JavaScript

JavaScript is the scripting language that can make pages dynamic by manipulating the DOM after a page has already loaded in the browser. As I mentioned earlier, getting to know D3 is also a process of getting to know JavaScript. We'll dig in deeper as we go, but here is a taste to get you started.

Hello, Console

Normally, we write JavaScript code (or, a "script") in a text file, and then load that file to the browser in a web page. But you can also just type JavaScript code directly into your browser. This is an easy and quick way to get your feet wet and test out code. We'll also use the JavaScript console for debugging, as it's an essential tool for seeing what's going on with your code.

In Chrome, select View→Developer→JavaScript Console. In Firefox, choose Tools→Web Developer→Web Console. In Safari, go to Develop→Show Error Console (see Figure 3-9).

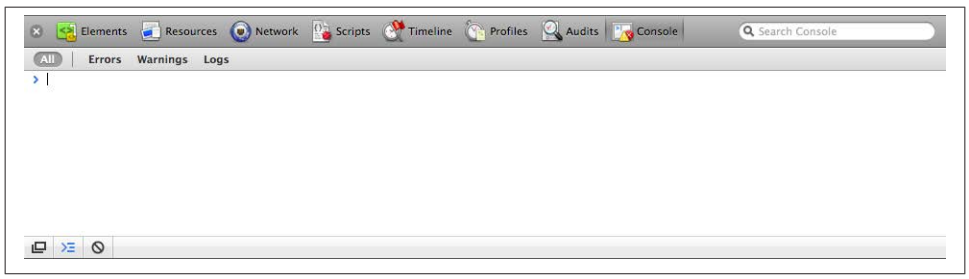

Figure 3-9. A fresh JavaScript console… delicious!

The console accepts one line of code at a time, and it always spits back the result of whatever you input. For example, if you enter the number `7`, the console returns the mind-numbingly obvious result of `7`. Brilliant.

Other times, you'll want your script to print values out to the console automatically, so you can keep an eye on what's going on. As you'll see in some examples that follow, you can use `console.log("something");` to do that.

Type the following examples into the console to see how it works.

Variables

Variables are containers for data. A simple variable holds one value:

```
var number = 5;
```

In that statement, `var` indicates you are declaring a new variable, the name of which is `number`. The equals sign is an *assignment operator* because it takes the value on the right (`5`) and *assigns* it to the variable on the left (`number`). So when you see something such as:

```
defaultColor = "hot pink";
```

try to read the equals sign not as "equals" but as "is set to." So that statement could be stated in plain English as "The variable `defaultColor` *is set to* hot pink."

As you've just seen, variables can store numeric values as well as strings of text, so called because they are formed by stringing together individual characters of text. Strings must be enclosed by quotation marks. True or false (Boolean) values can also be stored:

```
var thisMakesSenseSoFar = true;
```

Also note that in JavaScript, statements are concluded with a semicolon.

You can try making some variables yourself in the console. For example, type `var amount = 200`, then press Enter, then on a new line just enter `amount` and press Enter. You should see `200` returned to you in the console—proof that JavaScript remembered the value you assigned to `amount`!

Other Variable Types

A variable is a datum, the smallest building block of data. The variable is the foundation of all other data structures, which are simply different configurations of variables.

Now we'll address some of these more complex forms of data, including arrays, objects, and arrays of objects. You might want to skip this section for now, but reference it later, once you're ready to load your own data into D3.

Arrays

An array is a sequence of values, conveniently stored in a single variable.

Keeping track of related values in separate variables is inefficient:

```
var numberA = 5;
var numberB = 10;
var numberC = 15;
var numberD = 20;
var numberE = 25;
```

Rewritten as an array, those values are much simpler. Hard brackets `[]` indicate an array, and each value is separated by a comma:

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

Arrays are ubiquitous in data visualization, so you will become very comfortable with them. You can access (retrieve) a value in an array by using *bracket notation*:

```
numbers[2]  //Returns 15
```

The numeral in the bracket refers to a corresponding position in the array. Remember, array positions begin counting at zero, so the first position is 0, the second position is 1, and so on:

```
numbers[0]  //Returns 5
numbers[1]  //Returns 10
numbers[2]  //Returns 15
numbers[3]  //Returns 20
numbers[4]  //Returns 25
```

Some people find it helpful to think of arrays in spatial terms, as though they have rows and columns, like in a spreadsheet:

Position	Value
0	5
1	10
2	15
3	20
4	25

Arrays can contain any type of data, not just integers:

```
var percentages = [ 0.55, 0.32, 0.91 ];
var names = [ "Ernie", "Bert", "Oscar" ];

percentages[1]  //Returns 0.32
names[1]        //Returns "Bert"
```

Although I don't recommend it, different types of values can even be stored within the same array:

```
var mishmash = [ 1, 2, 3, 4.5, 5.6, "oh boy", "say it isn't", true ];
```

Objects

Arrays are great for simple lists of values, but with more complex datasets, you'll want to put your data into an object. For our purposes, think of a JavaScript object as a custom data structure. We use curly brackets {} to indicate an object. In between the brackets, we include *properties* and *values*. A colon `:` separates each property and its value, and a comma separates each property/value pair:

```
var fruit = {
    kind: "grape",
    color: "red",
    quantity: 12,
```

```
    tasty: true
};
```

To reference each value, we use *dot notation*, specifying the name of the property:

```
fruit.kind      //Returns "grape"
fruit.color     //Returns "red"
fruit.quantity  //Returns 12
fruit.tasty     //Returns true
```

Think of the value as "belonging" to the object. Oh, look, some fruit. "What kind of fruit is that?" you might ask. As it turns out, `fruit.kind` is `"grape"`. "Are they tasty?" Oh, definitely, because `fruit.tasty` is `true`.

Objects and Arrays

You can combine these two structures to create arrays of objects, or objects of arrays, or objects of objects or, well, basically whatever structure makes sense for your dataset.

Let's say we have acquired a couple more pieces of fruit, and we want to expand our catalog accordingly. We use hard brackets `[]` on the outside, to indicate an array, followed by curly brackets `{}` and object notation on the inside, with each object separated by a comma:

```
var fruits = [
    {
        kind: "grape",
        color: "red",
        quantity: 12,
        tasty: true
    },
    {
        kind: "kiwi",
        color: "brown",
        quantity: 98,
        tasty: true
    },
    {
        kind: "banana",
        color: "yellow",
        quantity: 0,
        tasty: true
    }
];
```

To access this data, we just follow the trail of properties down to the values we want. Remember, `[]` means array, and `{}` means object. `fruits` is an array, so first we use bracket notation to specify an array index:

```
fruits[1]
```

Next, each array element is an object, so just tack on a dot and a property:

```
fruits[1].quantity   //Returns 98
```

Here's a map of how to access every value in the `fruits` array of objects:

```
fruits[0].kind      ==  "grape"
fruits[0].color     ==  "red"
fruits[0].quantity  ==  12
fruits[0].tasty     ==  true

fruits[1].kind      ==  "kiwi"
fruits[1].color     ==  "brown"
fruits[1].quantity  ==  98
fruits[1].tasty     ==  true

fruits[2].kind      ==  "banana"
fruits[2].color     ==  "yellow"
fruits[2].quantity  ==  0
fruits[2].tasty     ==  true
```

Yes, that's right, we have `fruits[2].quantity` bananas.

JSON

At some point in your D3 career, you will encounter JavaScript Object Notation. You can read up on the details, but JSON is basically a specific syntax for organizing data as JavaScript objects. The syntax is optimized for use with JavaScript (obviously) and AJAX requests, which is why you'll see a lot of web-based application programming interfaces (APIs) that return data formatted as JSON. It's faster and easier to parse with JavaScript than XML, and of course D3 works well with it.

All that, and it doesn't look much weirder than what we've already seen:

```
{
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
}
```

The only difference here is that our property names are now surrounded by double quotation marks `""`, making them string values.

JSON objects, like all other JavaScript objects, can of course be stored in variables like so:

```
var jsonFruit = {
    "kind": "grape",
    "color": "red",
    "quantity": 12,
    "tasty": true
};
```

GeoJSON

Just as JSON is just a formalization of existing JavaScript object syntax, GeoJSON is a formalized syntax of JSON objects, optimized for storing geodata. All GeoJSON object are JSON objects, and all JSON objects are JavaScript objects.

GeoJSON can store points in geographical space (typically as longitude/latitude coordinates), but also shapes (such as lines and polygons) and other spatial features. If you have a lot of geodata, it's worth it to parse it into GeoJSON format for best use with D3.

We'll get into the details of GeoJSON when we talk about geomaps, but for now, just know that this is what simple GeoJSON data could look like the following:

```
{
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [ 150.1282427, -24.471803 ]
            },
            "properties": {
                "type": "town"
            }
        }
    ]
}
```

Confusingly, longitude is always listed before latitude. Get used to thinking in terms of lon/lat instead of lat/lon.

Mathematical Operators

You can add, subtract, multiply, and divide values using the following operators, respectively:

```
+   //Add
-   //Subtract
*   //Multiply
/   //Divide
```

For example, type `8 * 2` into the console, press Enter, and you should see `16`. More examples:

```
1 + 2       //Returns 3
10 - 0.5    //Retuns 9.5
33 * 3      //Returns 99
3 / 4       //Returns 0.75
```

Comparison Operators

You can compare values against each other using the following operators:

```
==  //Equal to
!=  //Not equal to
<   //Less than
>   //Greater than
<=  //Less than or equal to
>=  //Greater than or equal to
```

These all compare some value on the left to some value on the right. If the result is true, then `true` is returned. If the result is false, `false` is returned.

Try it! Type each of the following into the console and see what you get:

```
3 == 3
3 == 5
3 >= 3
3 >= 2
100 < 2
298 != 298
```

(JavaScript also offers the `===` and `!==` operators, which perform equality comparisons without type coercion, but I won't be addressing that distinction in this book.)

Control Structures

Whenever your code needs to make a decision or repeat something, you need a control structure. There are lots to choose from, but we are primarily interested in `if` statements and `for` loops.

if() only

An `if` statement uses comparison operators to determine if a statement is true or false:

```
if (test) {
    //Code to run if true
}
```

If the test between parentheses is `true`, then the code between the curly brackets is run. If the test turns up `false`, then the bracketed code is ignored, and life goes on. (Technically, life goes on either way.)

```
if (3 < 5) {
    console.log("Eureka! Three is less than five!");
}
```

In the preceding example, the bracketed code will always be executed, because `3 < 5` is always true. `if` statements are more useful when comparing variables or other conditions that change.

```
if (someValue < anotherValue) {
    //Set someValue to anotherValue, thereby restoring
    //a sense of balance and equilibrium to the world.
    someValue = anotherValue;
}
```

for() now

You can use `for` loops to repeatedly execute the same code, with slight variations. A `for` loop uses this syntax:

```
for (initialization; test; update) {
    //Code to run each time through the loop
}
```

They are so-called because they loop through the code *for* as many times as specified. First, the initialization statement is run. Then, the test is evaluated, like a mini `if` statement. If the test is true, then the bracketed code is run. Finally, the `update` statement is run, and the test is reevaluated.

The most common application of a `for` loop is to increase some variable by 1 each time through the loop. The test statement can then control how many times the loop is run by referencing that value. (The variable is often named `i`, purely by convention, because it is short and easy to type.)

```
for (var i = 0; i < 5; i++) {
    console.log(i);  //Prints value to console
}
```

The preceding `for` loop prints the following to the console:

```
0
1
2
3
4
```

The first time through, a variable named `i` is declared and set to zero. `i` is less than five, so the bracketed code is run, printing the current value of `i` (zero) to the console. Finally, the value of `i` is increased by one. (`i++` is shorthand for `i = i + 1`.) So now `i` is `1`, so the test again evaluates as true, the bracketed code is run again, but this time the value printed to the console is 1.

As you can see, reading prose descriptions of loops is about as interesting as executing them by hand yourself. This is why we invented computers. So I'll skip to the end and point out that after the final iteration, `i` is increased to 5, after which the test returns false, and the loop is over.

It's important to note that counting began at zero, and not one. That is, the "first" value of `i` was `0`. Why not `1`? This is another arbitrary convention, but it nicely parallels how computers count arrays of values.

What arrays are made for()

Code-based data visualization would not be possible without arrays and the mighty `for()` loop. Together, they form a data geek's dynamic duo. (If you do not consider yourself a "data geek," then may I remind you that you are reading a book titled *Interactive Data Visualization for the Web*?)

An array organizes lots of data values in one convenient place. Then `for()` can quickly "loop" through every value in an array and perform some action with it—such as, express the value as a visual form. D3 often manages this looping for us, such as with its magical `data()` method.

Note this example, which loops through each of the values in an array called `numbers`:

```
var numbers = [ 8, 100, 22, 98, 99, 45 ];

for (var i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);  //Print value to console
}
```

See that `numbers.length`? That's the beautiful part. `length` is a property of every array. In this case, `numbers` contains six values, so `numbers.length` resolves to `6`, and the loop runs six times. If `numbers` were 10 positions long, the loop would run 10 times. If it were 10 million positions long … yeah, you get it. This is what computers are good at: taking a set of instructions and executing them over and over. And this is at the heart of why data visualization can be so rewarding—you design and code the visualization system, and the system will respond appropriately, even as you feed it different data. The system's mapping rules are consistent, even when the data is not.

Functions

Functions are chunks of code that do things.

More specifically, functions are special because they can take arguments or parameters as input, and then return values as output. Parentheses are used to *call* (execute) a function. If that function requires any arguments (input values), then they are *passed* to the function by including them in the parentheses.

Whenever you see something like the following code, you know it's a function:

```
calculateGratuity(38);
```

In fact, you've already seen functions at work with `console.log`, as in the following:

```
console.log("Look at me. I can do stuff!");
```

But the best part about functions is that you can define your own. There are several ways to define functions in JavaScript, but here's the simplest, which I'll use throughout this book:

```
var calculateGratuity = function(bill) {
    return bill * 0.2;
};
```

This declares a new variable named `calculateGratuity`. Then, instead of assigning a simple number or string, we store an entire function in the variable. In the parentheses, we name `bill`, another variable to be used only by the function itself. `bill` is the expected input. When called, the function will take that input, multiply it by `0.2`, and *return* the result as its output.

So now if we call:

```
calculateGratuity(38);
```

the function returns 7.6. Not bad—a 20 percent tip!

Of course, you could store the output to use it elsewhere later, as in:

```
var tip = calculateGratuity(38);
console.log(tip);  //Prints 7.6 to the console
```

There are also *anonymous* functions that, unlike `calculateGratuity`, don't have names. Anonymous functions are used all the time with D3, but I'll introduce them later.

Comments

```
/* JavaScript supports CSS-style comments like this. */

// But double-slashes can be used as well.
// Anything following // on the same line will be ignored.
// This is helpful for including brief notes to yourself
// as to what each line of code does, as in:

console.log("Brilliant");  //Prints "Brilliant" to the console
```

Referencing Scripts

Scripts can be included directly in HTML, between two `script` tags, as in:

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

or stored in a separate file with a *.js* suffix, and then referenced somewhere in the HTML (could be in the `head`, as shown here, or also just before the end of the closing `body` tag):

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

JavaScript Gotchas

As a bonus, and at no extra charge, I would like to share with you my top four JavaScript gotchas: things that, had I known earlier, would have saved me many hours of late-night debugging sessions, anxiety-induced panic, and increased cortisol levels. You might want to come back and reference this section later.

Dynamic typing

I do love how your fingers flutter across the keyboard, but that's not what I'm talking about. No, JavaScript is a *loosely typed* language, meaning you don't have to specify what *type* of information will be stored in a variable in advance. Many other languages, like Java (which is completely different from Java*Script*), require you to declare a variable's type, such as `int`, `float`, `boolean`, or `String`:

```
//Declaring variables in Java
int number = 5;
float value = 12.3467;
boolean active = true;
String text = "Crystal clear";
```

JavaScript, however, automatically *types* a variable based on what kind of information you assign to it. (Note that `''` or `""` indicate string values. I prefer double quotation marks `""`, but some people like singles `''`.)

```
//Declaring variables in JavaScript
var number = 5;
var value = 12.3467;
var active = true;
var text = "Crystal clear";
```

How boring—`var`, `var`, `var`, `var`!—yet handy, as we can declare and name variables before we even know what type of data will go into them. You can even change a variable's type on-the-fly without JavaScript freaking out on you:

```
var value = 100;
value = 99.9999;
value = false;
value = "This can't possibly work.";
value = "Argh, it does work! No errorzzzz!";
```

I mention this because one day a numeric value will be accidentally stored as a string, and it will cause some part of your code to behave strangely, and basically I don't want you to come crying to me when that happens. Thus, know that whenever a variable's type is in doubt, you can employ the `typeof` operator: