CHAPTER 8

Axes

Having mastered the use of D3 scales, we now have the scatterplot shown in Figure 8-1, using the code from Chapter 7's example *08_scaled_plot_sqrt_scale.html*.

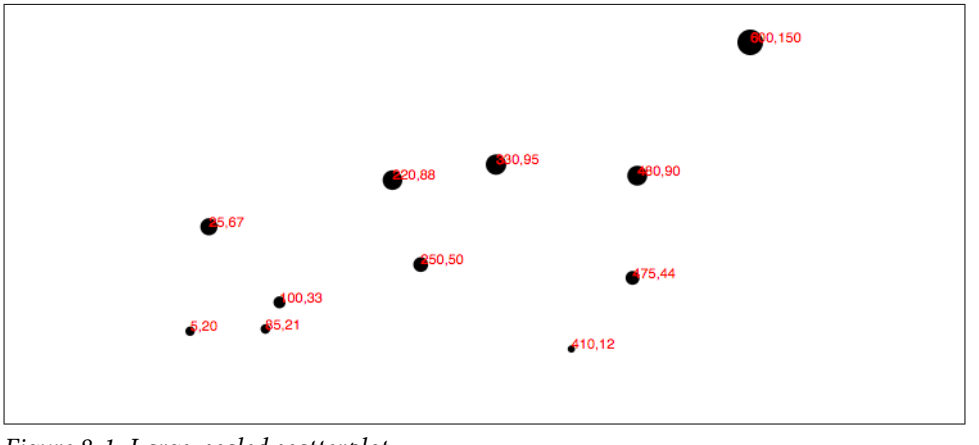

Figure 8-1. Large, scaled scatterplot

Let's add horizontal and vertical axes, so we can do away with the horrible red numbers cluttering up our chart.

Introducing Axes

Much like its scales, D3's *axes* are actually *functions* whose parameters you define. Unlike scales, when an axis function is called, it doesn't return a value, but generates the visual elements of the axis, including lines, labels, and ticks.

Note that the axis functions are SVG-specific, as they generate SVG elements. Also, axes are intended for use with quantitative scales (that is, scales that use numeric values, as opposed to ordinal, categorical ones).

Setting Up an Axis

There are four different axis function constructors, each one corresponding to a different orientation and placement of labels: `d3.axisTop`, `d3.axisBottom`, `d3.axisLeft`, and `d3.axisRight`. For vertical axes, use `d3.axisLeft` or `d3.axisRight`, with ticks and labels appearing to the left and right, respectively. For horizontal axes, use `d3.axisTop` or `d3.axisBottom`, with ticks and labels appearing above and below, respectively.

We'll start by using `d3.axisBottom()` to create a generic axis function:

```
var xAxis = d3.axisBottom();
```

At a minimum, each axis also needs to be told on what *scale* to operate. Here we'll pass in the `xScale` from the scatterplot code:

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

We could be more concise and write this in one line:

```
var xAxis = d3.axisBottom()
              .scale(xScale);
```

In fact, you could be even more concise by just passing the name of the scale into the axis constructor directly. This is exactly equivalent to the preceding statement:

```
var xAxis = d3.axisBottom(xScale);
```

I've chosen to call `scale()` explicitly, in the hope that this will make my code more human-readable.

Finally, to actually generate the axis and insert all those little lines and labels into our SVG, we must *call* the `xAxis` function. This is similar to the scale functions, which we first configured by setting parameters, and then later *called*, to put them into action.

I'll put this code at the end of our script, so the axis is generated after the other elements in the SVG, and therefore appears "on top":

```
svg.append("g")
    .call(xAxis);
```

This is where things get a little funky. You might be wondering why this looks so different from our friendly scale functions. Here's why: because an *axis* function actually draws something to the screen (by appending SVG elements to the DOM), we need to specify *where* in the DOM it should place those new elements. This is in contrast to

scale functions like `xScale()`, for example, which calculate a value and return those values, typically for use by yet another function, without impacting the DOM at all.

So what we're doing with the preceding code is to first reference `svg`, the SVG image in the DOM. Then, we `append()` a new `g` element to the end of the SVG. In SVG land, a `g` element is a *group* element. Group elements are invisible, unlike `line`, `rect`, and `circle`, and they have no visual presence themselves. Yet they help us in two ways: first, `g` elements can be used to contain (or "group") other elements, which keeps our code nice and tidy. Second, we can apply *transformations* to `g` elements, which affects how visual elements within that group (such as `lines`, `rects`, and `circles`) are rendered. We'll get to transformations in just a minute.

So we've created a new `g`, and then finally, the function `call()` is called on our new `g`. So what is `call()`, and who is it calling?

D3's `call()` function takes the incoming *selection*, as received from the prior link in the chain, and hands that selection off to any *function*. In this case, the selection is our new `g` group element. Although the `g` isn't strictly necessary, we are using it because the axis function is about to generate lots of crazy lines and numbers, and it's nice to contain all those elements within a single group object. `call()` hands off `g` to the `xAxis` function, so our axis is generated *within* `g`.

If we were messy people who loved messy code, we could also rewrite the preceding snippet as this exact equivalent:

```
svg.append("g")
    .call(d3.axisBottom()
    .scale(xScale));
```

See, you could cram the whole axis function within `call()`, but it's usually easier on our brains to define functions first, then call them later.

In any case, Figure 8-2 shows what that looks like. See code example *01_axes.html*.

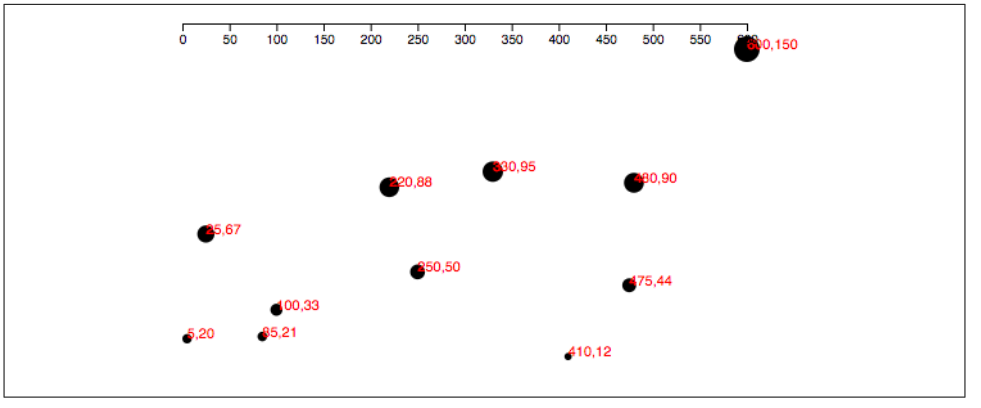

Figure 8-2. Simple axis, but in the wrong place

This isn't required, but I'd also recommend assigning a class of `axis` to the new `g` element. As your project grows in complexity, you'll find that naming `g` elements in this way makes your DOM easier to inspect and troubleshoot.

```
svg.append("g")
    .attr("class", "axis") //Assign "axis" class
    .call(xAxis);
```

Positioning Axes

By default, an axis is positioned using the range values of the specified scale. In our case, `xAxis` is referencing `xScale`, which has a range of `[20, 460]`, because we applied 20 pixels of padding on all edges of the SVG. So the left edge of our axis appears at an `x` of 20, and the right edge at an `x` of 460.

That's nice, as we want our axis to line up with the chart's visual marks. (Graphical honesty, FTW!) But we'll need to reposition the axis *vertically*, as, by convention, a bottom-oriented axis should appear at the bottom of the chart.

This is where SVG *transformations* come in. By adding one line of code, we can `transform` the entire axis group, pushing it to the bottom:

```
svg.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(0," + (h - padding) + ")")
    .call(xAxis);
```

Note that we use `attr()` to apply `transform` as an attribute of `g`. SVG transforms are quite powerful, and can accept several different kinds of transform definitions, including scales and rotations. But we are keeping it simple here with only a *translation* transform, which simply pushes the whole `g` group over and down by some amount.

Translation transforms are specified with the easy syntax of `translate(x,y)`, where `x` and `y` are, obviously, the number of horizontal and vertical pixels by which to translate the element. So, in the end, we would like our `g` to look like this in the DOM:

```
<g class="axis" transform="translate(0,280)">
```

As you can see, the `g.axis` isn't moved horizontally at all, but it is pushed 280 pixels down, conveniently to the base of our chart. (D3 also automatically generates `fill`, `font-size`, `font-family`, and `text-anchor` attributes, which I've omitted above for clarity.) We specify the downward translation in this line of code:

```
.attr("transform", "translate(0," + (h - padding) + ")")
```

Note the use of `(h - padding)`, so the group's top edge is set to `h`, the height of the entire image, minus the `padding` value we created earlier. `(h - padding)` is calculated to be `280`, and then connected to the rest of the string, so the final transform property value is `translate(0,280)`.

The result in Figure 8-3 is much better! Check out the code so far in *02_axes_bottom.html*.

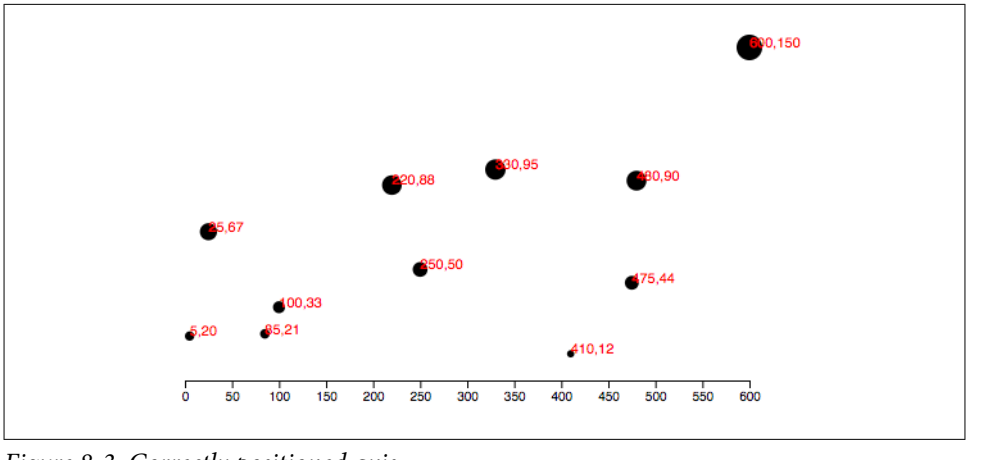

Figure 8-3. Correctly positioned axis

Using CSS to Style Axis Elements

Assigning your axis a class of `axis` makes it easy to override D3's default styles using the simple CSS selector `.axis`. The axes themselves are made up of `path`, `line`, and `text` elements, so those are the three elements to target in your CSS. The `path`s and `line`s can be styled together, with the same rules, and `text` gets its own rules around font and font size.

For example, we could introduce our first CSS styles, up in the `<head>` of our page:

```
.axis path,
.axis line {
    stroke: teal;
    shape-rendering: crispEdges;
}

.axis text {
    font-family: Optima, Futura, sans-serif;
    font-weight: bold;
    font-size: 14px;
    fill: teal;
}
```

These CSS rules will override D3's default styles, resulting in the admittedly not beautiful example in Figure 8-4.

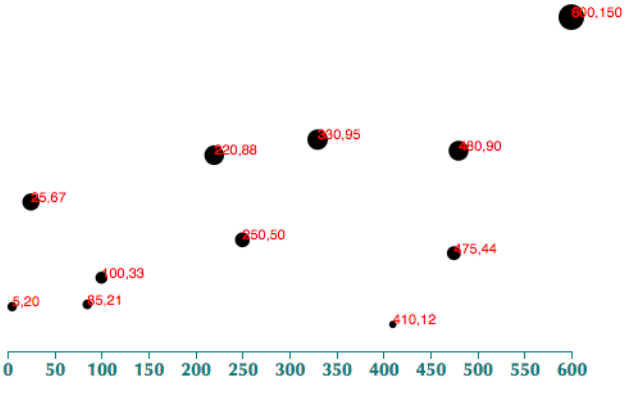

Figure 8-4. Axis with styles overridden with CSS

Note that when we use CSS rules to style SVG elements, only SVG attribute names—not regular CSS properties—should be used. This is confusing, because many properties share the same names in both CSS and SVG, but some do not. For example, in regular CSS, to set the color of some text, you would use the `color` property, as in:

```
p {
    color: olive;
}
```

That will set the text color of all `p` paragraphs to be `olive`. But try to apply this property to an SVG element, as with:

```
text {
    color: olive;
}
```

and it will have no effect because `color` is not a property recognized by SVG. Instead, you must use SVG's equivalent, `fill`:

```
text {
    fill: olive;
}
```

In my example CSS above, I've used `stroke`, `fill`, and `shape-rendering`, all of which are unique to SVG. (The `shape-rendering` property can be used to clean up visual artifacts from antialiasing, for you designers who require super-clean lines. No blurry axes for us!)

If you ever find yourself trying to style SVG elements, but for some reason the stupid CSS code just isn't working (*Grrr!*), I suggest you take a deep breath, pause, and then review your *property names* very closely to ensure you're using SVG names, not CSS ones. (You can reference the complete SVG attribute list on the MDN site.)

Check for Ticks

Some ticks spread disease, but D3's ticks communicate information. Yet more ticks are not necessarily better, and at a certain point, they begin to clutter your chart. You'll notice that we never specified how many ticks to include on the axis, nor at what intervals they should appear. Without clear instruction, D3 has automagically examined our scale `xScale` and made informed judgments about how many ticks to include, and at what intervals (every 50, in this case).

As you would expect, you can customize all aspects of your axes, starting with the rough number of ticks, using `ticks()`:

```
var xAxis = d3.axisBottom()
              .scale(xScale)
              .ticks(5);  //Set rough # of ticks
```

See *03_axes_clean.html* for that code.

You'll notice in Figure 8-5 that, although we specified only five ticks, D3 has made an executive decision and ordered up a total of seven. That's because D3 has got your back, and figured out that including only *five* ticks would require slicing the input domain into less-than-gorgeous values—in this case, 0, 150, 300, 450, and 600. D3 interprets the `ticks()` value as merely a suggestion and will override your suggestion with what it determines to be the most clean and human-readable values—in this case, intervals of 100—even when that requires including slightly more or fewer ticks than you requested. This is actually a totally brilliant feature that increases the scalability of your design; as the dataset changes and the input domain expands or contracts (bigger numbers or smaller numbers), D3 ensures that the tick labels remain easy to read.

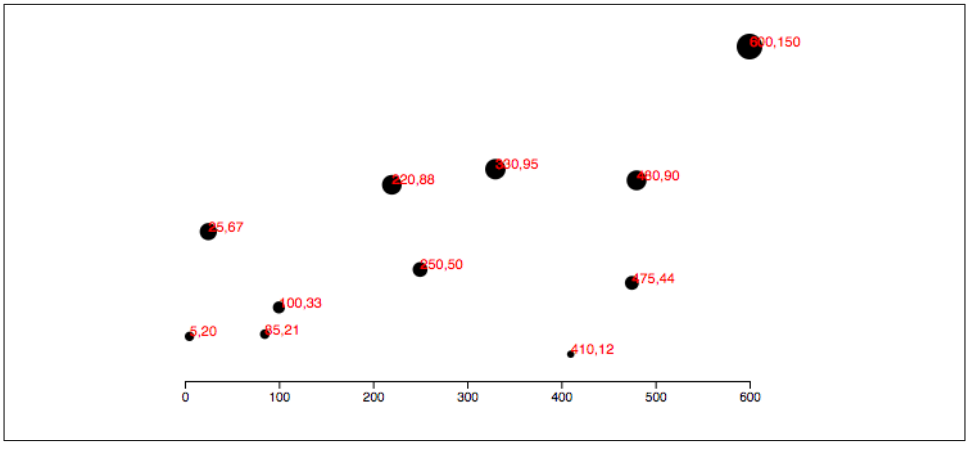

Figure 8-5. Fewer ticks

Specifying Tick Values Manually

For more control, you can specify tick values manually by calling `tickValues()` instead of `ticks()`, and passing in an array of whatever values you'd like labeled. This overrides D3's default tick-selection logic. (Sometimes humans know best.) For example, we could modify the earlier example:

```
var xAxis = d3.axisBottom()
              .scale(xScale)
              .tickValues([0, 100, 250, 600]);
```

Note the results in Figure 8-6.

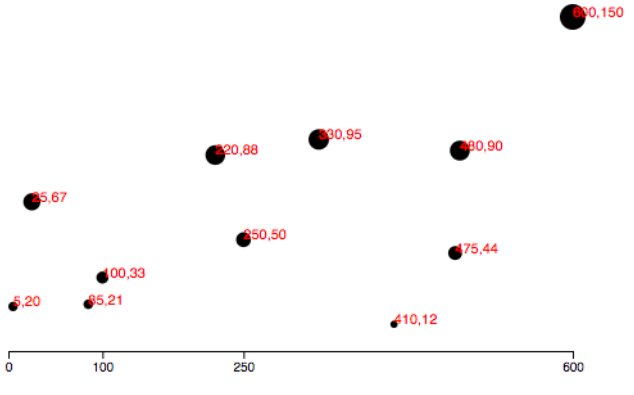

Figure 8-6. Manually specified tick values

Y Not?

Time to label the vertical axis! By copying and tweaking the code we already wrote for the `xAxis`, we add this near the top of our code:

```
//Define Y axis
var yAxis = d3.axisLeft()
              .scale(yScale)
              .ticks(5);
```

and this, near the bottom:

```
//Create Y axis
svg.append("g")
   .attr("class", "axis")
   .attr("transform", "translate(" + padding + ",0)")
   .call(yAxis);
```

Note in Figure 8-7 that the axis is oriented vertically, the labels are placed to the left of the axis, and the `yAxis` group `g` is translated to the right by the amount `padding`.

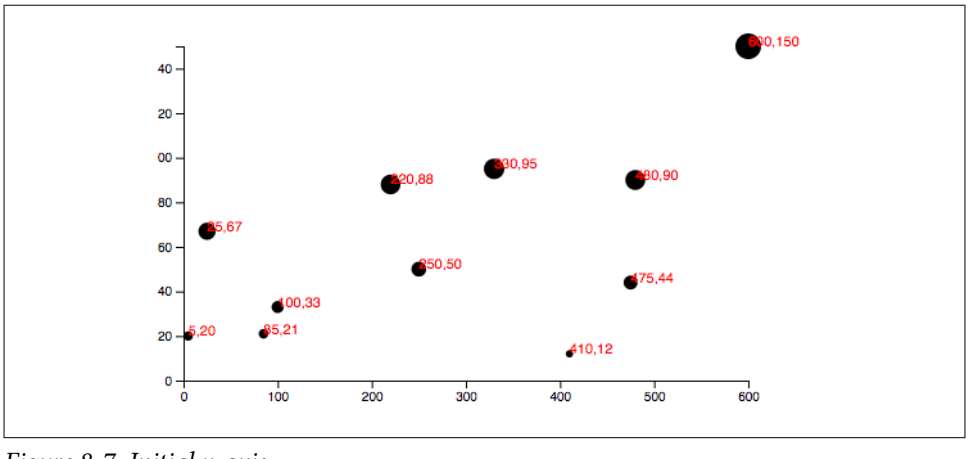

Figure 8-7. Initial y-axis

This is starting to look like a real chart! But the `yAxis` labels are getting cut off. To give them more room on the left side, I'll bump up the value of `padding` from 20 to 30:

```
var padding = 30;
```

Of course, you could also introduce separate `padding` variables for each axis, say `xPadding` and `yPadding`, for more control over the layout.

See the updated code in *04_axes_y.html*. It looks like Figure 8-8.

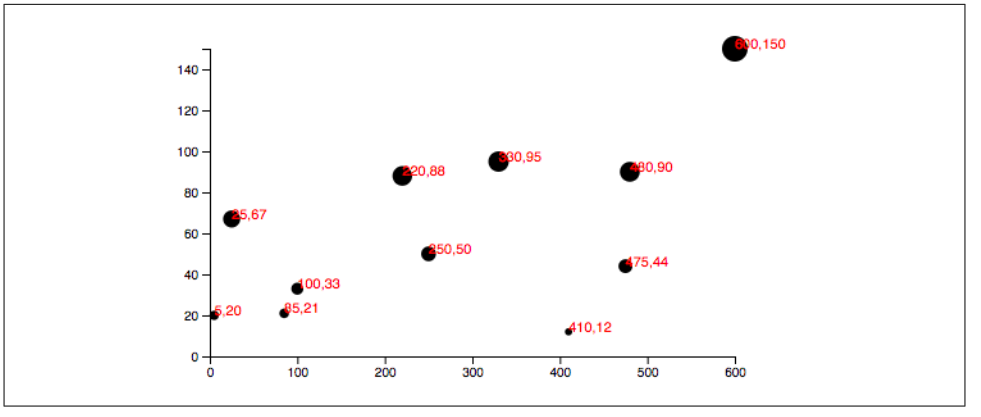

Figure 8-8. Scatterplot with y-axis

Final Touches

I appreciate that so far you have been very quiet and polite, and not at all confrontational. Yet I still feel as though I have to win you over. So to prove to you that our new axes are dynamic and scalable, I'd like to switch from using a static dataset to using randomized numbers:

```
//Dynamic, random dataset
var dataset = [];
var numDataPoints = 50;
var xRange = Math.random() * 1000;
var yRange = Math.random() * 1000;
for (var i = 0; i < numDataPoints; i++) {
    var newNumber1 = Math.floor(Math.random() * xRange);
    var newNumber2 = Math.floor(Math.random() * yRange);
    dataset.push([newNumber1, newNumber2]);
}
```

This code initializes an empty array, then loops through 50 times, chooses two random numbers each time, and adds ("pushes") that pair of values to the `dataset` array (see Figure 8-9).

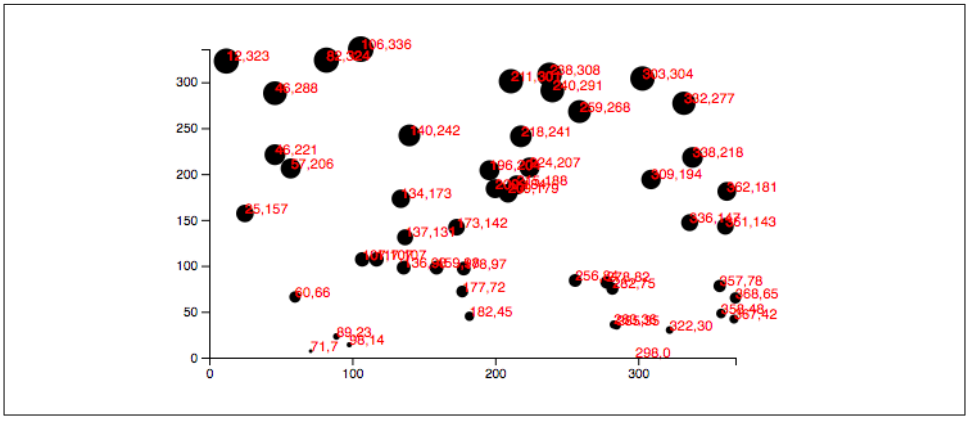

Figure 8-9. Scatterplot with random data

Try out that randomized dataset code in *05_axes_random.html*. Each time you reload the page, you'll get different data values. Notice how both axes scale to fit the new domains, and ticks and label values are chosen accordingly.

Having made my point, I think we can finally cut those horrible red labels, by commenting out the relevant lines of code.

The result is shown in Figure 8-10. Our final scatterplot code lives in *06_axes_no_labels.html*.

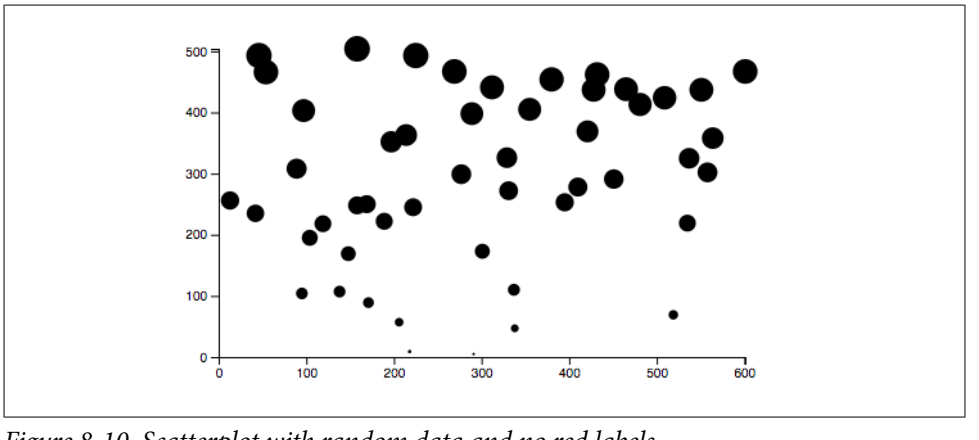

Figure 8-10. Scatterplot with random data and no red labels

Formatting Tick Labels

One last thing: so far, we've been working with integers—whole numbers—which are nice and easy. But data is often messier, and in those cases, you might want more control over how the axis labels are formatted. Enter `tickFormat()`, which enables you to

specify how your numbers should be formatted. For example, you might want to include three places after the decimal point, or display values as percentages, or both.

To use `tickFormat()`, first define a new number-formatting function. This one, for example, says to treat values as percentages with one decimal point precision. That is, if you give this function the number `0.23`, it will return the string `"23.0%"`. (See the reference entry for `d3.format()` for more options.)

```
var formatAsPercentage = d3.format(".1%");
```

Then, tell your axis to use that formatting function for its ticks, for example:

```
xAxis.tickFormat(formatAsPercentage);
```

Testing Formatting Functions the Easy Way

I find it easiest to test these formatting functions out in the JavaScript console. For example, just open any page that loads D3, such as *06_axes_no_labels.html*, and type your format rule into the console. Then test it by feeding it a value, as you would with any other function.

You can see in Figure 8-11 that a data value of `0.54321` is converted to `54.3%` for display purposes—perfect!

Test out the following statements in the console and note the results:

- `formatAsPercentage(.365)`
- `formatAsPercentage(1.2)`
- `formatAsPercentage(-.5)`

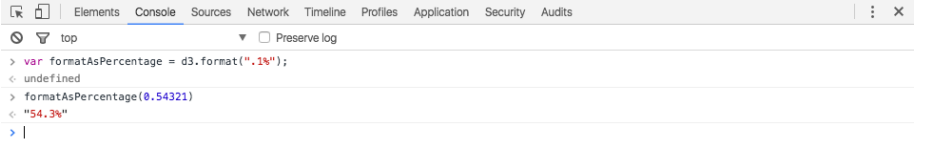

Figure 8-11. Testing d3.format() in the console

You can play with that code in *07_axes_format.html*. Obviously, a percentage format doesn't make sense with our scatterplot's current dataset, but as an exercise, you could try tweaking how the random numbers are generated, to make more appropriate, nonwhole number values, or just experiment with the format function itself. (Also try adjusting the padding, so the labels on the left side are fully visible.)

Time-Based Axes

How hard is it to make time-based axes?

Not hard.

Let's revisit Chapter 7's example, *09_time_scale.html*. I've created a new example, *08_time_axis.html*, into which I've copied and pasted the code where we define both axis generators and call them. With *no other changes*, we see the result shown in Figure 8-12.

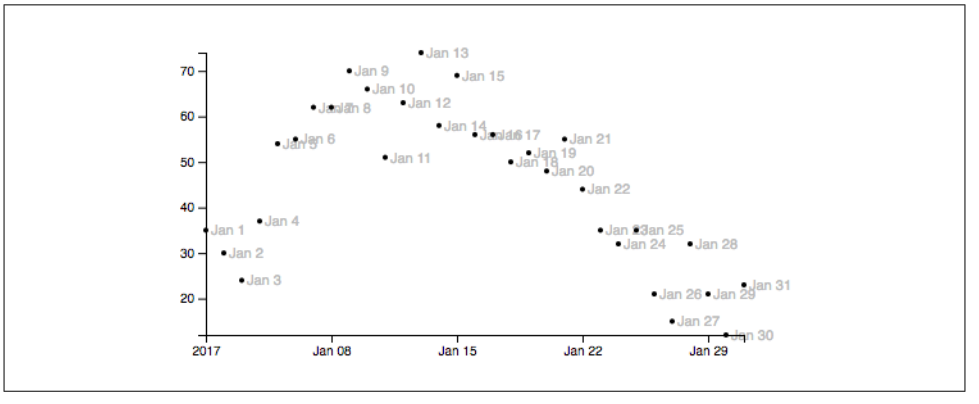

Figure 8-12. Easy time-based axis

Remember how we have to tell each axis generator which scale to reference? In this case, all the hard work was already done, when we set up the time scale (and parsed the incoming strings into dates). Once the scale is in place, all the axis has to do is follow that scale's lead.

In *09_time_axis_prettier.html*, I've cleaned this chart up a bit by removing the value labels, adjusting the axis ticks, expanding the x-axis's domain by a day in either direction (effectively from December 31 to February 1), and adding light gray guide lines to better illustrate how the circles are being positioned. See the result in Figure 8-13.

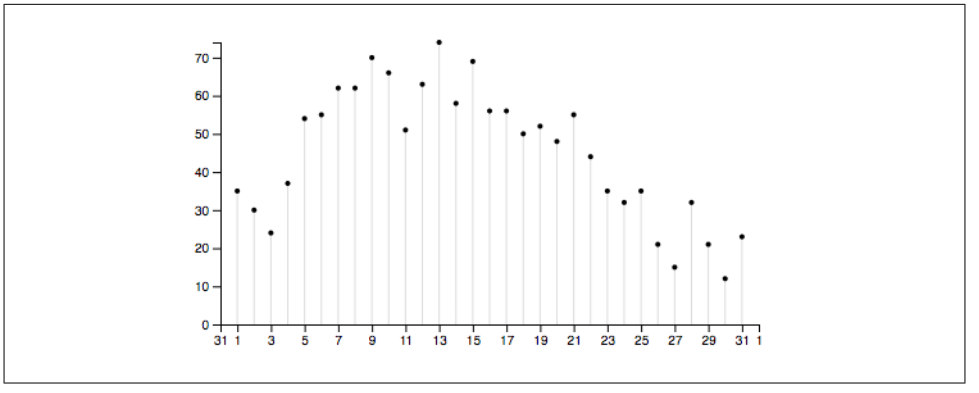

Figure 8-13. Time series, cleaned up