CHAPTER 5
Data

Data is an extremely broad term, only slightly less vague than the nearly all-encompassing *information*. What is data? (What *isn't* data?) What kinds of data are there, and what can we use with D3?

Broadly speaking, data is structured information with potential for meaning.

In the context of programming for visualization, data is stored in a digital file, typically in either text or binary form. Of course, potentially every piece of digital ephemera may be considered "data"—not just text, but bits and bytes representing images, audio, video, databases, streams, models, archives, and anything else.

Within the scope of D3 and browser-based visualization, however, we will limit ourselves to *text-based data*—that is, anything that can be represented as numbers and strings of alpha characters. If you can get your data into a *.txt* plain-text file, a *.csv* comma-separated value file, or a *.json* JSON document, then you can use it with D3.

Whatever your data, it can't be made useful and visual until it is *attached* to something. In D3 lingo, the data must be *bound* to elements within the page. Let's address how to create new page elements first. Then attaching data to those elements will be a cinch.

Generating Page Elements

Typically, when using D3 to generate new DOM elements, the new elements will be circles, rectangles, or other visual forms that represent your data. But to avoid confusing matters, we'll start with a simple example and create a lowly `p` paragraph element.

Begin by creating a new document with our simple HTML template from the last chapter. You can find it in the sample code files as *01_empty_page_template.html*, and it looks like the following code. (Eagle-eyed viewers will notice that I've modified the src path here to work with the directory structure of the code samples. If that doesn't mean anything to you, don't worry about it.)

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

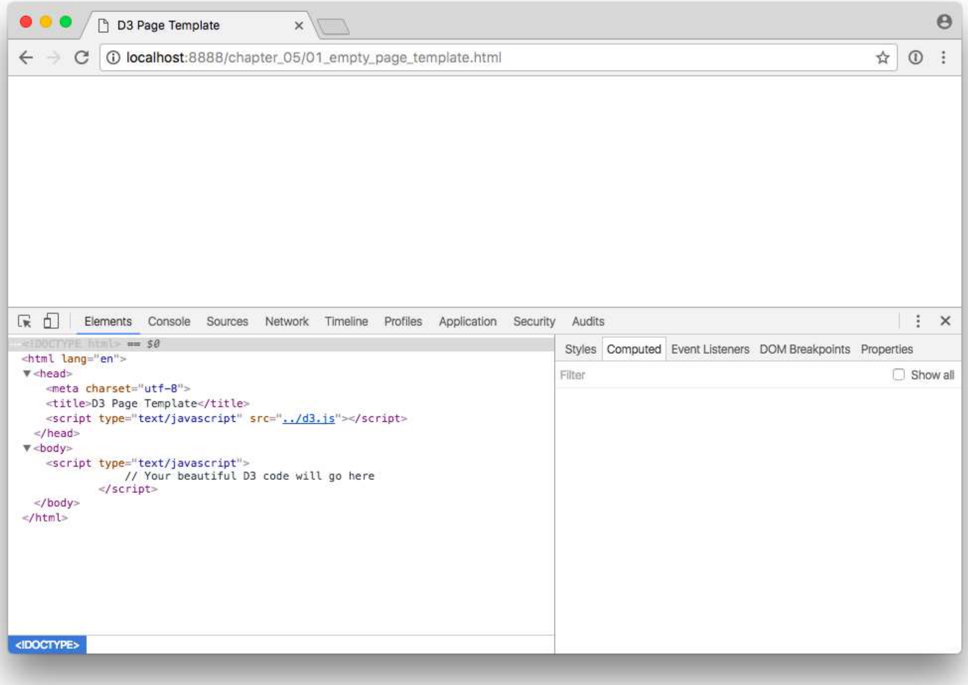
Open that page in your web browser. Make sure you're accessing the page via your local web server, as we discussed in Chapter 4. So the URL in your browser's location bar should look something like this:

```
http://localhost:8888/d3-book/chapter_05/01_empty_page_template.html
```

If not viewed through a web server, the URL path will start with *file:///* instead of *http://*. Confirm that the URL does *not* look like this:

```
file:///…/d3-book/chapter_05/01_empty_page_template.html
```

Once you're viewing the page, pop open the web inspector. (As a reminder, see "Developer Tools" on page 26 on how to do that.) You should see an empty web page, with the DOM contents shown in Figure 5-1.



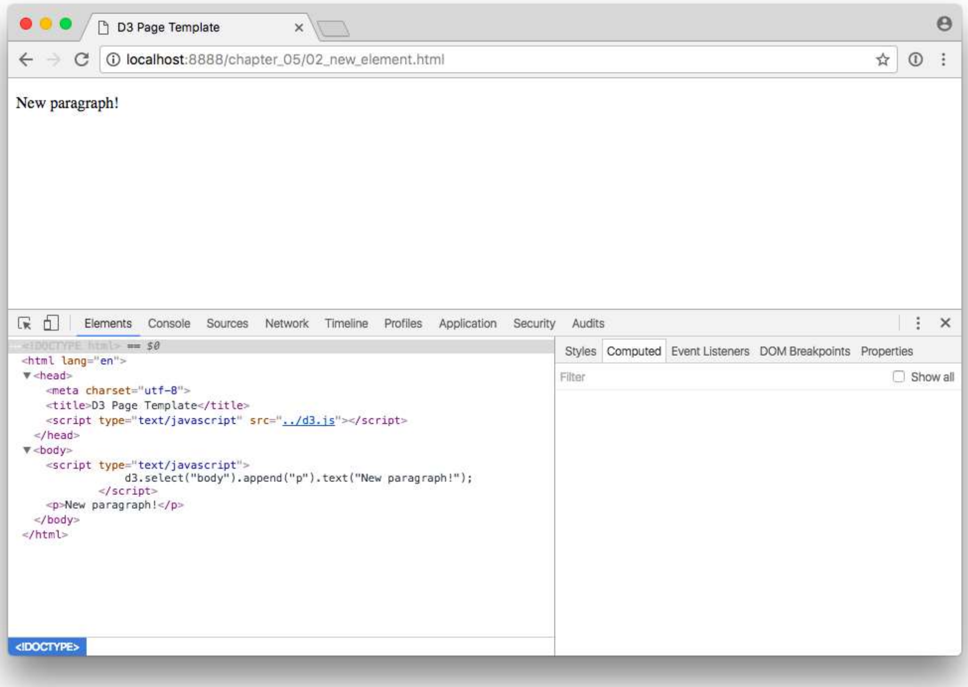
```
<!DOCTYPE html> == $0
<html lang="en">
▼<head>
    <meta charset="utf-8">
    <title>D3 Page Template</title>
    <script type="text/javascript" src="../d3.js"></script>
  </head>
▼<body>
    <script type="text/javascript">
            // Your beautiful D3 code will go here
        </script>
  </body>
</html>
```

Figure 5-1. Web inspector

Back in your text editor, replace the comment between the `script` tags with:

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

Save and refresh, and voilà! There is text in the formerly empty browser window, and the web inspector will look like Figure 5-2.



New paragraph!

```
<!DOCTYPE html> == $0
<html lang="en">
▼<head>
    <meta charset="utf-8">
    <title>D3 Page Template</title>
    <script type="text/javascript" src="../d3.js"></script>
  </head>
▼<body>
    <script type="text/javascript">
            d3.select("body").append("p").text("New paragraph!");
        </script>
    <p>New paragraph!</p>
  </body>
</html>
```

Figure 5-2. Web inspector, reflecting the modified DOM

See the difference? Now in the DOM, there is a new paragraph element that was generated on the fly! This might not be exciting yet, but you will soon use a similar technique to dynamically generate tens or hundreds of elements, each one corresponding to a piece of your dataset.

Let's walk through what just happened. (You can follow along with *02_new_element.html*.) To understand that first line of D3 code, you must first meet your new best friend, *chain syntax*.

Chaining Methods

D3 smartly employs a technique called *chain syntax*, which you might recognize from jQuery. By "chaining" methods together with periods, you can perform several actions in a single line of code. It can be fast and easy, but it's important to understand how it works, to save yourself hours of debugging headaches later.

By the way, *functions* and *methods* are just two different words for the same concept: a chunk of code that accepts an argument as input, performs some action, and returns some other information as output.

The following code:

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

might look like a big mess, especially if you're new to programming. So the first thing to know is that 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!");
```

Both I and your optometrist highly recommend putting each method on its own indented line. But programmers have their own coding style; use whatever indents, line breaks, and whitespace (tabs or spaces) are most legible for you.

One Link at a Time

Let's deconstruct each link in this chain of code:

`d3`

References the D3 object, so we can access its methods. Our D3 adventure begins here.

`.select("body")`

Give the `select()` method 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.) In this case, we just want the `body` of the document, so a reference to `body` is handed off to the next method in our chain.

`.append("p")`

`append()` creates whatever new DOM element you specify and appends it to the end (but *just inside*) of whatever selection it's 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. So an empty `p` paragraph is *appended* to the `body`. Finally, `append()` hands off 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. Because the previous method passed down a reference to our new `p`, this code just inserts the new text between `<p>` and `</p>`. (In cases where there is existing content, it will be overwritten.)

`;`
: The all-important semicolon indicates the end of this line of code. Chain over.

The Handoff

Many, but not all, D3 methods return a selection (actually, a 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 on, but not always.

So remember this: when you're 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 handoff will function more like a dropped baton in a middle-school relay race.

When sussing out what each function expects and returns, the API reference is your friend. 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:

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

Ugh! What a mess. Yet 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. Or because you want to organize your code in a way that makes more sense to you.

Now that you know how to generate new page elements with D3, it's time to attach data to them.

Binding Data

What is binding, and why would I want to do it to my data?

Data visualization is a process of *mapping* data to visuals. Data in, visual properties out. Maybe bigger numbers make taller bars, or special categories trigger brighter colors. The mapping rules are up to you.

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, unmappable DOM elements. No one wants that.

In a Bind

We use D3's `data()` method to bind data to DOM elements. But there are two things we need in place first, before we can bind data:

- The data
- A selection of DOM elements

Let's tackle these one at a time.

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). It can handle JSON (and GeoJSON) gracefully, and even has a built-in method to help you load in CSV files.

But to keep things simple, for now we will start with a boring array of five numbers. Here is our sample dataset:

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

If you're feeling adventurous, or already have some data in CSV or JSON format that you want to play with, we'll see how to do that now. Otherwise, just skip ahead to "Please Make Your Selection" on page 79.

Loading CSV data

CSV stands for comma-separated values. A CSV datafile might look something like this:

```
Food,Deliciousness
Apples,9
Green Beans,5
Egg Salad Sandwich,4
Cookies,10
Liver,0.2
Burrito,7
```

Each line in the file has the same number of values (two, in this case), and values are separated by a comma. The first line in the file often serves as a header, providing names for each of the "columns" of data.

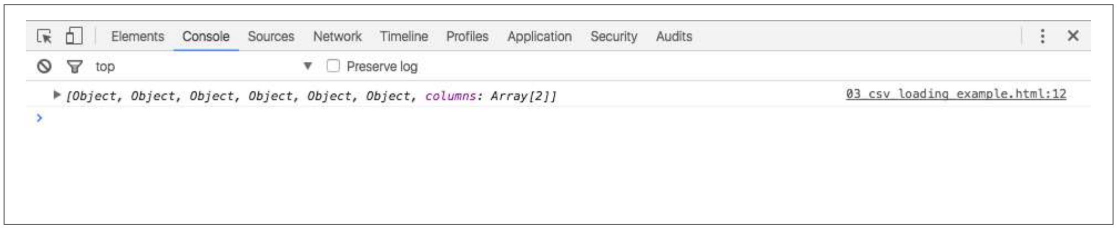
If you have data in an Excel file, it probably follows a similar structure of rows and columns. To get that data into D3, open it in Excel, then choose "Save as…" and select CSV as the file type.

If we saved the preceding CSV data into a file called *food.csv*, then we could load the file into D3 by using the `d3.csv()` method:

```
d3.csv("food.csv", function(data) {
    console.log(data);
});
```

`csv()` takes two arguments: a string representing the path of the CSV file to load in, and an anonymous function to be used as a *callback function*. The callback function is "called" only *after* the CSV file has been loaded into memory. So you can be sure that, by the time the callback is called, `d3.csv()` is done executing.

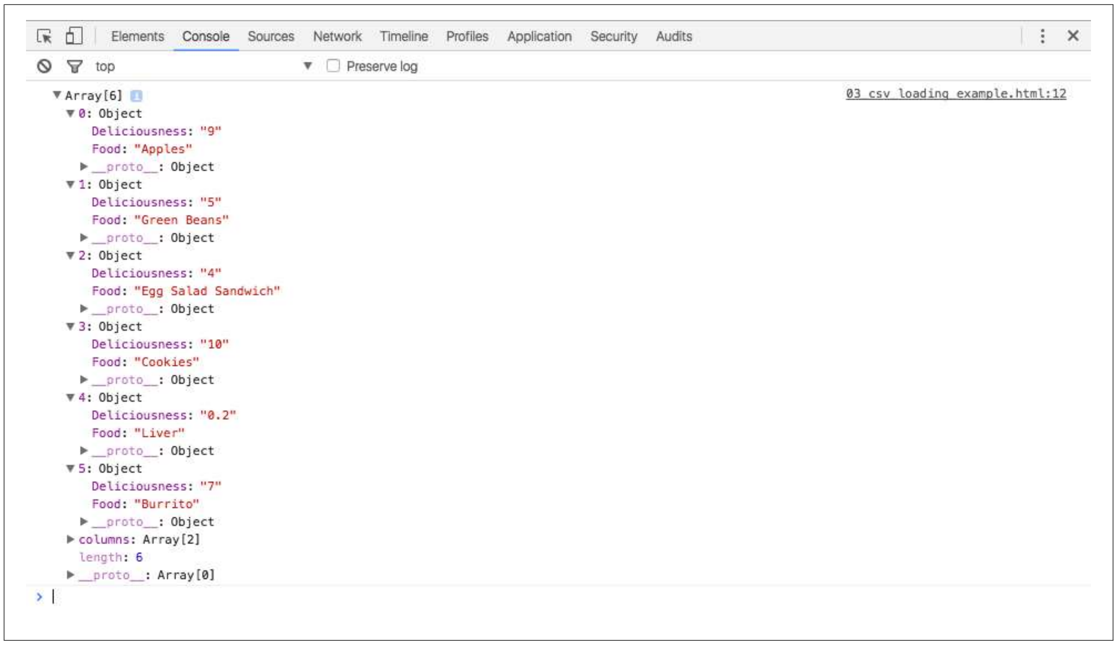
When called, the anonymous function is handed the result of the CSV loading and parsing process—that is, the data. Here I'm naming it `data`, but this could be called whatever you like. You should use this callback function to do all the things you can do only *after* the data has been loaded. In the preceding example, we are just logging the value of the `data` array to the console, to verify it, as shown in Figure 5-3. (See *03_csv_loading_example.html* in the example code.)



```
▶ [Object, Object, Object, Object, Object, Object, columns: Array[2]]      03_csv_loading_example.html:12
```

Figure 5-3. Array logged to console

You can see that `data` is an array (because of the hard brackets `[]` on either end) with six elements, each of which is an object. By toggling the disclosure triangles next to each object, we can see their values (see Figure 5-4).



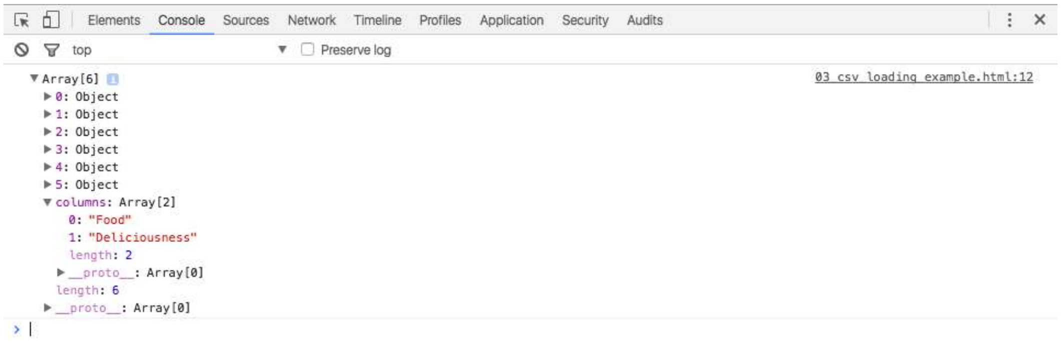
```
▼Array[6]
  ▼0: Object
      Deliciousness: "9"
      Food: "Apples"
    ▶__proto__: Object
  ▼1: Object
      Deliciousness: "5"
      Food: "Green Beans"
    ▶__proto__: Object
  ▼2: Object
      Deliciousness: "4"
      Food: "Egg Salad Sandwich"
    ▶__proto__: Object
  ▼3: Object
      Deliciousness: "10"
      Food: "Cookies"
    ▶__proto__: Object
  ▼4: Object
      Deliciousness: "0.2"
      Food: "Liver"
    ▶__proto__: Object
  ▼5: Object
      Deliciousness: "7"
      Food: "Burrito"
    ▶__proto__: Object
  ▶columns: Array[2]
   length: 6
  ▶__proto__: Array[0]
> |
```

Figure 5-4. Array elements expanded

Aha! Each object has both a `Food` property and a `Deliciousness` property, the values of which correspond to the values in our CSV! (There is also a third property, `__proto__`, but that has to do with how JavaScript handles objects, and you can ignore it for now.) D3 has employed the first row of the CSV for property names, and subsequent rows for values. You might not realize it, but this just saved you a *lot* of time.

A Handy Listing of CSV Column Names

You may have noticed a sneaky *seventh* item in the `data` array named `columns`. D3 helpfully stores all the column names detected in your CSV here, as seen in Figure 5-5.



```
▼Array[6]
  ▶0: Object
  ▶1: Object
  ▶2: Object
  ▶3: Object
  ▶4: Object
  ▶5: Object
  ▼columns: Array[2]
      0: "Food"
      1: "Deliciousness"
      length: 2
    ▶__proto__: Array[0]
   length: 6
  ▶__proto__: Array[0]
> |
```

Figure 5-5. The secret columns array

Notice how the presence of `columns` doesn't affect the tallied length of the array: `Array[6]` indicates that the `array.length` is still 6, because only the numerically indexed items are counted. Arrays in JavaScript are just objects, and `columns` is just a named property added on like any other. `columns`, however, is ignored for purposes of data binding, so you can ignore it, too.

One more thing to note is that each value from the CSV is stored as a string, even the numbers. (You can tell because 9 is surrounded by quotation marks, as in `"9"` and not simply `9`.) This could cause unexpected behavior later, if you try to reference your data as a numeric value but it is still typed as a string.

To save yourself a debugging headache later, you can specify a row conversion function, in which you specify how the values in each row of the CSV should be typed. In our example, the `Food` column contains strings already, so it needs no conversion. But `Deliciousness` contains integers and floats. Next, I define a new function, `rowConverter`, and tell it how to handle each column.

```
var rowConverter = function(d) {
    return {
        Food: d.Food,  //No conversion
        Deliciousness: parseFloat(d.Deliciousness)
    };
}

d3.csv("food.csv", rowConverter, function(data) {
    console.log(data);
});
```

Note that `rowConverter` has been included as a new parameter passed into `d3.csv()`. So now `d3.csv()` will load the file in, run each row through the row conversion function, and finally store it all in `data`. You can verify this for yourself; in the console, you'll see `Deliciousness: 9` instead of `Deliciousness: "9"`.

Handling Data-Loading Errors

Note that `d3.csv()` is an *asynchronous* method, meaning that the rest of your code is executed even while JavaScript is simultaneously waiting for the file to finish downloading into the browser. (The same is true of D3's other functions that load external resources, such as `d3.json()`.)

This can potentially be *very* confusing, because you—as a reasonable human person—might assume that the CSV file's data is available, when in fact it hasn't finished loading yet. A common mistake is to include references to the external data *outside of* the callback function. Save yourself some headaches and make sure to reference your data only from *within* the callback function (or from within other functions that you call within the callback function).

Personally, I like to declare a global variable first, then call `d3.csv()` to load the data. Within the callback function, I copy the data into my global variable, and finally I call any functions that rely on that data being present. By declaring a global variable and storing your data inside that global variable, you can ensure that the data is available later to any subsequent functions, *even outside of* `d3.csv()`. For example:

```
var dataset;  //Declare global variable, initially empty (undefined)

d3.csv("food.csv", function(data) {
    dataset = data;    //Once loaded, copy to dataset.
    generateVis();     //Then call other functions that
    hideLoadingMsg();  //depend on data being present.
});

var useTheDataLater = function() {
    //Assuming useTheDataLater() is called sometime after
    //d3.csv() has successfully loaded in the data,
    //then the global dataset would be accessible here.
};
```

To further confuse matters, the callback function is executed *whether or not the datafile was loaded successfully*. That's right: if the network connection fails, or the filename is misspelled, or for some reason an error occurs on the web server end, the callback function will *still* be executed. When the data fails to load, and you call functions that rely on that data being present, you will probably see an error in the console, and the visualization won't be created. This scenario might happen only rarely, but it's useful to know how to handle it.

Fortunately, you can include an optional `error` parameter in the callback function definition. If there is a problem loading the file, then `error` will be set to the error message returned by the web server, and `data` will be `undefined`. If the file loads successfully and there is no error, then `error` will be `null`, and the `data` array will be populated as expected. Note that `error` must be the first parameter, and `data` the second:

```
var dataset;

d3.csv("food.csv", function(error, data) {

    if (error) {  //If error is not null, something went wrong.
        console.log(error);  //Log the error.
    } else {      //If no error, the file loaded correctly. Yay!
        console.log(data);   //Log the data.

        //Include other code to execute after successful file load here
        dataset = data;
        generateVis();
```

```
            hideLoadingMsg();
        }

    });
```

Verifying your data is a great use of the `csv()` callback function, but typically this is where you'd call other functions that construct the visualization, now that the data is available, as in:

```
var dataset;  //Declare global var

d3.csv("food.csv", function(data) {

    //Hand CSV data off to global var,
    //so it's accessible later.
    dataset = data;

    //Call some other functions that
    //generate your visualization, e.g.:
    generateVisualization();
    makeAwesomeCharts();
    makeEvenAwesomerCharts();
    thankAwardsCommittee();

});
```

One more tip: if you have *tab*-separated data in a TSV file, try the `d3.tsv()` method, which otherwise behaves exactly as the preceding method.

Loading JSON data

We'll spend more time talking about JSON later, but for now, all you need to know is that the `d3.json()` method works the same way as `csv()`. Load your JSON data in this way:

```
d3.json("waterfallVelocities.json", function(json) {
    console.log(json);  //Log output to console
});
```

Here, I've named the parsed output `json`, but it could be called `data` or whatever you like.

Please meet the indispensable Mr. Data Converter, a project by Shan Carter, an early and prolific D3 user formerly of the *New York Times*. Mr. Data Converter takes your Excel, CSV, or tab-separated data and converts it to JSON or several other formats. It is one of those great tools that does a single thing extremely well; bookmark it now.

Please Make Your Selection

The data is ready to go. As a reminder, we are working with this simple array:

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

Now you need to decide what to select. That is, what elements will your data be associated with? Again, let's keep it super simple and say that we want to make a new paragraph for each value in the dataset. So you might imagine something like this would be helpful:

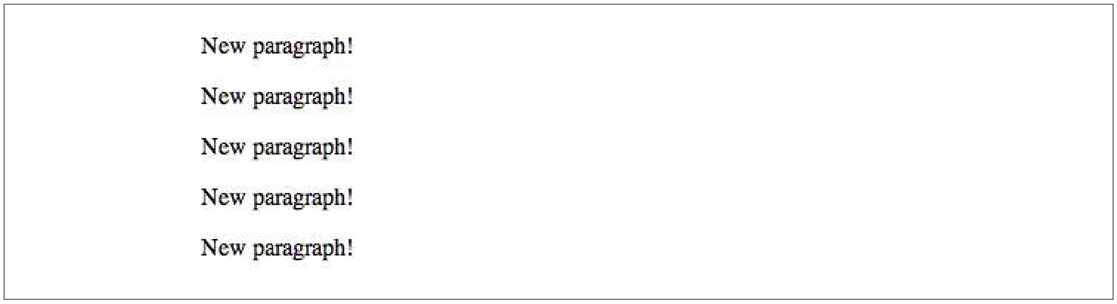
```
d3.select("body").selectAll("p")
```

and you'd be right, but there's a catch: the paragraphs we want to select *don't exist yet*. And this gets at one of the most common points of confusion with D3: how can we select elements that don't yet exist? Bear with me, as the answer might require bending your mind a bit.

The answer lies with `enter()`, a truly magical method. See this code, which I'll explain:

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

View the example code *04_creating_paragraphs.html* and you should see five new paragraphs, each with the same content, as shown in Figure 5-6.



New paragraph!

New paragraph!

New paragraph!

New paragraph!

New paragraph!

Figure 5-6. Dynamic paragraphs

Here's what's happening:

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

`.selectAll("p")`

Selects all paragraphs in the DOM. Because none exist yet, this returns an empty selection. Think of this empty selection as representing the paragraphs that *will soon exist*.

`.data(dataset)`

Counts and parses our data values. There are five values in our array called `dataset`, 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 current DOM selection, 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* on which you can work your magic. It then hands off a reference to this new placeholder to the next step in the chain.

`.append("p")`

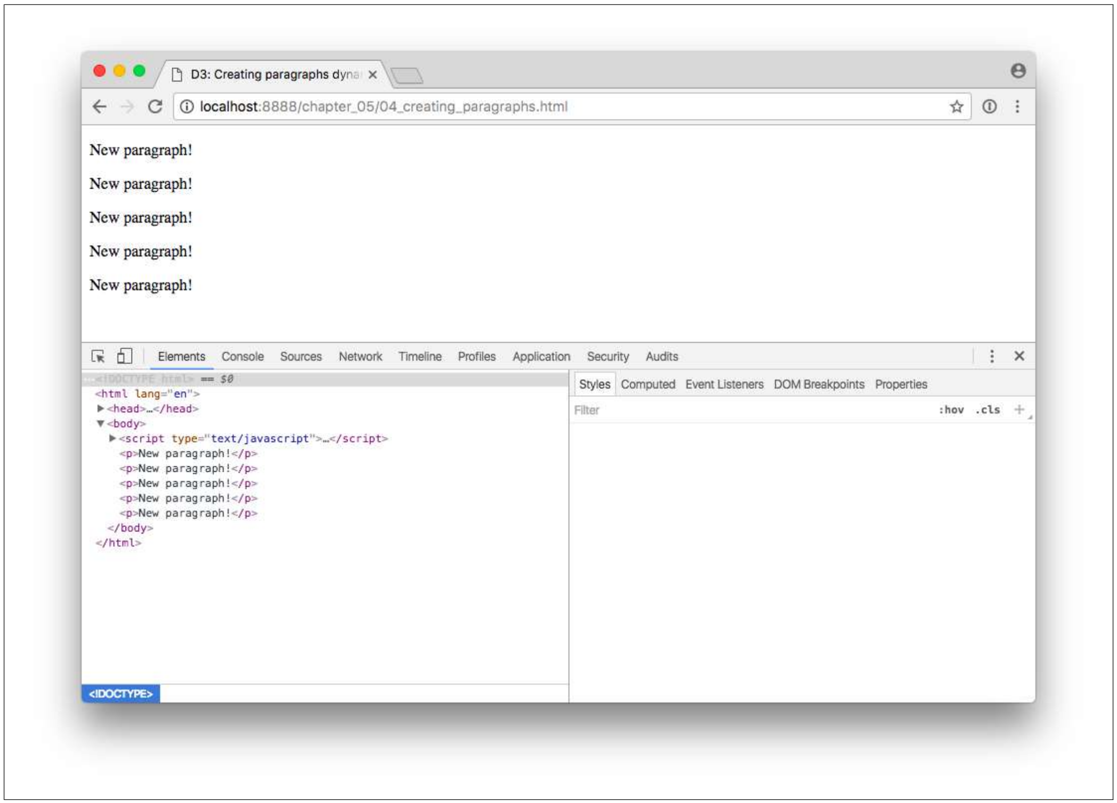
Takes the empty placeholder selection created by `enter()` and appends a `p` element into the DOM. Hooray! Then it hands off a reference to the element it just created to the next step in the chain.

`.text("New paragraph!")`

Takes the reference to the newly created `p` and inserts a text value.

Bound and Determined

All right! Our data has been read, parsed, and bound to new `p` elements that we created in the DOM. Don't believe me? Take another look at *04_creating_paragraphs.html* and whip out your web inspector, shown in Figure 5-7.



New paragraph!

New paragraph!

New paragraph!

New paragraph!

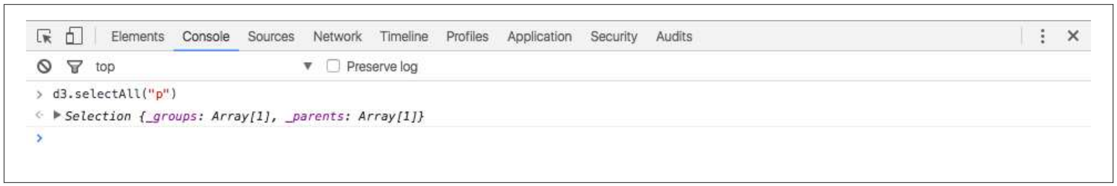
New paragraph!

```
<!DOCTYPE html> == $0
<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>
```

Figure 5-7. New p elements in the web inspector

Okay, I see five paragraphs, but where's the data? Switch to the JavaScript console, type in the following code, and click Enter. The results are shown in Figure 5-8.

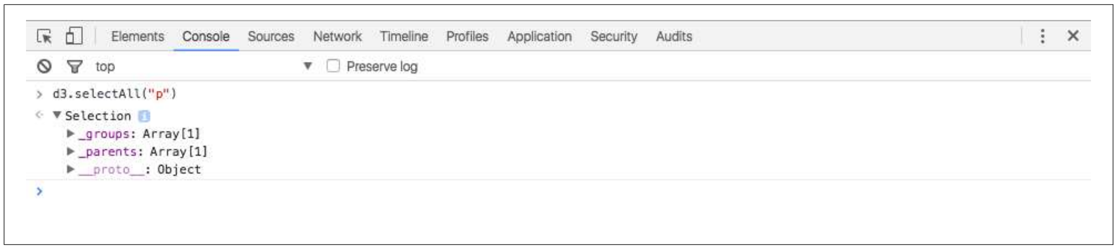
```
d3.selectAll("p")
```



```
> d3.selectAll("p")
< ▶Selection {_groups: Array[1], _parents: Array[1]}
>
```

Figure 5-8. A selection of five paragraphs

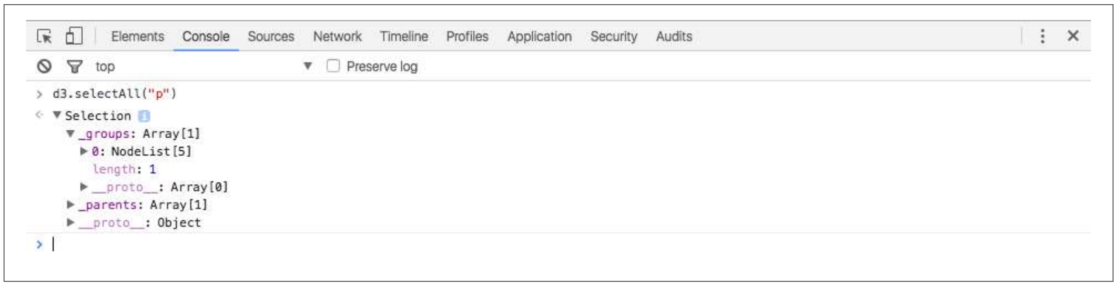
Interesting: the selection is actually an object containing both a `_groups` array and a `_parents` array. Click the gray disclosure triangle to expand the object, shown in Figure 5-9.



```
> d3.selectAll("p")
< ▼Selection
   ▶_groups: Array[1]
   ▶_parents: Array[1]
   ▶__proto__: Object
>
```

Figure 5-9. Selection, expanded

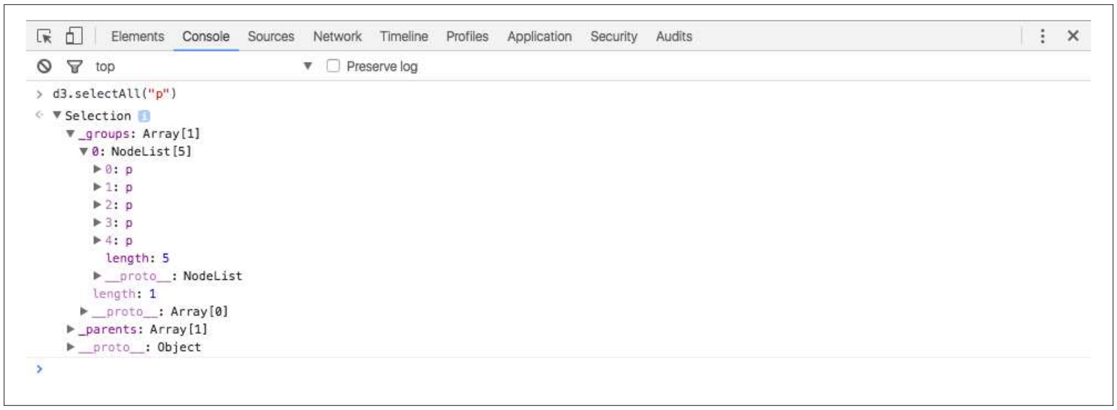
Then let's expand `_groups` to see its contents, shown in Figure 5-10.



```
> d3.selectAll("p")
< ▼Selection
   ▼_groups: Array[1]
     ▶0: NodeList[5]
      length: 1
     ▶__proto__: Array[0]
   ▶_parents: Array[1]
   ▶__proto__: Object
> |
```

Figure 5-10. Our selection's groups

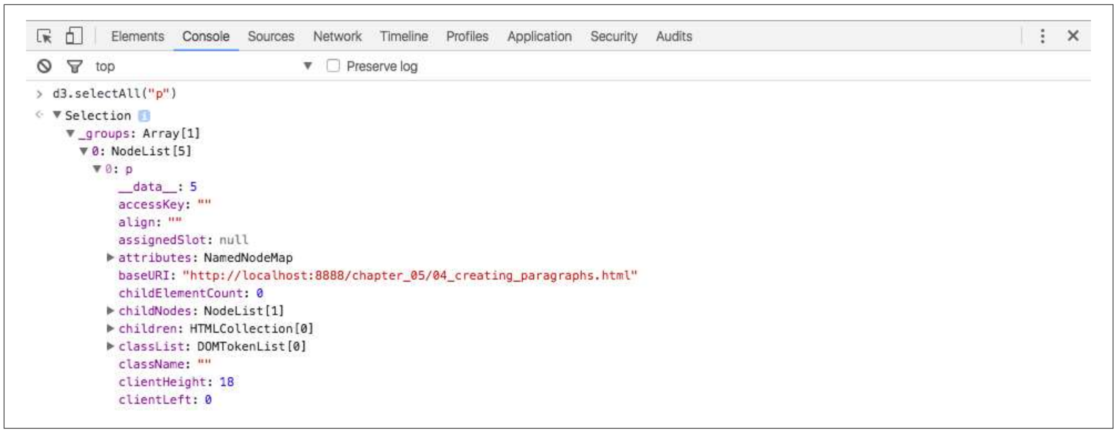
Note that `_groups` contains only one item, a `NodeList` array, itself containing five more items. Let's expand that, as shown in Figure 5-11.



```
> d3.selectAll("p")
< ▼Selection
   ▼_groups: Array[1]
     ▼0: NodeList[5]
       ▶0: p
       ▶1: p
       ▶2: p
       ▶3: p
       ▶4: p
        length: 5
       ▶__proto__: NodeList
      length: 1
     ▶__proto__: Array[0]
   ▶_parents: Array[1]
   ▶__proto__: Object
>
```

Figure 5-11. Expanded NodeList

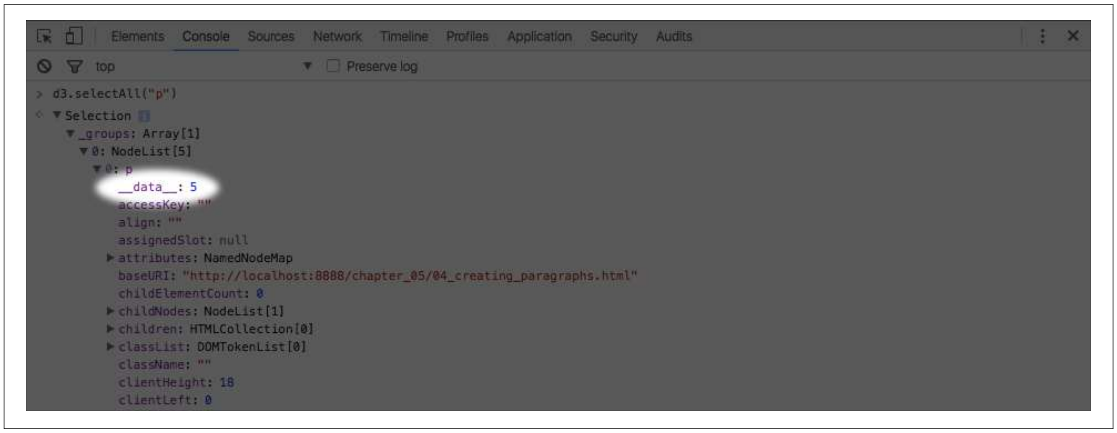
You'll notice the five `p`s, numbered 0 through 4. Click the disclosure triangle next to the first one (number zero), which results in the view shown in Figure 5-12.



```
> d3.selectAll("p")
< ▼Selection
   ▼_groups: Array[1]
     ▼0: NodeList[5]
       ▼0: p
          __data__: 5
          accessKey: ""
          align: ""
          assignedSlot: null
         ▶attributes: NamedNodeMap
          baseURI: "http://localhost:8888/chapter_05/04_creating_paragraphs.html"
          childElementCount: 0
         ▶childNodes: NodeList[1]
         ▶children: HTMLCollection[0]
         ▶classList: DOMTokenList[0]
          className: ""
          clientHeight: 18
          clientLeft: 0
```

Figure 5-12. The p element, expanded

See it? Do you see it? I can barely contain myself. There it is (Figure 5-13).



```
> d3.selectAll("p")
< ▼Selection
   ▼_groups: Array[1]
     ▼0: NodeList[5]
       ▼0: p
          __data__: 5
          accessKey: ""
          align: ""
          assignedSlot: null
         ▶attributes: NamedNodeMap
          baseURI: "http://localhost:8888/chapter_05/04_creating_paragraphs.html"
          childElementCount: 0
         ▶childNodes: NodeList[1]
         ▶children: HTMLCollection[0]
         ▶classList: DOMTokenList[0]
          className: ""
          clientHeight: 18
          clientLeft: 0
```

Figure 5-13. Finally, bound data

Our first data value, the number 5, is showing up under the first paragraph's `__data__` attribute. Expand the other paragraph elements, and you'll see they also contain `__data__` values: 10, 15, 20, and 25, just as we specified.

You see, 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. And the console is where you can go to confirm whether or not your data was bound as expected.

The data is ready. Let's do something with it.

Using Your Data

We can see that the data has been loaded into the page and is bound to our newly created elements in the DOM, but can we *use* it? Here's our code so far:

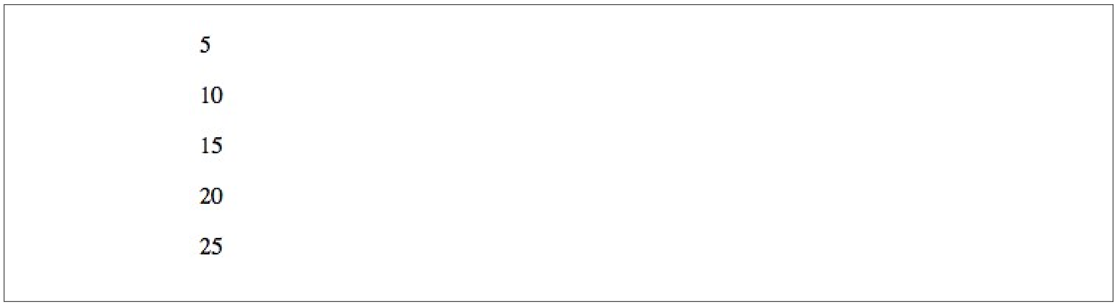
```
var dataset = [ 5, 10, 15, 20, 25 ];
```

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

Let's change the last line to:

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

Now test out that new code in *05_creating_paragraphs_text.html*. You should see the result shown in Figure 5-14.



```
5

10

15

20

25
```

Figure 5-14. More dynamic paragraphs

Whoa! We used our data to populate the contents of each paragraph, all thanks to the magic of the `data()` method. When you chain methods together, anytime after you call `data()`, you can create an anonymous function that accepts `d` as input. The magical `data()` method ensures that `d` is set to the corresponding value in your original dataset, 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 our dataset (or `dataset[2]`). So the third paragraph gets text content of "15".

High-Functioning

In case you're new to writing your own functions (a.k.a. methods), the basic structure of a function definition is:

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

The function we used earlier is dead simple, nothing fancy:

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

This is called an *anonymous function*, because it doesn't have a name. Contrast that with a function that's stored in a variable, which is a *named function*:

```
var doSomething = function() {
    //Code to do something here
};
```

We'll write lots of anonymous functions when using D3. They are the key to accessing individual data values and calculating dynamic properties.

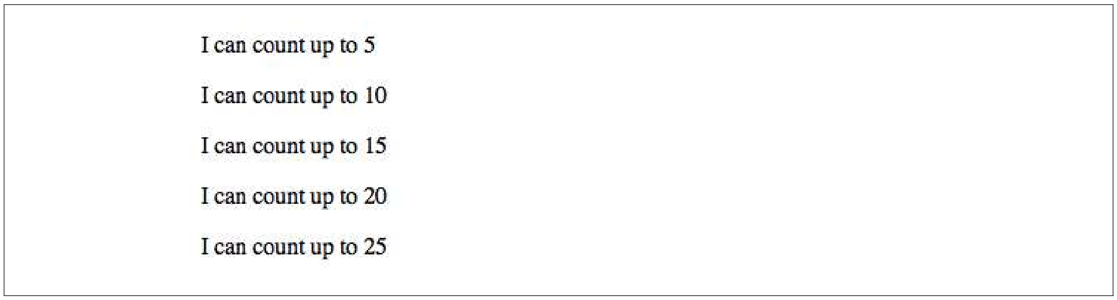
This particular anonymous function is wrapped within D3's `text()` function. So our anonymous function is executed first. Then its result is handed off to `text()`. Then `text()` finally works its magic (by inserting its input argument as text within the selected DOM element):

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

But we can (and will) get much fancier because you can customize these functions any way you like. Yes, this is both the pleasure and pain of writing your own JavaScript. Maybe you'd like to add some extra text, as in:

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

which produces the result shown in Figure 5-15, as seen in example file *06_creating_paragraphs_counting.html*.



```
I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25
```

Figure 5-15. Still more dynamic paragraphs

Data Wants to Be Held

You might be wondering why you have to write out `function(d) { … }` instead of just `d` on its own. For example, this won't work:

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

In this context, without being wrapped `d` in an anonymous function, `d` is undefined. Think of `d` as a lonely little placeholder value that just needs a warm, containing hug from a kind, caring function's parentheses. (Extending this metaphor further, yes, it is creepy that the hug is being given by an *anonymous* function, but that only confuses matters.)

Here is d being held gently and appropriately by a function:

```
.text(function(d) {  // <-- Note tender embrace at left
    return "I can count up to " + d;
});
```

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

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

or the result of a function:

```
.text(someFunction())  // Presumably, someFunction() would return a string
```

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

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

Here, you 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. Here, I've named the argument d just by convention. You could call it datum or info or whatever you like. All D3 is looking for is *any* argument name into which it can pass the current datum. Throughout this book, we'll use d because it is concise and familiar from many of the other D3 examples found online.

In any case, without that function in place, D3 couldn't relay the current data value. Without an anonymous function and its argument there to receive the value of d, D3 could get confused and even start crying. (D3 is more emotional than you'd expect.)

At first, this might seem silly and like a lot of extra work to just get at d, but the value of this approach will become clear as we work on more complex pieces.

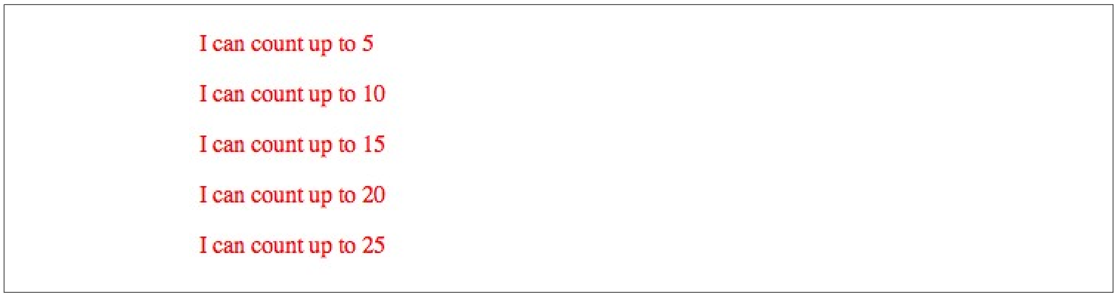
Beyond Text

Things get a lot more interesting when we explore D3's other methods, like attr() and style(), which allow us to set HTML attributes and CSS properties on selections, respectively.

For example, adding one more line to our code:

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

produces the result shown in Figure 5-16, as seen in *07_creating_paragraphs_with_style.html*.



I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

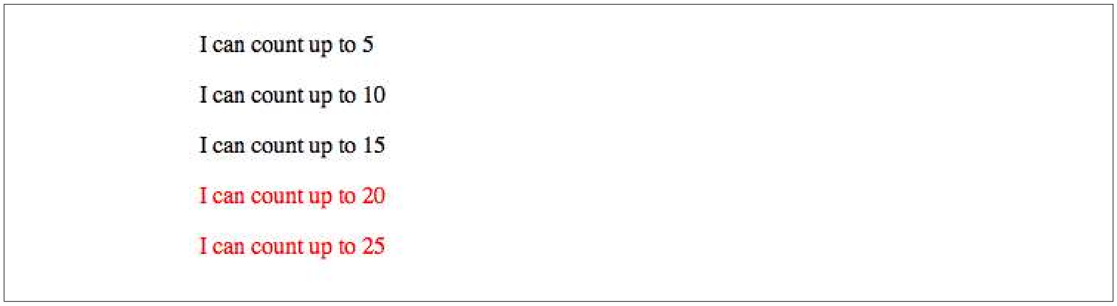
I can count up to 25

Figure 5-16. Red paragraphs

All the text is now red; big deal. But we could use a custom function to make the text red only if the current datum exceeds a certain threshold. So we revise that last line to use a function instead of a string:

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

See the resulting change, displayed in Figure 5-17, in *08_creating_paragraphs_with_style_functions.html*.



I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25

Figure 5-17. Dynamically styled paragraphs

Notice how the first three lines are black, but once `d` exceeds the arbitrary threshold of 15, the text turns red.

Okay, we've got data loaded in, and dynamically created DOM elements bound to that data. I'd say we're ready to start drawing with data!