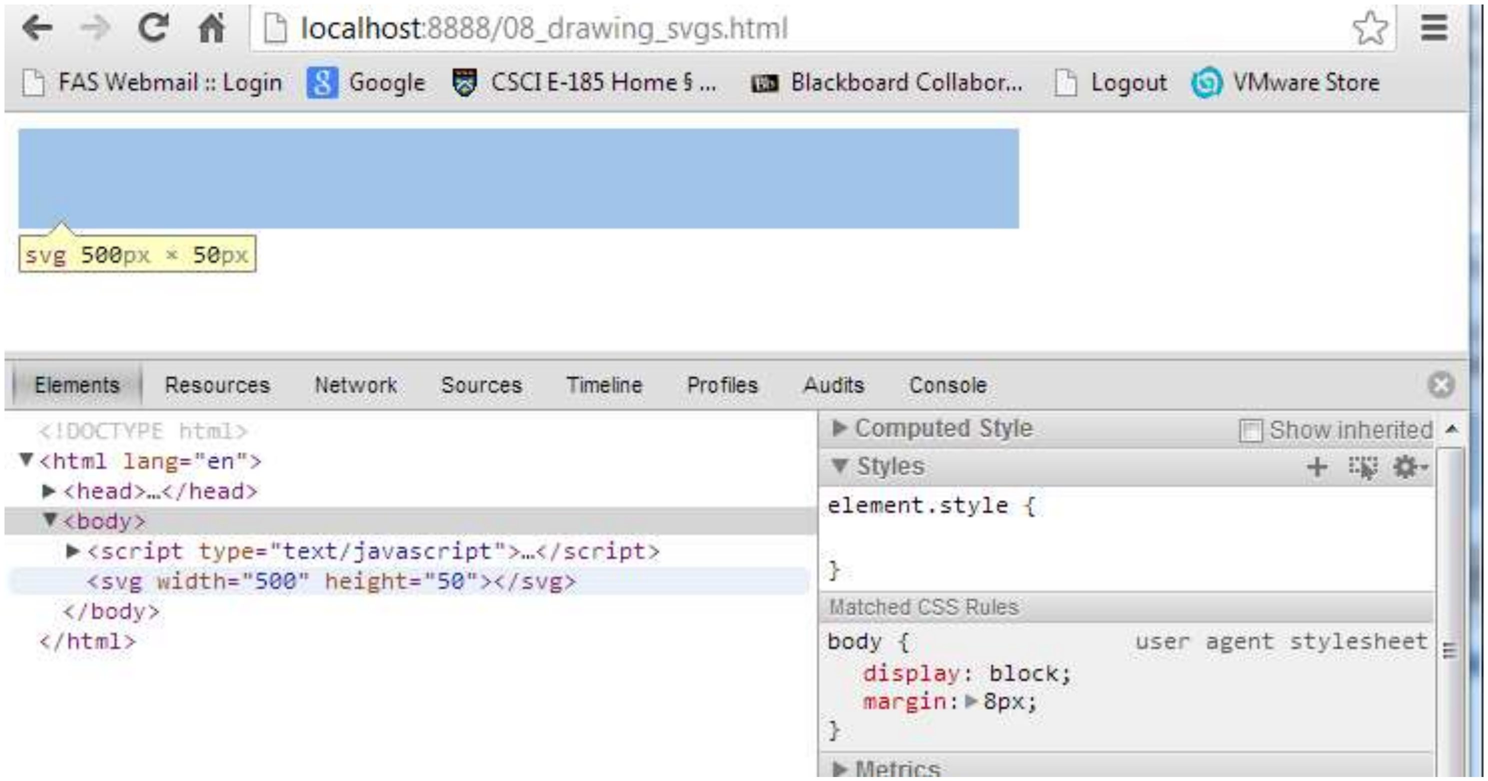
SVG generated by D3, `08_drawing_svgs.html`

- We can inject previous <svg> element into an HTML page using D3:

```
<!DOCTYPE html>
<html lang="en">
        <head>
                <meta charset="utf-8">
                <title>D3: Creating an empty SVG</title>
                <script type="text/javascript"
                        src="d3/d3.js"></script>
                <style type="text/css">
                        /* No style rules here yet */
                </style>
        </head>
        <body>
                <script type="text/javascript">
                        //Create SVG element
                        var svg = d3.select("body")
                                .append("svg")
                                .attr("width", 500)
                                .attr("height", 50);
                </script>
        </body>
</html>
```

First SVG

- Since our SVG element has no colors, we need the Developer's tool to see it.
- The tool shows that we injected an <svg> element into the DOM
- If you scroll over it in the Developer's tool, the SVG will pop-up.



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
     <svg width="500" height="50"></svg>
   </body>
</html>
```

```
element.style {

}
body {                    user agent stylesheet
   display: block;
   margin: ▶8px;
}
```

Simple Shapes

- There are a number of visual elements that you can include between those `svg` tags, including `rect`, `circle`, `ellipse`, `line`, `text`, and `path`.
- SVG uses the usual pixel-based coordinates system in which 0,0 is the top-left corner of the drawing space. Increasing x values move to the right, while increasing y values move down.
- `rect` draws a rectangle. Use `x` and `y` to specify the coordinates of the upper-left corner, and width and height to specify the dimensions. This rectangle fills the entire space of our SVG:

```
<rect x="0" y="0" width="500" height="50"/>
```

- `circle` draws a circle. Use `cx` and `cy` to specify the coordinates of the *center*, and `r` to specify the radius. This circle is centered in the middle of our 500-pixel-wide SVG because its cx ("center-x") value is 250.

```
<circle cx="250" cy="25" r="25"/>
```

- `ellipse` is similar, but expects separate values for spread along each axis. Instead of r, use `rx` and `ry`.

```
<ellipse cx="250" cy="25" rx="100" ry="25"/>
```

- `line` draws a line. Use `x1` and `y1` to specify the coordinates of one end of the line, and `x2` and `y2` to specify the coordinates of the other end.
- A stroke color must be specified for the line to be visible.

```
<line x1="0" y1="0" x2="500" y2="50" stroke="black"/>
```

Simple Shapes

- `text` renders text. Use x to specify the position of the left edge, and y to specify the vertical position of the type’s *baseline*.
- `<text x="250" y="25">Hello there</text>`
- `text` will inherit the CSS-specified font styles of its parent element unless specified otherwise. We could override that formatting as follows:

```
<text x="250" y="25" font-family="sans-serif"  font-size="25"
fill="gray">Hello there</text>
```

- When any visual element runs up against the edge of the SVG, it will be clipped.
- We have to be careful when using text so our descenders don’t get cut off You can see this happen when we set the baseline (y) to 50, the same as the height of our SVG:

-
  ```
  <text x="250" y="50" font-family="sans-serif"  font-size="25"
  fill="gray">Easy-peasy</text>
  ```

Drawing with SVG

- You have noticed that all properties of SVG elements are specified as *attributes*. That is, they are included as property/value pairs within each element tag, like this:

```
<element property="value"/>
```

- That looks exactly like HTML!

```
<p class="eureka">
```

- We know how to use D3's `append()` and `attr()` methods to create new HTML elements and set their attributes.
- Since SVG elements exist in the DOM, just as HTML elements do, we can use `append()` and `attr()` in exactly the same way to generate SVG images!

Creating the SVG

- First, we need to create the SVG element in which to place all our shapes.

```
d3.select("body").append("svg");
```

- That will find the body and append a new `svg` element just before the closing `</body>` tag.
- It is recommend to store returned reference of `append()` call:

```
var svg = d3.select("body").append("svg");
```

- Most D3 methods return a reference to the DOM element on which they act?
- Think of `svg` not as a "variable" but as a "reference pointing to the SVG object that we just created." This reference will save a lot of coding. For example, instead of having to search for that SVG each time — as in `d3.select("svg")` — we just reference `svg`.

```
svg.attr("width", 500) .attr("height", 50);
```

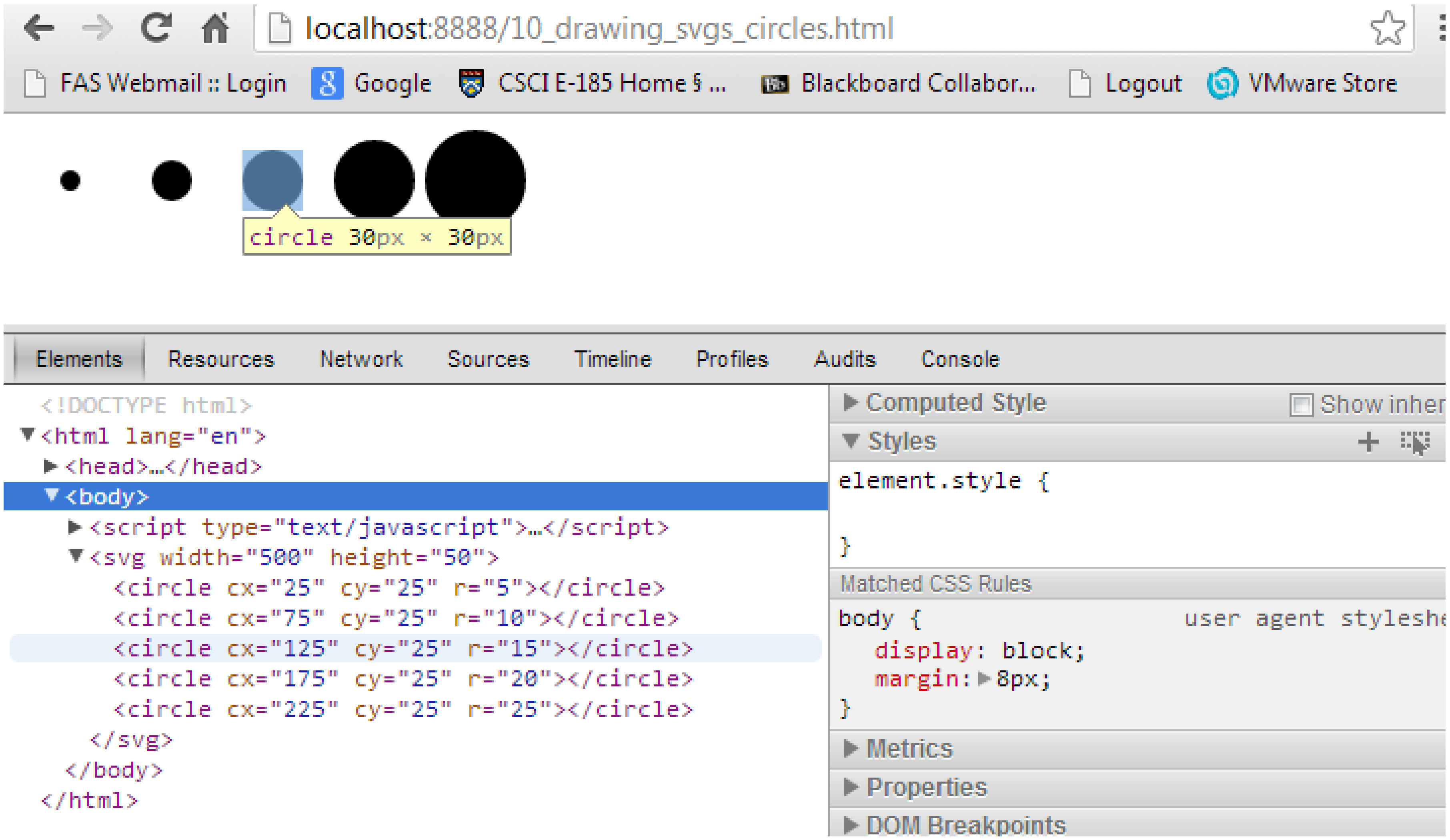
- Alternatively, above two lines could be written as one line of code:

```
var svg = d3.select("body") .append("svg") .attr("width", 500)
.attr("height", 50);
```

10_drawing_circles.html

```
<!DOCTYPE html>
<html lang="en">
         <head>
             <meta charset="utf-8">
             <title>D3: Drawing SVG circles with data</title>
             <script type="text/javascript" src="../d3/d3.js"></script>
             <style type="text/css"> /* No style rules here yet */ </style>
         </head>
         <body>
             <script type="text/javascript">
                  var w = 500; //Width and height
                  var h = 50;
                  var dataset = [ 5, 10, 15, 20, 25 ];   //Data
                  var svg = d3.select("body")            //Create SVG element
                     .append("svg").attr("width", 500).attr("height", 50);
                  var circles = svg.selectAll("circle") # circle is an SVG tag
                               .data(dataset).enter().append("circle");
                  circles.attr("cx", function(d, i) {
                             return (i * 50) + 25;
                                 })
                             .attr("cy", h/2).attr("r", function(d) {
                                        return d;
                                       });
                  </script>
         </body>
</html>
```

10_drawing_circles.html



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
    ▼<svg width="500" height="50">
        <circle cx="25" cy="25" r="5"></circle>
        <circle cx="75" cy="25" r="10"></circle>
        <circle cx="125" cy="25" r="15"></circle>
        <circle cx="175" cy="25" r="20"></circle>
        <circle cx="225" cy="25" r="25"></circle>
     </svg>
   </body>
</html>
```

```
element.style {

}
```

```
body {                     user agent styleshe
   display: block;
   margin:▶8px;
}
```

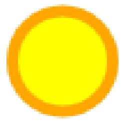
Styling SVG Elements

- SVG’s default style is a black fill with no stroke. If you want anything else, you’ll have to apply styles to your elements. Common SVG properties are:
- `fill` — A color value. Just as with CSS, colors can be specified as
  - named colors — `orange`
  - hex values — `#3388aa or #38a`
  - RGB values — `rgb(10, 150, 20)`
  - RGB with alpha transparency — `rgba(10, 150, 20, 0.5)`
- `stroke` — A color value.
- `stroke-width` — A numeric measurement (typically in pixels).
- `opacity` — A numeric value between 0.0 (completely transparent) and 1.0 (completely opaque).
- With text, you can also use those properties, which work just like in CSS:
- `font-family`
- `font-size`

Applying Style to SVG

- There are two ways to apply styles to an SVG element: either directly (inline) as an attribute of the element, or with a CSS style rule.
- These style properties are applied directly to a circle as attributes:

```
<circle cx="25" cy="25" r="22" fill="yellow" stroke="orange"
stroke-width="5"/>
```



- Alternatively, we could strip the style attributes, assign to the circle a class `pumpkin` (just as if it were a normal HTML element)

```
<circle cx="25" cy="25" r="22" class="pumpkin"/>
```

- and then put the fill, stroke, and stroke-width rules into a CSS style that targets the new class pumpkin:

```
.pumpkin { fill: yellow; stroke: orange; stroke-width: 5; }
```

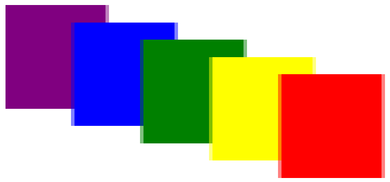
- The CSS approach has a few obvious benefits:
  - You can specify a style once and have it applied to multiple elements.
  - CSS code is generally easier to read than inline attributes.

Layering

- There are no "layers" in SVG, and no real concept of depth. SVG does not support CSS's z-index property, so shapes can only be arranged within the two-dimensional x/y plane.
- If we draw multiple shapes, they overlap:

```
<rect x="0" y="0" width="30" height="30" fill="purple"/>
<rect x="20" y="5" width="30" height="30" fill="blue"/>
<rect x="40" y="10" width="30" height="30" fill="green"/>
<rect x="60" y="15" width="30" height="30" fill="yellow"/>
<rect x="80" y="20" width="30" height="30" fill="red"/>
```

- The order in which elements are coded determines their depth order. The purple square appears first in the code, so it is rendered first. Then, the blue square is rendered "on top" of the purple one, then the green square on top of that, and so on.



Bar Chart in SVG

- First things first, we need to decide on the size of the new SVG:

```
//Width and height
var w = 500; var h = 100;
```

- Then, we tell D3 to create an empty SVG element and add it to the DOM:
- //Create SVG element

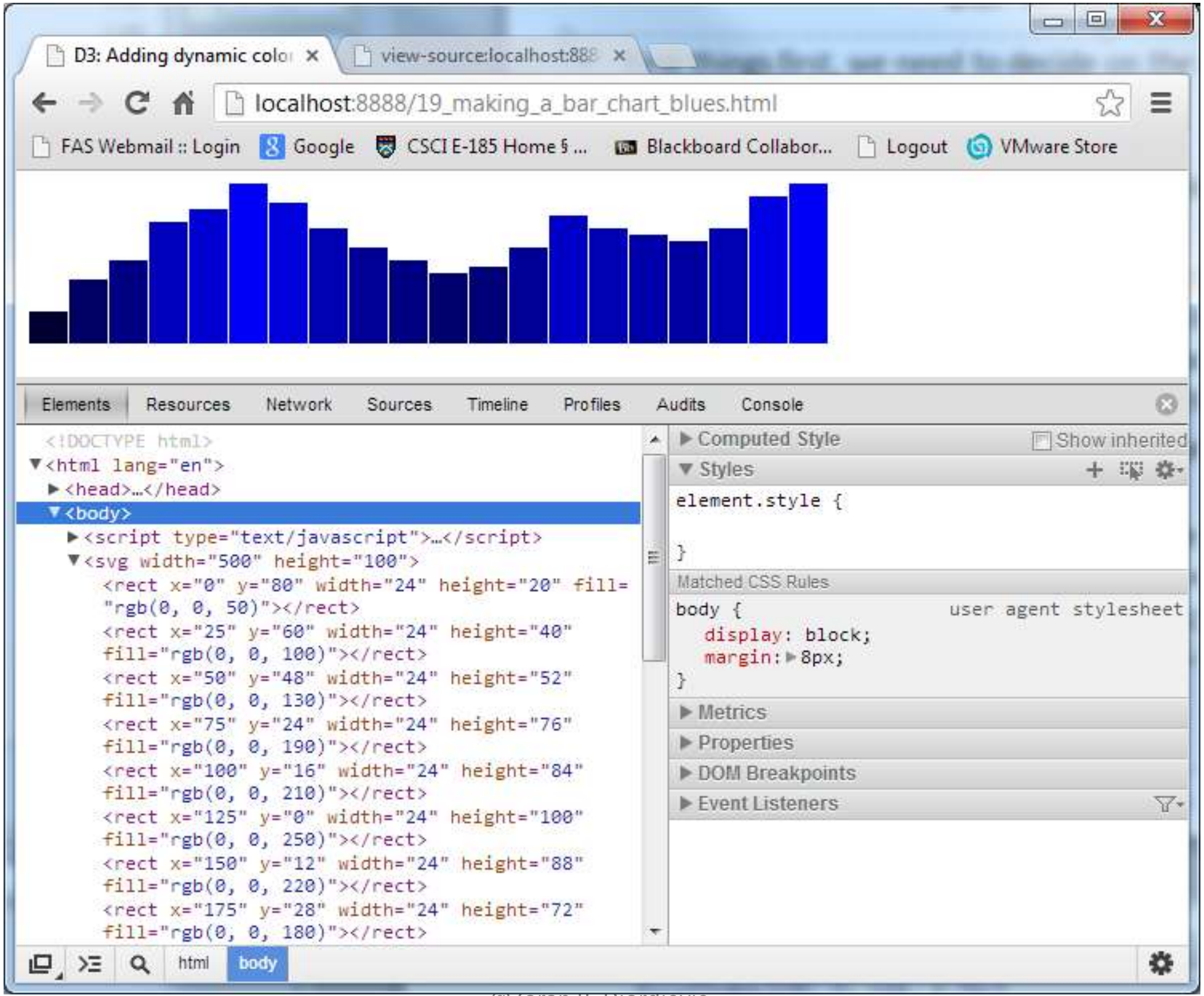
```
var svg = d3.select("body") .append("svg")
          .attr("width", w) .attr("height", h);
```

- This inserts a new `<svg>` element just before the closing `</body>` tag, and assigns the SVG a width and height of 500 by 100 pixels.
- This statement also puts the result into our new variable called `svg`, so we can easily reference the new SVG .
- Next, instead of creating divs, we generate `rects` and add them to `svg`.

```
svg.selectAll("rect") .data(dataset) .enter() .
     append("rect") .attr("x", 0) .attr("y", 0)
     .attr("width", 20) .attr("height", 100);
```

- And so on ….

SVG Bar Chart



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/javascript">…</script>
    ▼<svg width="500" height="100">
       <rect x="0" y="80" width="24" height="20" fill=
       "rgb(0, 0, 50)"></rect>
       <rect x="25" y="60" width="24" height="40"
       fill="rgb(0, 0, 100)"></rect>
       <rect x="50" y="48" width="24" height="52"
       fill="rgb(0, 0, 130)"></rect>
       <rect x="75" y="24" width="24" height="76"
       fill="rgb(0, 0, 190)"></rect>
       <rect x="100" y="16" width="24" height="84"
       fill="rgb(0, 0, 210)"></rect>
       <rect x="125" y="0" width="24" height="100"
       fill="rgb(0, 0, 250)"></rect>
       <rect x="150" y="12" width="24" height="88"
       fill="rgb(0, 0, 220)"></rect>
       <rect x="175" y="28" width="24" height="72"
       fill="rgb(0, 0, 180)"></rect>
```

```
element.style {

}
```

```
body {                    user agent stylesheet
   display: block;
   margin:▶8px;
}
```

19_making_a_bar_chart_blues.html

```
<!DOCTYPE html>
<html lang="en">
          <head>
                    <meta charset="utf-8">
                    <title>D3: Adding dynamic color, based on data</title>
                    <script type="text/javascript" src="d3/d3.v3.js"></script>
                    <style type="text/css">       </style>
          </head>
          <body>
                    <script type="text/javascript">
                    //Width and height
                    var w = 500; var h = 100;
                    var barPadding = 1;
                    var dataset = [ 5, 10, 13, 19, 21, 25, 22, 18, 15, 13,
                    11, 12, 15, 20, 18, 17, 16, 18, 23, 25 ];
                    var svg = d3.select("body").append("svg").attr("width", w)
                         .attr("height", h);
                         svg.selectAll("rect")
                             .data(dataset).enter().append("rect")
                             .attr("x", function(d, i){return i *(w/dataset.length);})
                             .attr("y", function(d) {      return h - (d * 4); })
                             .attr("width", w / dataset.length - barPadding)
                             .attr("height", function(d) { return d * 4;})
                             .attr("fill",function(d){return "rgb(0, 0, "+(d * 10)+ ")";});
                    </script>
          </body>
</html>
```

Scatterplot

- The scatterplot is a common type of visualization that represents two sets of corresponding values on two different axes: horizontal & vertical: `x` and `y`.
- You have a lot of flexibility around how you structure your data set.
- For the scatterplot, we could use an array of arrays.
- The primary array will contain one element for each data "point." Each of those "point" elements will be another array, with just two values: one for the x value, and one for y.

```
var dataset = [
          [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
          [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
        ];
```

- [] means array, so nested hard brackets [[]] indicate an array within another array.
- We separate array elements with commas, so an array containing three other arrays would look like: `[[],[],[]]`

Scatterplot

- We carry over most of the code from our bar chart experiments, including the piece that creates the SVG element:

```
//Create SVG element
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);
```

- Instead of creating `rects`, however, we'll make a circle for each data point:

```
svg.selectAll("circle")
   .data(dataset)
   .enter()
   .append("circle")
```

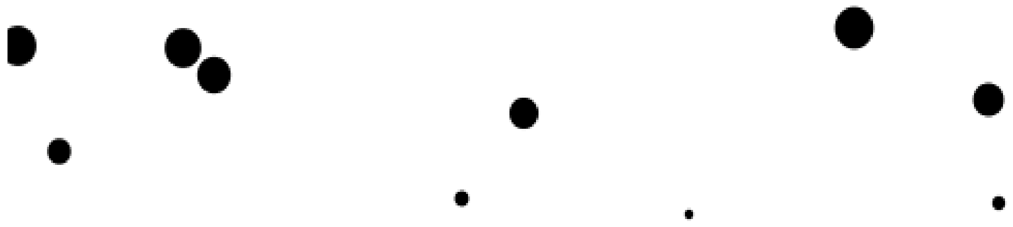
- Also, instead of specifying the `rects'` attributes: `x, y, width,` and `height`, circles need `cx, cy,` and `r`:

```
   .attr("cx", function(d) {
        return d[0];
   })
   .attr("cy", function(d) {
        return d[1];
   })
   .attr("r", 5);
```

Size

- Maybe you want the circles to be different sizes, so their radii correspond to their `y` values. Instead of setting all `r` values to 5, try:

```
.attr("r", function(d) {
    return Math.sqrt(h - d[1]);

});
```



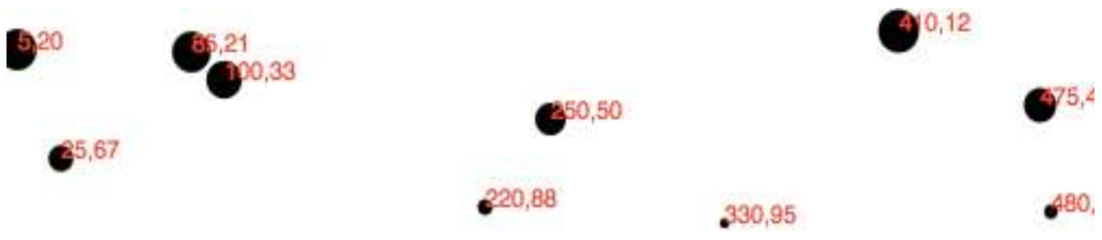
Labels

- We could label the data points with text elements.

```
svg.selectAll("text")
    .data(dataset)
    .enter()
    .append("text")
```

- So far, this looks for all text elements in the SVG (there aren't any yet), and then appends a new text element for each data point. Use the text() method to specify each element's contents:

```
.text(function(d) {
      return d[0] + "," + d[1];
 })
```



Axes

- Note that the axis functions are SVG-specific, as they generate SVG elements. We use `d3.svg.axis()` to create a generic axis function:

```
var xAxis = d3.svg.axis();
```

- At a minimum, each axis also needs to be told on what *scale* to operate.

```
xAxis.scale(xScale);
```

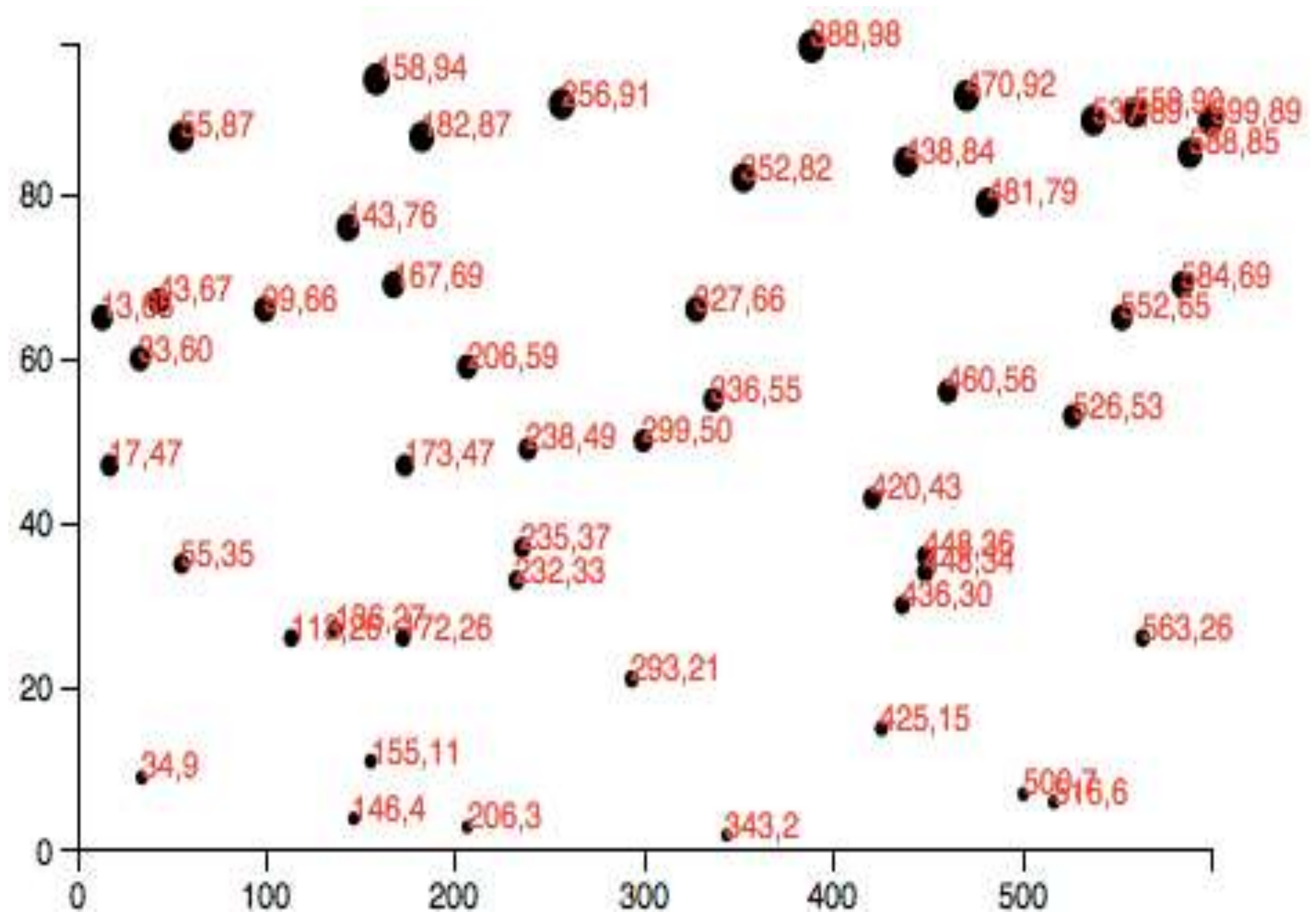
- We can also specify where the labels should appear relative to the axis itself. The default is bottom, meaning the labels will appear below the axis line. (Although this is the default, it can't hurt to specify it explicitly.)

```
xAxis.orient("bottom");
```

- Of course, we can be more concise and string all this together into one line:

```
var xAxis = d3.svg.axis().scale(xScale).orient("bottom");
```

Axes, `24_scatter_plot_labels.html`



24_scatter_plot_labels.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
  <title>D3: A simple scatterplot with value labels</title>
  <script type="text/javascript" src="../d3/d3.js"></script>
  <style type="text/css">/* No style rules */      </style>
</head>
  <body>
        <script type="text/javascript">
        //Width and height
           var w = 500; var h = 100;
           var dataset = [
                 [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                 [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]];
        //Create SVG element
        var svg = d3.select("body")
        .append("svg").attr("width", w).attr("height",h);
```

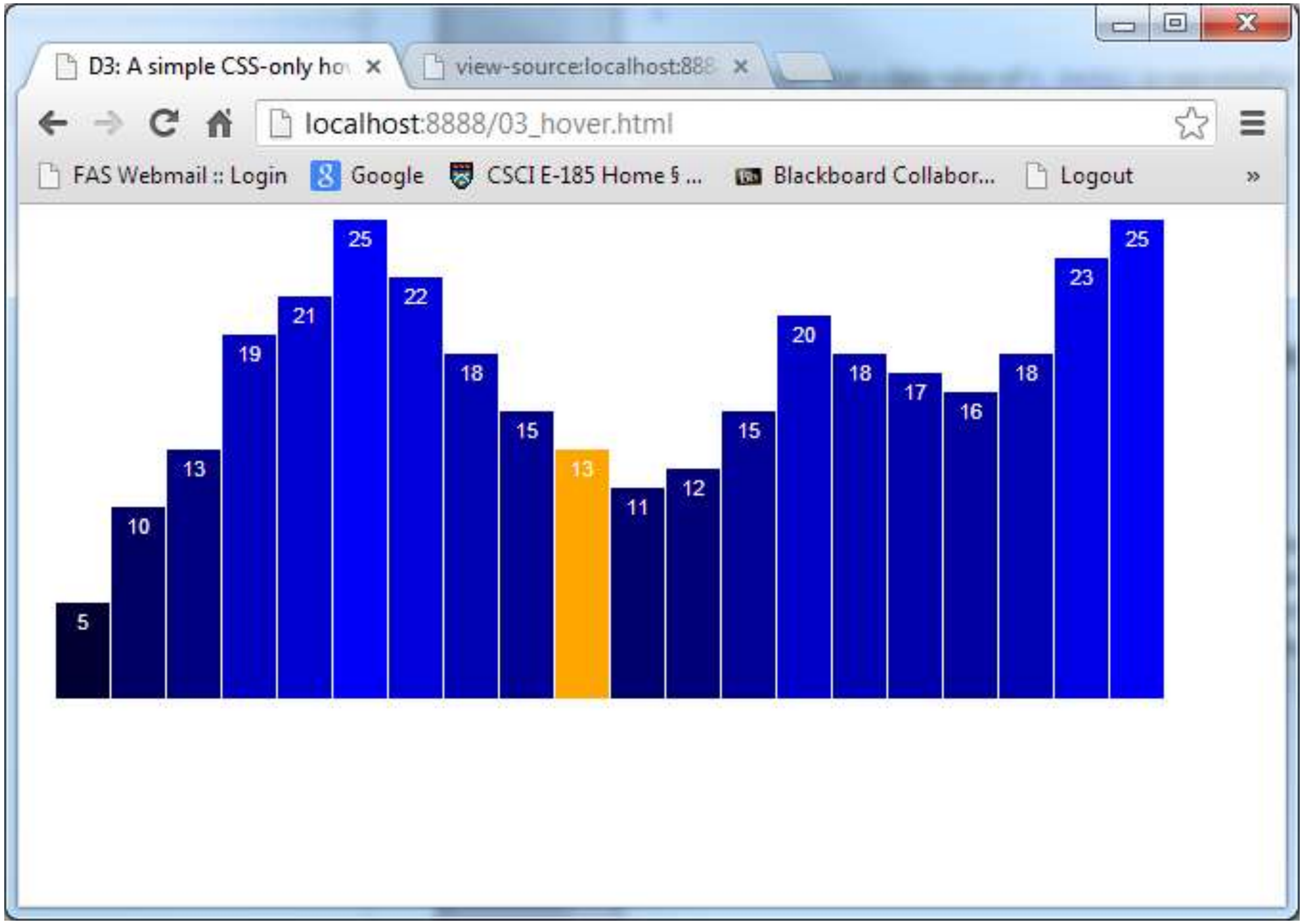
24_scatter_plot_labels.html

```
        svg.selectAll("circle")
               .data(dataset).enter().append("circle")
               .attr("cx", function(d) { return d[0];  })
               .attr("cy", function(d) { return d[1];  })
              .attr("r", function(d) {
                          return Math.sqrt(h - d[1]); });
        svg.selectAll("text")
              .data(dataset)
               .enter()
               .append("text")
               .text(function(d) {
                          return d[0] + "," + d[1]; })
               .attr("x", function(d) {return d[0]; })
               .attr("y", function(d) {return d[1]; })
               .attr("font-family", "sans-serif")
               .attr("font-size", "11px")
               .attr("fill", "red");
               </script>
        </body>
</html>
```

Interactivity, `03_hover.html`

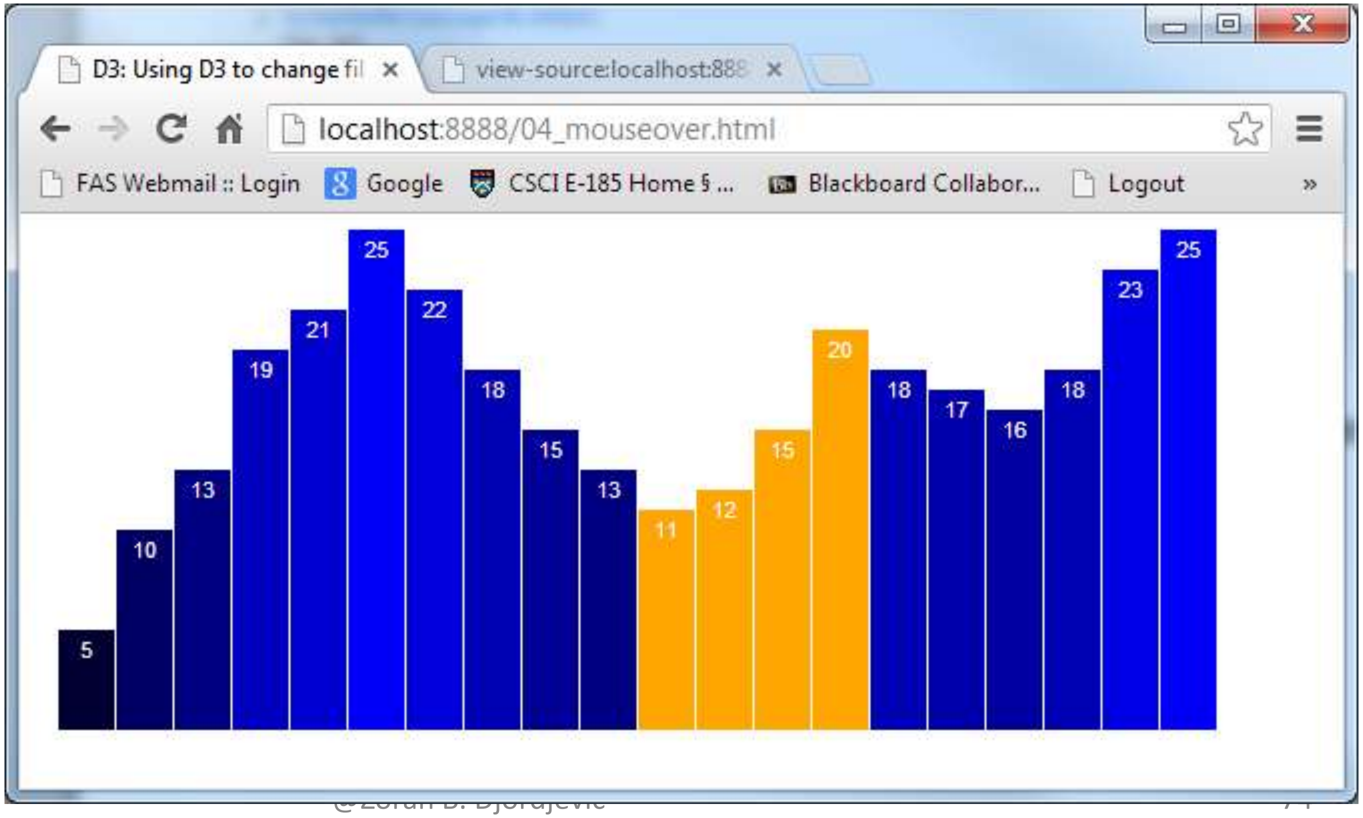
- D3 allows us to bind event listeners to all objects and
- Define response to those events. Some events can be dealt with CSS alone, like `hover:`

```
<style
type="text/css">
   rect:hover {
    fill: orange
   }
</style>
```



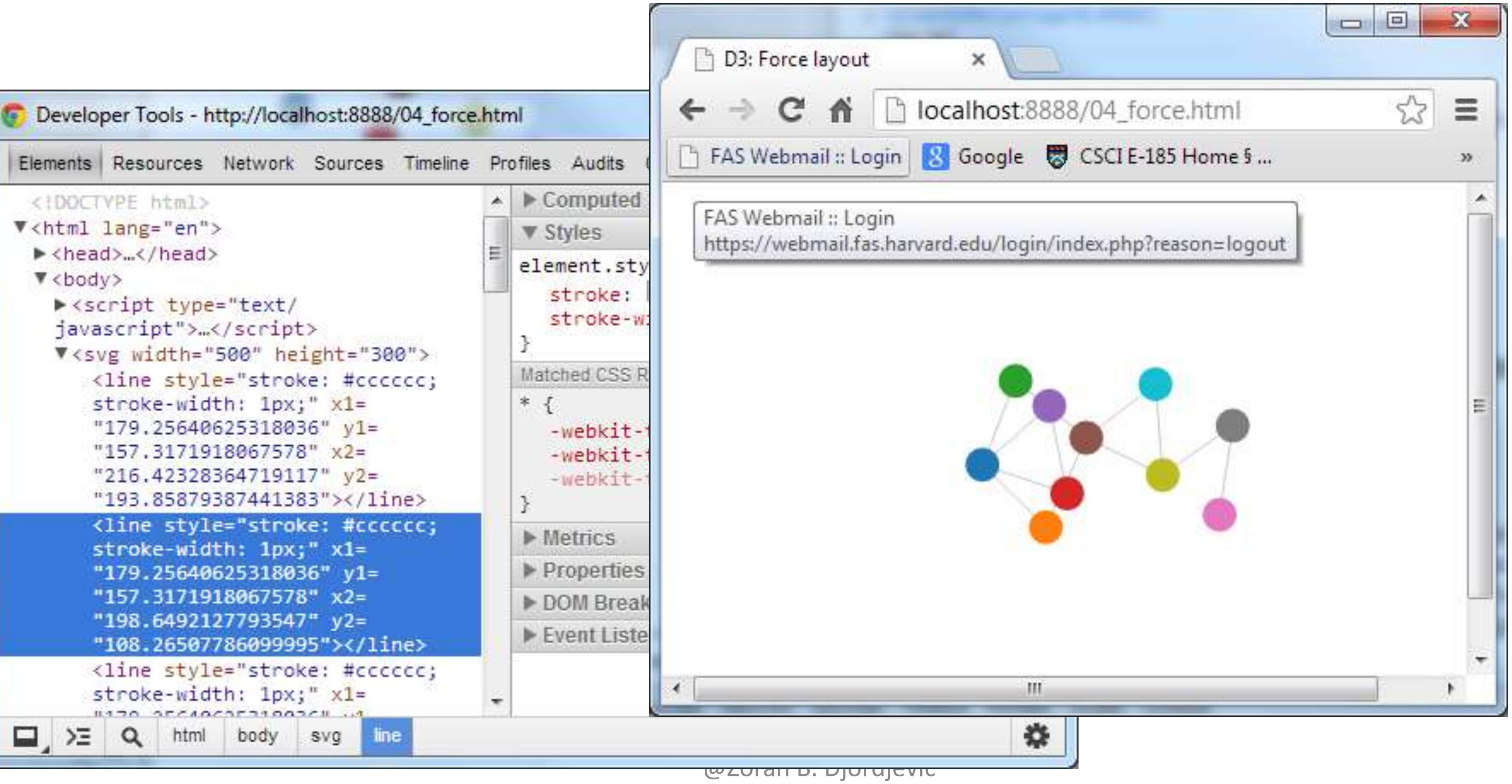
D3 and SVG give you great flexibility

```
//Create bars
svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .on("mouseover", function() {
        d3.select(this)
       .attr("fill", "orange");
});
```



Force-directed layouts, `04_force.html`

- Force layouts are typically used with network data they represent graph.
- Graphs have nodes and edges.
- We can position nodes as if they are dynamic particles repelling each outer while connected by springs that bind them in a cluster.



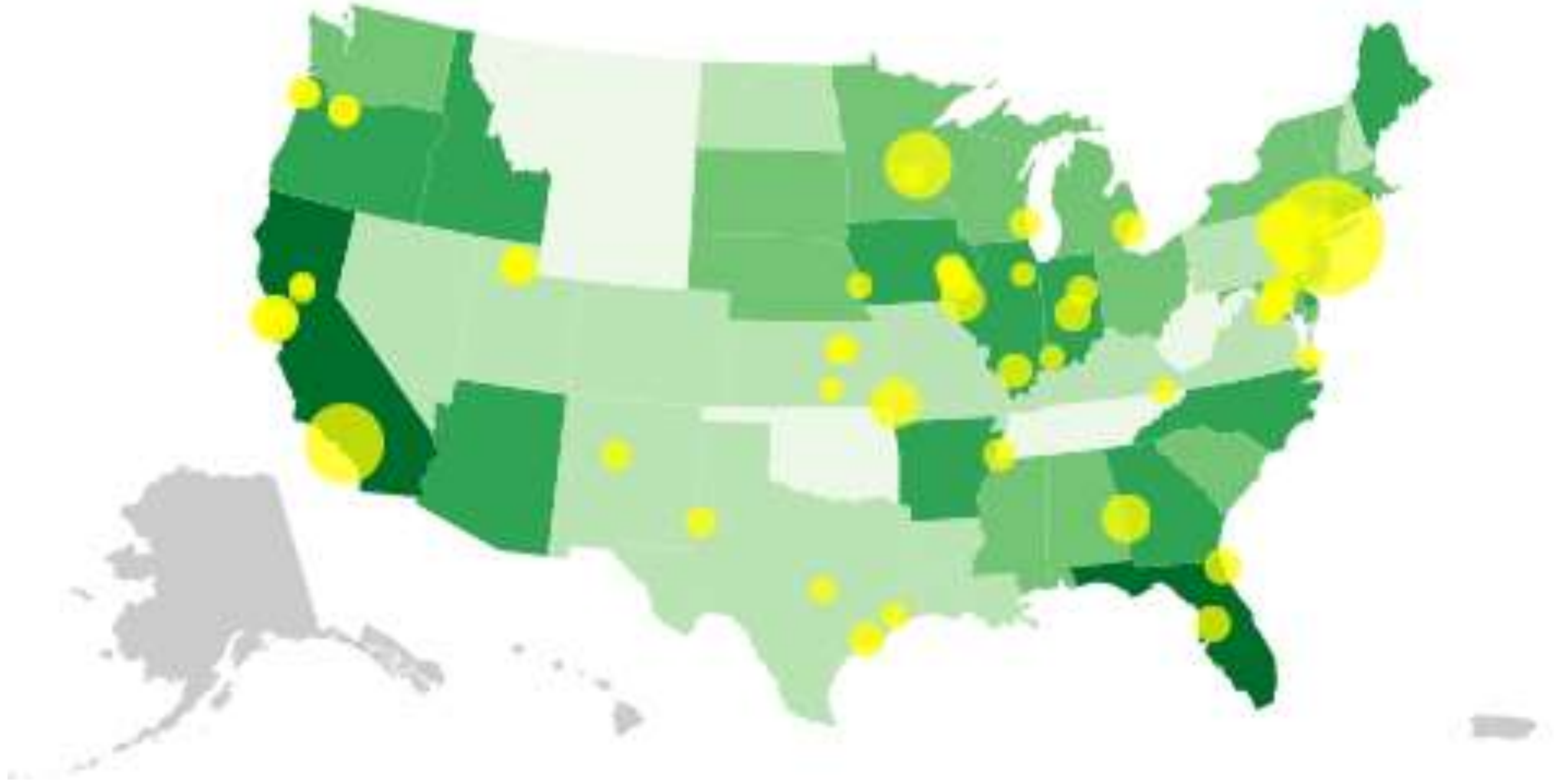
```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<script type="text/
    javascript">…</script>
    ▼<svg width="500" height="300">
        <line style="stroke: #cccccc;
        stroke-width: 1px;" x1=
        "179.25640625318036" y1=
        "157.3171918067578" x2=
        "216.42328364719117" y2=
        "193.85879387441383"></line>
        <line style="stroke: #cccccc;
        stroke-width: 1px;" x1=
        "179.25640625318036" y1=
        "157.3171918067578" x2=
        "198.6492127793547" y2=
        "108.26507786099995"></line>
        <line style="stroke: #cccccc;
        stroke-width: 1px;" x1=
```

```
element.sty
  stroke:
  stroke-w
}
* {
  -webkit-
  -webkit-
  -webkit-
}
```

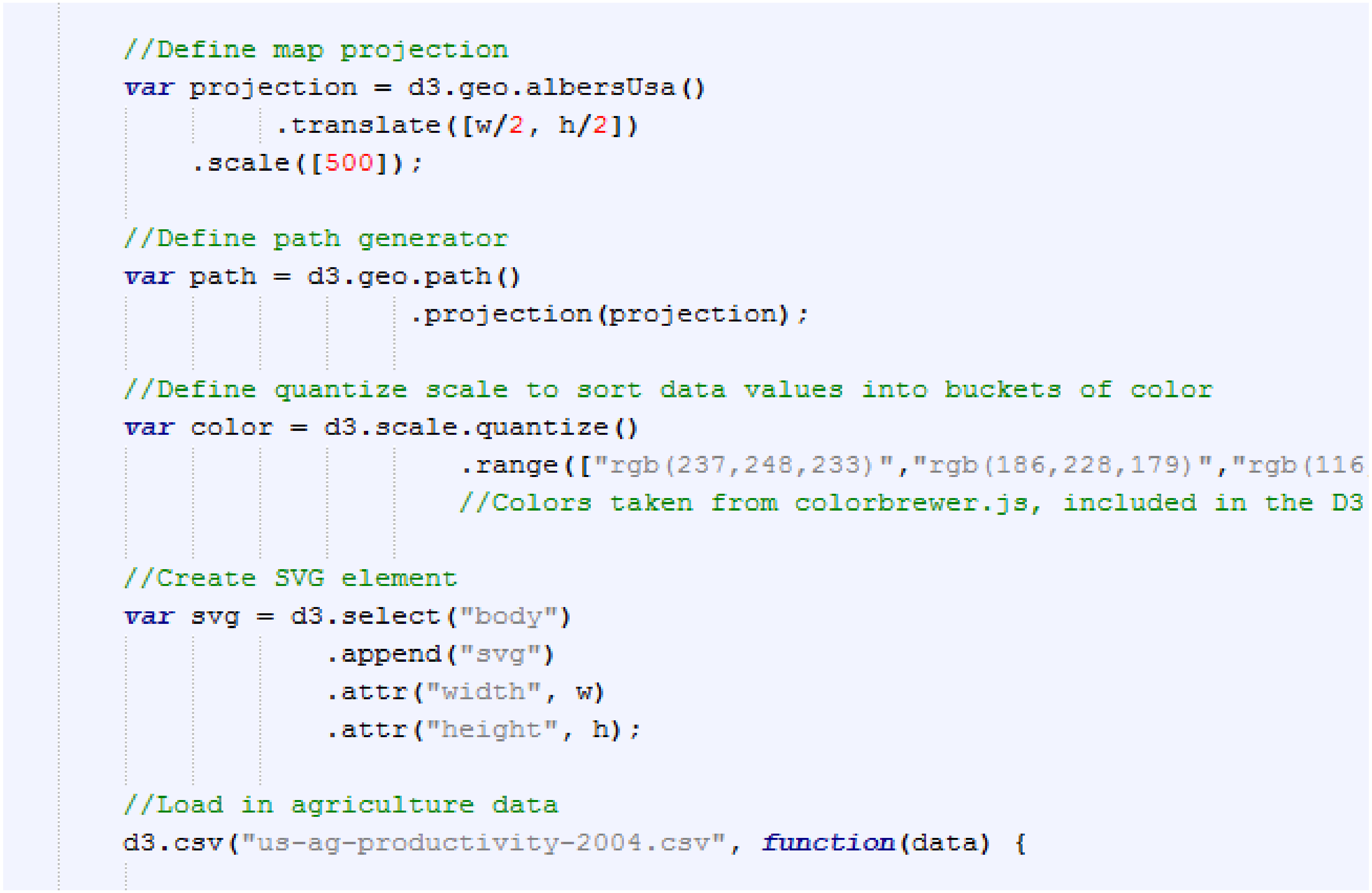
FAS Webmail :: Login
https://webmail.fas.harvard.edu/login/index.php?reason=logout

Geo Mapping, `07_points_sized.html`

- D3 appears to come specially well equipped to deal with geographic data, maps and projections.
- D3 understands GeoJSON, a JSON based standard for geo-data on web application.



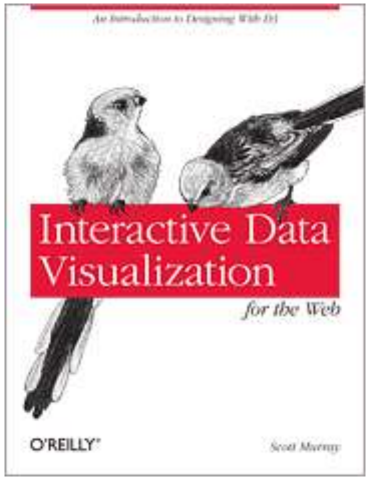
07_points_sized.html



```
//Define map projection
var projection = d3.geo.albersUsa()
         .translate([w/2, h/2])
    .scale([500]);

//Define path generator
var path = d3.geo.path()
                 .projection(projection);

//Define quantize scale to sort data values into buckets of color
var color = d3.scale.quantize()
                    .range(["rgb(237,248,233)","rgb(186,228,179)","rgb(116
                    //Colors taken from colorbrewer.js, included in the D3

//Create SVG element
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

//Load in agriculture data
d3.csv("us-ag-productivity-2004.csv", function(data) {
```

References

- *Interactive Data Visualization for the Web*, by Scott Murray, O'Reilly 2013
- *D3 Tips and Tricks*, by Malcolm Maclean, Leanpub 2013-2015
- *SVG Essentials*, by J. David Eisenberg, O'Reilly, 2002
- *Leaflet Tips and Tricks*, by Malcolm Maclean, Leanpub 2013-2014
- *The functional art, an introduction to information graphics and visualization*, by Alberto Cairo, New Readers, 2013
- *The Visual Display of Quantitative Information*, by Edward R. Tufte, Graphics Press, 2001


An Introduction to Designing With D3
Interactive Data Visualization
for the Web
O'REILLY
Scott Murray