CHAPTER 7
Scales

"Scales are functions that map from an input domain to an output range."

That's Mike Bostock's definition of D3 scales from an earlier version of the D3 docs, and there is no clearer way to say it.

The values in any dataset are unlikely to correspond exactly to pixel measurements for use in your visualization. Scales provide a convenient way to map those data values to new values useful for visualization purposes.

D3 scales are *functions* with parameters that you define. Once they are created, you call the `scale` function and pass it a data value, and it nicely returns a scaled output value. You can define and use as many scales as you like.

It might be tempting to think of a scale as something that appears visually in the final image—like a set of tick marks, indicating a progression of values. *Do not be fooled!* Those tick marks are part of an *axis*, which is a *visual representation* of a scale. A scale is a mathematical relationship, with no direct visual output. I encourage you to think of scales and axes as two different, yet related, elements.

This chapter addresses primarily *linear* scales, because they are most common and easiest understood. Once you understand linear scales, the others—ordinal, logarithmic, square root, and so on—will be a piece of cake.

Apples and Pixels

Imagine that the following dataset represents the number of apples sold at a roadside fruit stand each month:

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

First of all, this is great news, as the stand is selling 100 additional apples each month! Business is booming. To showcase this success, you want to make a bar chart illustrating the steep upward climb of apple sales, with each data value corresponding to the height of one bar.

Until now, we've used data values directly as display values, ignoring unit differences. So if 500 apples were sold, the corresponding bar would be 500 pixels tall.

That could work, but what about next month, when 600 apples are sold? And a year later, when 1,800 apples are sold? Your audience would have to purchase ever-larger displays just to be able to see the full height of those very tall apple bars! (Mmm, apple bars!)

This is where scales come in. Because apples are not pixels (which are also not oranges), we need scales to translate between them.

Domains and Ranges

A scale's *input domain* is the range of possible input data values. Given the preceding apple data, appropriate input domains would be either 100 and 500 (the minimum and maximum values of the dataset) or 0 and 500.

A scale's *output range* is the range of possible output values, commonly used as display values in pixel units. The output range is completely up to you, as the information designer. If you decide the shortest apple bar will be 10 pixels tall, and the tallest will be 350 pixels tall, then you could set an output range of 10 and 350.

For example, create a scale with an input domain of `[100,500]` and an output range of `[10,350]`. If you handed the low input value of `100` to that scale, it would return its lowest range value, `10`. If you gave it `500`, it would spit back `350`. If you gave it `300`, it would hand `180` back to you on a silver platter. (`300` is in the center of the domain, and `180` is in the center of the range.)

We can visualize the domain and range as corresponding axes, side-by-side, displayed in Figure 7-1.

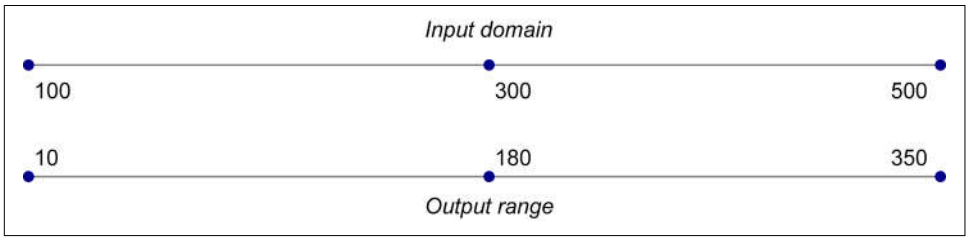

Figure 7-1. An input domain and an output range, visualized as parallel axes

One more thing: to prevent your brain from mixing up the *input domain* and *output range* terminology, I'd like to propose a little exercise. When I say "input," you say "domain." Then I say "output," and you say "range." Ready? Okay:

> Input! *Domain!*
>
> Output! *Range!*
>
> Input! *Domain!*
>
> Output! *Range!*

Got it? Great.

Normalization

If you're familiar with the concept of *normalization*, it might be helpful to know that, with a linear scale, that's all that is really going on here.

Normalization is the process of mapping a numeric value to a new value between 0 and 1, based on the possible minimum and maximum values. For example, with 365 days in the year, day number 310 maps to about 0.85, or 85% of the way through the year.

With linear scales, we are just letting D3 handle the math of the normalization process. The input value is normalized according to the domain, and then the normalized value is scaled to the output range.

Creating a Scale

D3's linear scale function generator is accessed with `d3.scaleLinear()`. I recommend opening the sample code page *01_scale_test.html* and typing each of the following into the console:

```
var scale = d3.scaleLinear();
```

Congratulations! Now `scale` is a function to which you can pass input values. (Don't be misled by the `var`. Remember that in JavaScript, variables can store functions.)

```
scale(2.5);  //Returns 2.5
```

Because we haven't set a domain and a range yet, this function will map input to output on a 1:1 scale. That is, whatever we input will be returned unchanged.

We can set the scale's input domain to `100,500` by passing those values to the `domain()` method as an array. Note the hard brackets indicating an array:

```
scale.domain([100, 500]);
```

Set the output range in similar fashion, with `range()`:

```
scale.range([10, 350]);
```

These steps can be done separately, as just shown, or chained together into one line of code:

```
var scale = d3.scaleLinear()
              .domain([100, 500])
              .range([10, 350]);
```

Either way, our scale is ready to use!

```
scale(100);  //Returns 10
scale(300);  //Returns 180
scale(500);  //Returns 350
```

Typically, you will call scale functions from within an `attr()` method or similar, not on their own. Let's modify our scatterplot visualization to use dynamic scales.

Scaling the Scatterplot

To revisit our dataset from the scatterplot:

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

You'll recall that this `dataset` is an array of arrays. We mapped the first value in each array onto the x-axis, and the second value onto the y-axis. Let's start with the x-axis.

Just by eyeballing the x values, it looks like they range from 5 to 480, so a reasonable input domain to specify might be `0,500`, right?

Why are you giving me that look? Oh, because you want to keep your code flexible and scalable, so it will continue to work even if the data changes in the future. Very smart! Remember, if we were building a data dashboard for the roadside apple stand, we'd want our code to accommodate the enormous projected growth in apple sales. Our chart should work just as well with 5 apples sold as 5 million.

d3.min() and d3.max()

Instead of specifying fixed values for the domain, we can use the convenient array functions `d3.min()` and `d3.max()` to analyze our dataset on the fly. For example, this loops through each of the x values in our arrays and returns the value of the greatest one:

```
d3.max(dataset, function(d) {
    return d[0];  //References first value in each subarray
});
```

That code will return the value 480, because 480 is the largest x value in our dataset. Let me explain how it works.

Both `min()` and `max()` work the same way, and they can take either one or two arguments. The first argument must be a reference to the array of values you want evaluated, which is `dataset`, in this case. If you have a simple, one-dimensional array of numeric values, like `[7, 8, 4, 5, 2]`, then it's obvious how to compare the values against each other, and no second argument is needed. For example:

```
var simpleDataset = [7, 8, 4, 5, 2];
d3.max(simpleDataset);  // Returns 8
```

The `max()` function simply loops through each value in the array, and identifies the largest one.

But our `dataset` is not just an array of numbers; it is an array of arrays. Calling `d3.max(dataset)` might produce unexpected results:

```
var dataset = [
                [5, 20], [480, 90], [250, 50], [100, 33], [330, 95],
                [410, 12], [475, 44], [25, 67], [85, 21], [220, 88]
              ];
d3.max(dataset);  // Returns [85, 21].  What???
```

To tell `max()` which *specific* values we want compared, we must include a second argument, an *accessor function*:

```
d3.max(dataset, function(d) {
    return d[0];
});
```

The accessor function is an anonymous function to which `max()` hands off each value in the data array, one at a time, as `d`. The accessor function specifies *how to access* the value to be used for the comparison. In this case, our data array is `dataset`, and we want to compare only the x values, which are the first values in each subarray, meaning in position `[0]`. So our accessor function looks like this:

```
function(d) {
    return d[0];  //Return the first value in each subarray
}
```

Note that this looks suspiciously similar to the syntax we used when generating our scatterplot circles, which also used anonymous functions to retrieve and return values:

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

This is a common D3 pattern. Soon you will be very comfortable with all manner of anonymous functions crawling all over your code.

Setting Up Dynamic Scales

Putting together what we've covered, let's create the scale function for our x-axis:

```
var xScale = d3.scaleLinear()
                .domain([0, d3.max(dataset, function(d) { return d[0]; })])
                .range([0, w]);
```

First, notice that I named it `xScale`. Of course, you can name your scales whatever you want, but a name like `xScale` helps me remember what this function does.

Second, notice that both the domain and range are specified as two-value arrays in hard brackets.

Third, notice that I set the low end of the input domain to 0. (Alternatively, you could use `min()` to calculate a dynamic value.) The upper end of the domain is set to the maximum value in `dataset` (which is currently 480, but could change in the future).

Finally, observe that the output range is set to `0` and `w`, the SVG's width.

We'll use very similar code to create the scale function for the y-axis:

```
var yScale = d3.scaleLinear()
                .domain([0, d3.max(dataset, function(d) { return d[1]; })])
                .range([0, h]);
```

Note that the `max()` function here references `d[1]`, the y value of each subarray. Also, the upper end of `range()` is set to `h` instead of `w`.

The scale functions are in place! Now all we need to do is use them.

Incorporating Scaled Values

Revisiting our scatterplot code, we now simply modify the original line where we created a `circle` for each data value. As a reminder, what follows is operating on a selection of newly created circles. These lines set those circles' positions along the x-axis:

```
.attr("cx", function(d) {
    return d[0];  //Returns original value bound from dataset
})
```

to return a scaled value (instead of the original value):

```
.attr("cx", function(d) {
    return xScale(d[0]);  //Returns scaled value
})
```

Likewise, for the y-axis, this:

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

is modified as:

```
.attr("cy", function(d) {
    return yScale(d[1]);
})
```

For good measure, let's make the same change where we set the coordinates for the text labels, so these lines:

```
.attr("x", function(d) {
    return d[0];
})
.attr("y", function(d) {
    return d[1];
})
```

become this:

```
.attr("x", function(d) {
    return xScale(d[0]);
})
.attr("y", function(d) {
    return yScale(d[1]);
})
```

And there we are!

Check out the working code in *02_scaled_plot.html*. Visually, the result in Figure 7-2 is disappointingly similar to our original scatterplot! Yet we are making more progress than might be apparent.

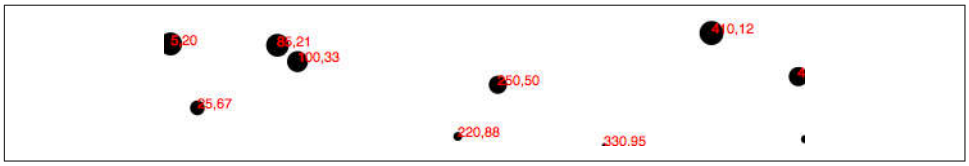

Figure 7-2. Scatterplot using x and y scales

Refining the Plot

You might have noticed that smaller y values are at the top of the plot, and the larger y values are toward the bottom. Now that we're using D3 scales, it's super easy to reverse that, so greater values are higher up, as you would expect. It's just a matter of changing the output range of `yScale` from:

```
.range([0, h]);
```

to:

```
.range([h, 0]);
```

See *03_scaled_plot_inverted.html* for the code that results in Figure 7-3.

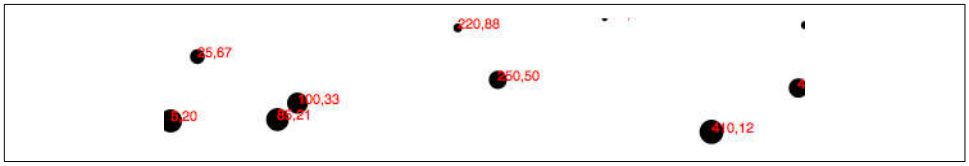

Figure 7-3. Scatterplot with y scale inverted

Yes, now a *smaller* input to `yScale` will produce a *larger* output value, thereby pushing those `circles` and `text` elements down, closer to the base of the image. I know, it's almost too easy!

Yet some elements are getting cut off. Let's introduce a `padding` variable:

```
var padding = 20;
```

Then we'll incorporate the `padding` amount when setting the range of both scales. This will help push our elements in, away from the edges of the SVG, to prevent them from being clipped.

The range for `xScale` was `range([0, w])`, but now it's:

```
.range([padding, w - padding]);
```

The range for `yScale` was `range([h, 0])`, but now it's:

```
.range([h - padding, padding]);
```

This should provide us with 20 pixels of extra room on the left, right, top, and bottom edges of the SVG. And it does; see Figure 7-4.

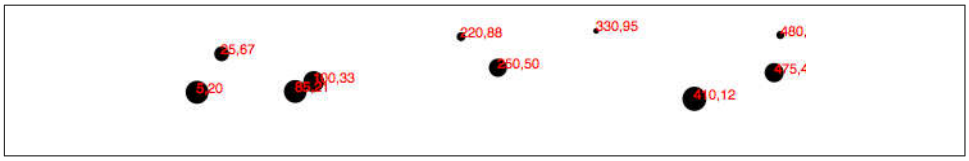

Figure 7-4. Scatterplot with padding

But the text labels on the far right are still getting cut off, so I'll double the amount of `xScale`'s padding on the right side by multiplying by two to achieve the result shown in Figure 7-5.

```
.range([padding, w - padding * 2]);
```

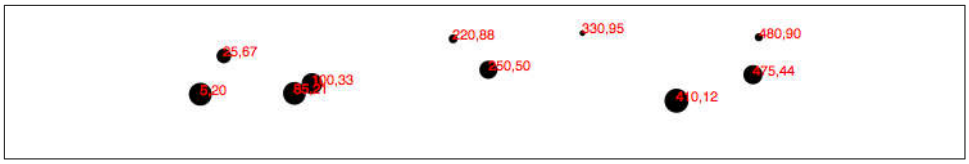

Figure 7-5. Scatterplot with more padding

The way I've introduced padding here is simple, but not elegant. Eventually, you'll want more control over how much padding is on each side of your charts (top, right, bottom, left), and it's useful to standardize how you specify those values across projects. Although I haven't used Mike Bostock's margin convention for the code samples in this book, I recommend taking a look to see if it could work for you.

Better! Reference the code so far in *04_scaled_plot_padding.html*. But there's one more change I'd like to make. Instead of setting the radius of each `circle` as the square root of its y value (which was a bit of a hack, and not visually useful), why not create another custom scale?

```
var rScale = d3.scaleLinear()
                .domain([0, d3.max(dataset, function(d) { return d[1]; })])
                .range([2, 5]);
```

Now we can use `rScale` to scale each circle's radius. (My admonishment to always scale circles by *area* and not *radius* still stands. Bear with me for a moment; we'll come back to this.)

```
.attr("r", function(d) {
    return rScale(d[1]);
});
```

This is exciting because we are guaranteeing that our radius values will *always* fall within the range of `2,5`. (Or *almost* always; see reference to `clamp()` later.) So data values of `0` (the minimum input) will get circles of radius `2` (or a diameter of 4 pixels). The very largest data value will get a circle of radius `5` (diameter of 10 pixels).

Voila: Figure 7-6 shows our first scale used for a visual property other than an axis value. (See *05_scaled_plot_radii.html*.)

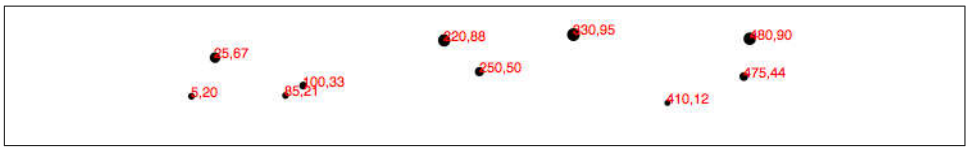

Figure 7-6. Scatterplot with scaled radii

Finally, just in case the power of scales hasn't yet blown your mind, I'd like to add one more array to the dataset: `[600, 150]`.

Boom! Check out *06_scaled_plot_big.html*. Notice how all the old points in Figure 7-7 maintained their relative positions but have migrated closer together, down and to the left, to accommodate the newcomer in the top-right corner.

Figure 7-7. Scatterplot with big numbers added

And now, one final revelation: we can now very easily change the size of our SVG, and *everything scales accordingly*. In Figure 7-8, I've increased the value of `h` from `100` to `300` and made *no other changes*.

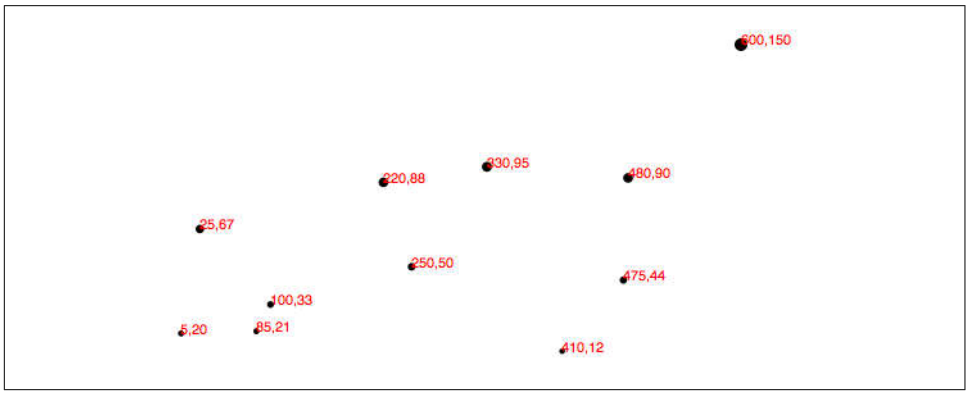

Figure 7-8. Large, scaled scatterplot

Boom, again! See *07_scaled_plot_large.html*. Hopefully, you are seeing this and realizing: no more late nights tweaking your code because the client decided the graphic should be 800 pixels wide instead of 600. Yes, you will get more sleep because of me (and D3's brilliant built-in methods). Being well-rested is a competitive advantage. You can thank me later.

Other Methods

`d3.scaleLinear()` has several other handy methods that deserve a brief mention here:

`nice()`
: This tells the scale to take whatever input domain that you gave to `domain()` and expand both ends to the nearest round value. From the D3 wiki: "For example, for a domain of [0.201479…, 0.996679…], a nice domain might be [0.2, 1.0]." This is useful for humans, who are not computers and find it hard to read numbers like 0.20147987687960267.

`rangeRound()`
: Use `rangeRound()` in place of `range()`, and all values output by the scale will be rounded to the nearest whole number. This is useful if you want shapes to have exact pixel values, to avoid the fuzzy edges that could arise with antialiasing.

`clamp()`
: By default, a linear scale *can* return values outside of the specified range. For example, if given a value outside of its expected input domain, a scale will return a number also outside of the output range. Calling `clamp(true)` on a scale, however, forces all output values to be within the specified range. This means excessive values will be rounded to the range's low or high value (whichever is nearest).

To use any of these special methods, just tack them onto the chain in which you define the original scale function. For example, to use `nice()`:

```
var scale = d3.scaleLinear()
              .domain([0.123, 4.567])
              .range([0, 500])
              .nice();
```

Other Scales

In addition to `linear scales` (discussed earlier), D3 has several other built-in scale methods:

`scaleSqrt`
: A square root scale.

`scalePow`
: A power scale (good for the gym, er, I mean, useful when working with exponential series of values, as in "to the power of" some exponent).

`scaleLog`
: A logarithmic scale.

`scaleQuantize`
: A linear scale with discrete values for its output range, for when you want to sort data into "buckets."

`scaleQuantile`
: Similar to `scaleQuantize`, but with discrete values for its input domain (when you already have "buckets").

`scaleOrdinal`
: Ordinal scales use nonquantitative values (like category names) for output; perfect for comparing apples and oranges.

`schemeCategory10`, `schemeCategory20`, `schemeCategory20b`, *and* `schemeCategory20c`
: Handy preset ordinal scales that output either 10 or 20 categorical colors.

`scaleTime`
: A scale method for date and time values, with special handling of ticks for dates.

Before we move on to axes, let's explore how to use two of these: square root and time scales.

Square Root Scales

In Chapter 6, I mentioned how circles should always be scaled by *area*, not by *radius* value, to better match how we perceive circle sizes. To get from data value (effectively the "area" value) to a radius value, we used `Math.sqrt()`, like so:

```
.attr("r", function(d) {
    return Math.sqrt(d);
});
```

If we used a square root scale instead, the code would look a little different:

```
.attr("r", function(d) {
    return aScale(d);  //'a' scale for 'area'!
});
```

The main benefit is we get to take advantage of the scale's ability to set up a domain and a range (and to update those in the future, should our needs change). Why not let the scale handle all the math?

Let's replace our radius `rScale` with an area `aScale` as follows:

```
var aScale = d3.scaleSqrt()      // <--New!
               .domain([0, d3.max(dataset, function(d) { return d[1]; })])
               .range([0, 10]);  // <--New!
```

There are only three changes here:

1. Switching `scaleLinear` to `scaleSqrt`.
2. Renaming the scale to `aScale`. This is still being used to set each circle's `r` radius value (because there is no `circle-area` attribute in SVG), but the value is calculated with the data value mapping to *area*, per best practices.
3. Adjusting the `range`, which I only did to make the relative differences in size more obvious. Note that the values 0 and 10 are arbitrary. Remember, what matters is the *relative* size (areas) of the circles, not the actual or absolute sizes (areas).

See *08_scaled_plot_sqrt_scale.html* for the result, which looks like Figure 7-9.

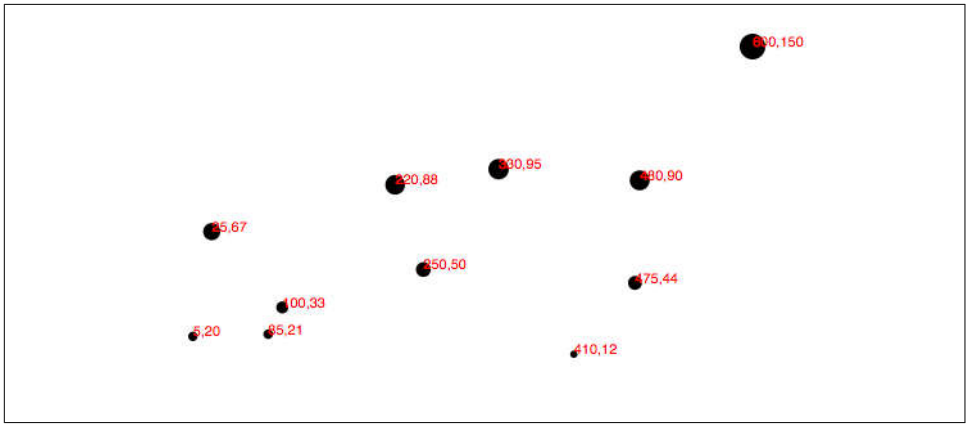

Figure 7-9. Using a square root scale for circle areas

Time Scales

We frequently need to visualize data with a temporal dimension. Fortunately, D3 has you covered, although there are a few extra steps required.

First, meet the JavaScript `Date` object. It's how JavaScript understands time. Open your console and type `new Date`. In return, you'll see a string representation of a new `Date` object that represents whatever date and time it is *right now*.

```
Mon Feb 20 2017 13:44:00 GMT-0800 (PST)
```

As a smart human being, you can mentally parse this to understand I was writing this chapter on February 20, 2017, at 1:44 pm. That happened to be a Monday, and I'm in the Pacific Standard Time zone, which is 8 hours behind Coordinated Universal Time.

(I won't go down the temporal rabbit hole here, but it can be enlightening to read about UTC, the nature of time, and various attempts to corral this relative concept in code, such as the wonderful Moment.js. I also highly recommend episode #70 of the Data Stories podcast with Enrico Bertini and Moritz Stefaner, "Rocket Science with Rachel Binx," in which Rachel discusses the challenges of coordinating interplanetary time measurements.)

JavaScript and D3 can only perform time and date calculations on `Date` objects, not on strings (even strings that look very much like dates, to human eyes). So working with dates in D3 involves:

1. Converting strings to `Date` objects
2. Using time scales, as needed
3. Formatting `Date` objects as human-friendly strings, for display to the user

Let's take these one at a time.

Converting strings to dates

I've provided some dummy data as a time series in *time_scale_data.csv*, which looks like this:

```
Date,Amount
01/01/17,35
01/02/17,30
01/03/17,24
01/04/17,37
01/05/17,54
…
```

Each row represents a daily measurement, with one `Amount` for each day in January 2017. (The data may be random and meaningless, but at least it's timely!)

As a human who can interpret context, you've already deduced that these dates are in the format month/day/year, a sequence common in the United States and pretty much nowhere else. In order to tell the context-insensitive computer how to interpret these strings, we write:

```
//For converting strings to Dates
var parseTime = d3.timeParse("%m/%d/%y");
```

The strange syntax of `"%m/%d/%y"` tells D3 to look for three values, separated by slashes: month with leading zero, day of the month with leading zero, and two-digit year number.

Personally, I can never remember all the various signifiers for each date/time component. I recommend the API reference for all possible time formatting values.

To verify this really works, try opening *09_time_scale.html* in your browser. Then, in the console, try parsing your birthday into a `Date` object by using the format mm/dd/yy. For example:

```
parseTime("02/20/17")
//Returns: Mon Feb 20 2017 00:00:00 GMT-0800 (PST)
```

Since we specified only a two-digit year, JavaScript has to guess about which century we intended: the 21st, or the 20th? If you were born in 1969 or since then, JavaScript will guess right:

```
parseTime("02/20/69")
//Returns: Thu Feb 20 1969 00:00:00 GMT-0800 (PST)
```

If you were born in 1968 or earlier, however, it may guess you are from the future:

```
parseTime("02/20/68")
//Returns: Mon Feb 20 2068 00:00:00 GMT-0800 (PST)
```

Finally, if you really *are* from the future, please email me. I have lots of questions.

In all seriousness, to avoid ambiguity with dates, I strongly recommend using four-digit years and `%Y`. Use your spreadsheet to reformat date values before exporting to CSV for D3.

Using a technique described in Chapter 5, I'll define a row conversion function. Once our CSV data is loaded in, this function is called by `d3.csv()` on each row. We have just two values per row (date and amount), and this specifies how to convert each from a string to a `Date` object or number, respectively. (It's easier to do this conversion now, rather than remember to do it later.) Note how we pass the string `d.Date` to `parseTime()`, and it returns a `Date` object.

```
//Function for converting CSV values from strings to Dates and numbers
var rowConverter = function(d) {
    return {
        Date: parseTime(d.Date),
        Amount: parseInt(d.Amount)
     };
}
```

To verify that this worked, I jump to the console and type `dataset`. See the result in Figure 7-10.

Figure 7-10. Look mom, no strings!

Scaling time

The hard part is over. Our data is loaded in, and the strings are now dates. We use `scaleTime()` to define a time scale:

```
xScale = d3.scaleTime()
           .domain([
                d3.min(dataset, function(d) { return d.Date; }),
                d3.max(dataset, function(d) { return d.Date; })
           ])
           .range([padding, w - padding]);
```

You'll recognize `d3.min()` and `d3.max()` from earlier. How cool that they work on dates, and not just simple, numeric values! We can verify that the domain matches our dataset by typing `xScale.domain()` in the console, as in Figure 7-11.

Figure 7-11. Verifying the xScale domain runs from January 1 through January 31, 2017

Finally, as with any scale, there is one more step: to actually call the scale, to convert data values as needed. In this case, I'm using:

```
.attr("cx", function(d) {
     return xScale(d.Date);
})
```

The only change here is that we're specifying `d.Date` as the value to be scaled. See the final chart in Figure 7-12.

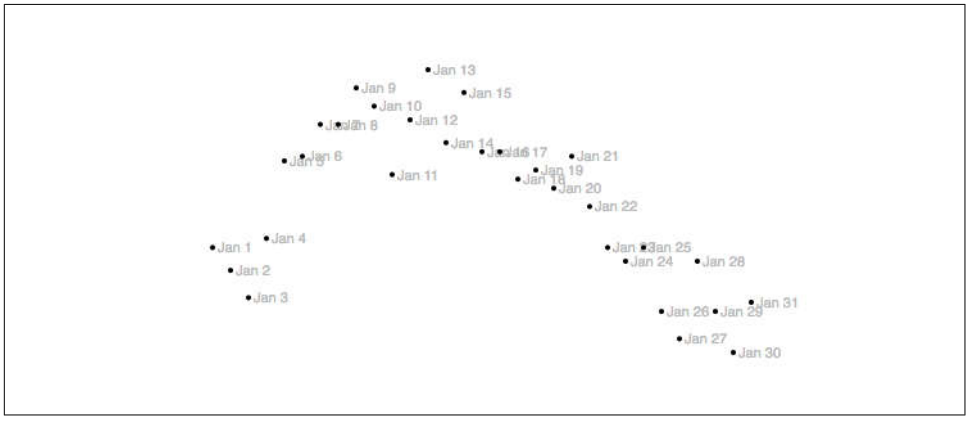

Figure 7-12. Time-scaled circles

It's not clean or beautiful (or meaningful), but it works! You can see the circles being positioned in chronological order, from left to right, and also positioned vertically, per their `Amount` values.

Formatting dates and strings

To create the labels on that chart, we need to convert our `Date` values back to human-readable strings. I've chosen to use the format of the month name's three-letter abbreviation followed by the day of the month. This is defined in a new time formatting function as:

```
//For converting Dates to strings
var formatTime = d3.timeFormat("%b %e");
```

Again, please see the API reference for time formatting options. You can take my word for it that `%b` results in Jan, Feb, Mar, and so on, while `%e` results in 1, 2, 3, and so on up until the end of the month.

Having specified the formatter, we call it whenever we need those human-readable strings generated, such as for inserting the text for those labels:

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

Explore the working code in *09_time_scale.html*. Try tweaking the time formatter options. Can you get it to return the full month name? How about the four-digit year? How about the time zone or day of the week? Once you are comfortable with time scales, you'll see it's no harder working on a small scale (seconds) than on a large one (centuries).

Now that you have mastered the power of scales, it's time to express them visually as, yes, *axes*!

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*.

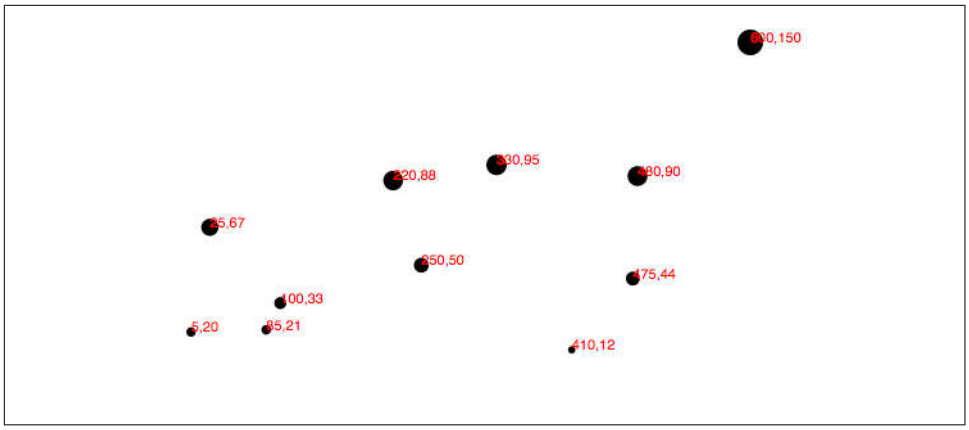

Figure 8-1. Large, scaled scatterplot

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*.

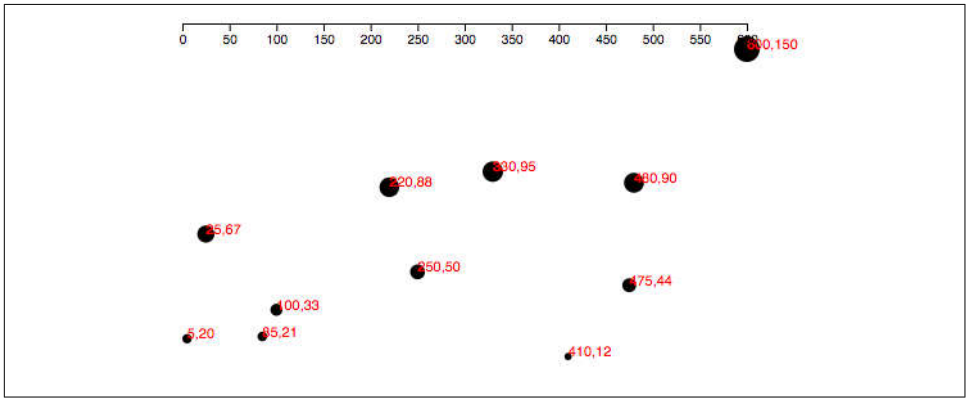

Figure 8-2. Simple axis, but in the wrong place

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*.

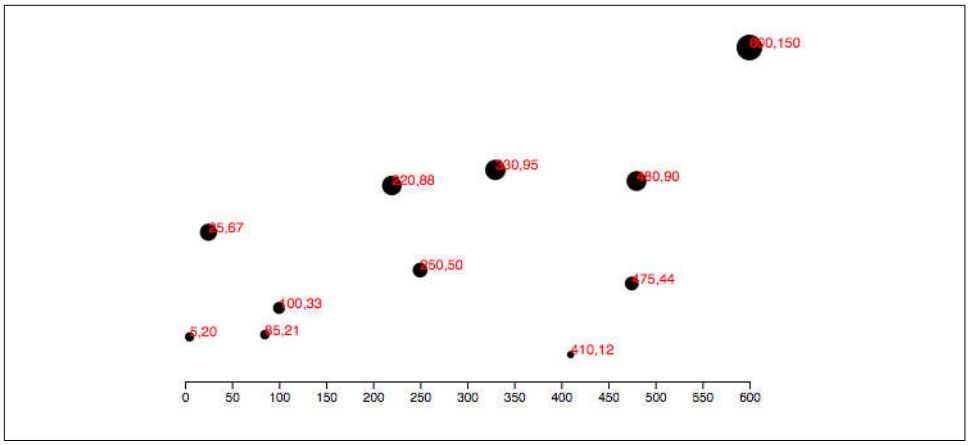

Figure 8-3. Correctly positioned axis

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.

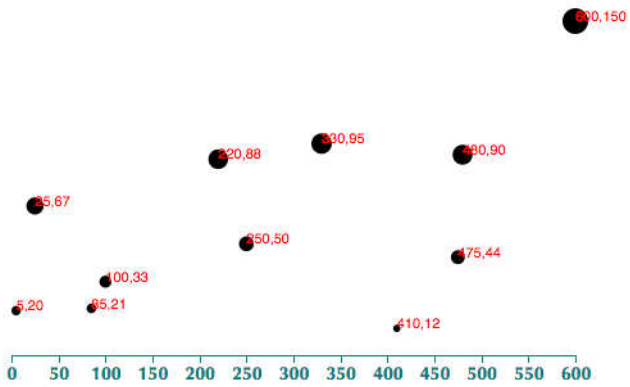

Figure 8-4. Axis with styles overridden with CSS

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.

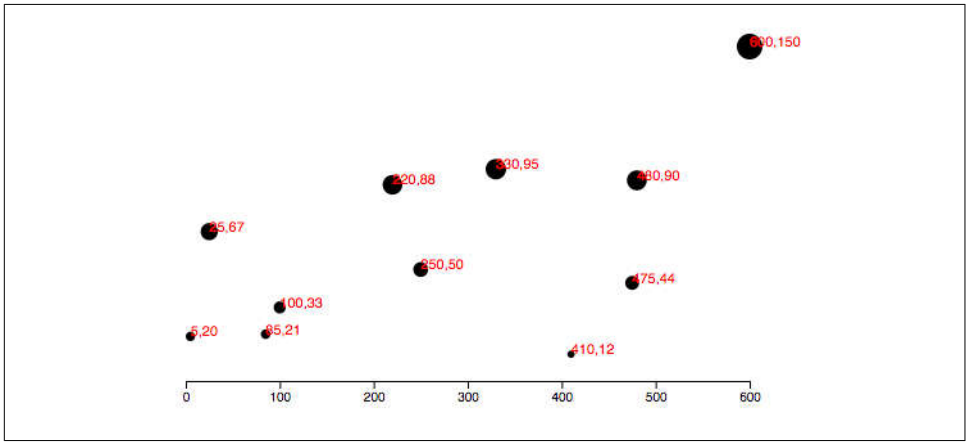

Figure 8-5. Fewer ticks

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.

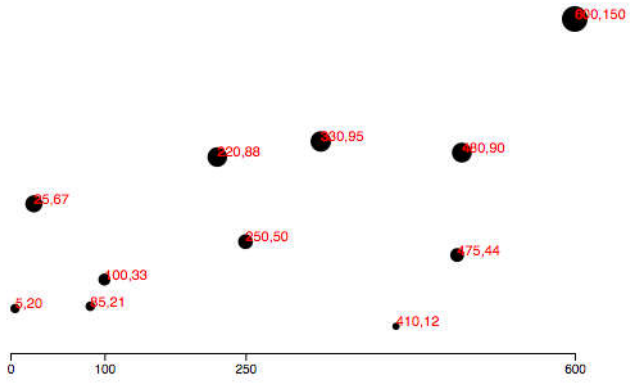

Figure 8-6. Manually specified tick values

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`.

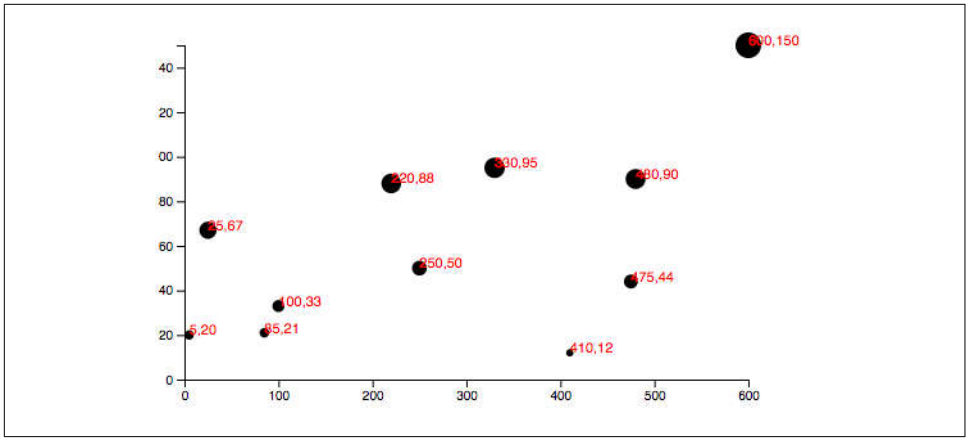

Figure 8-7. Initial y-axis

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.

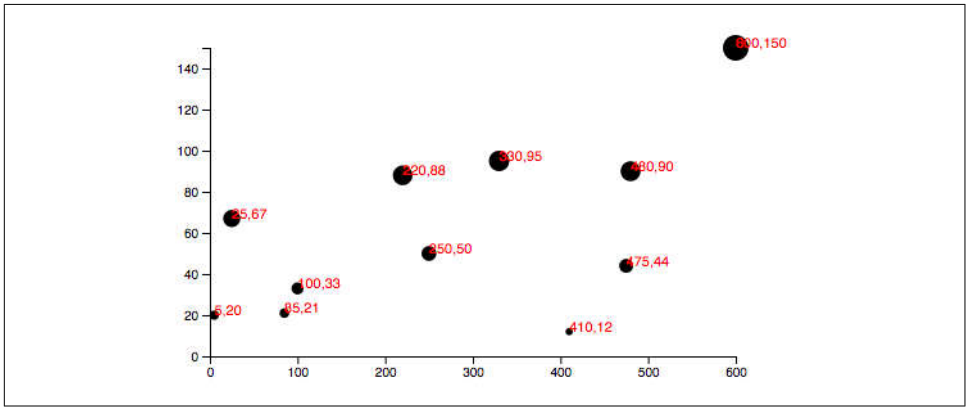

Figure 8-8. Scatterplot with y-axis

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).

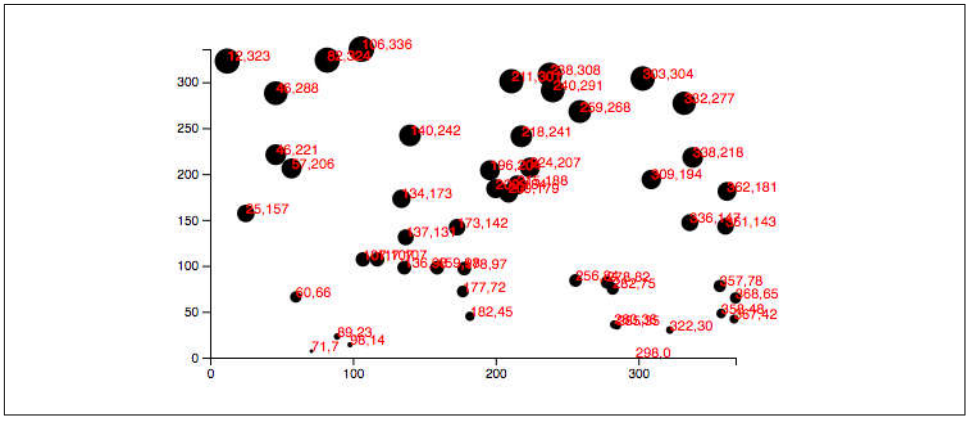

Figure 8-9. Scatterplot with random data

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*.

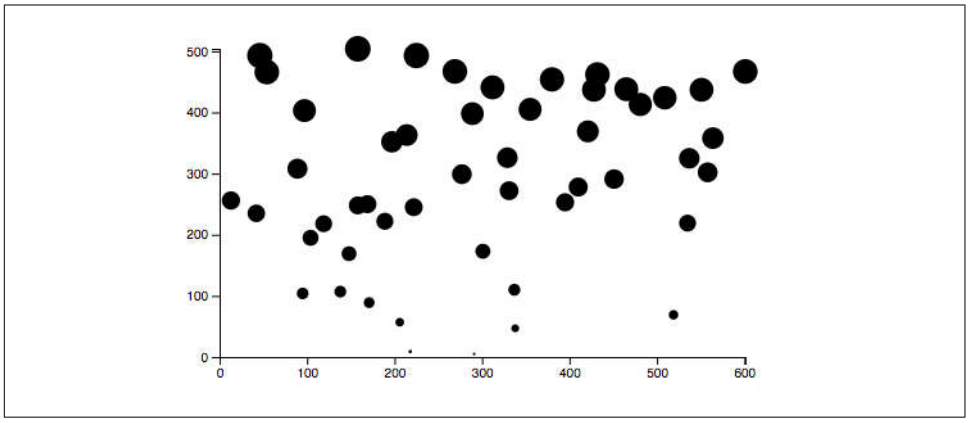

Figure 8-10. Scatterplot with random data and no red labels

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.

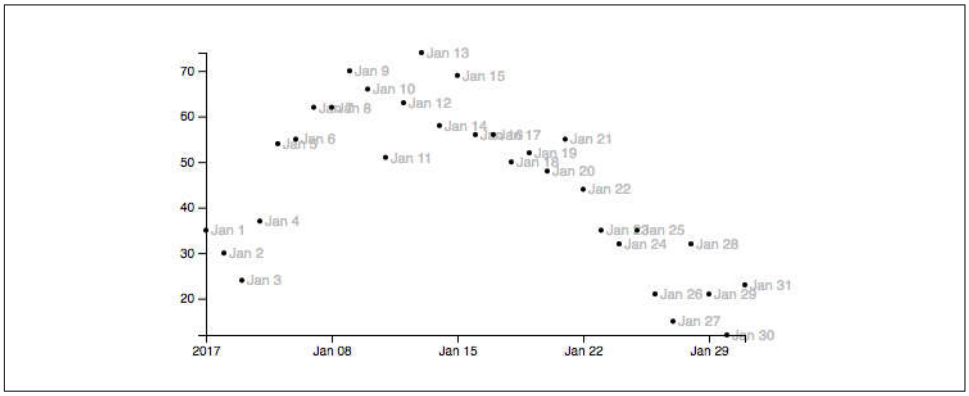

Figure 8-12. Easy time-based axis

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.

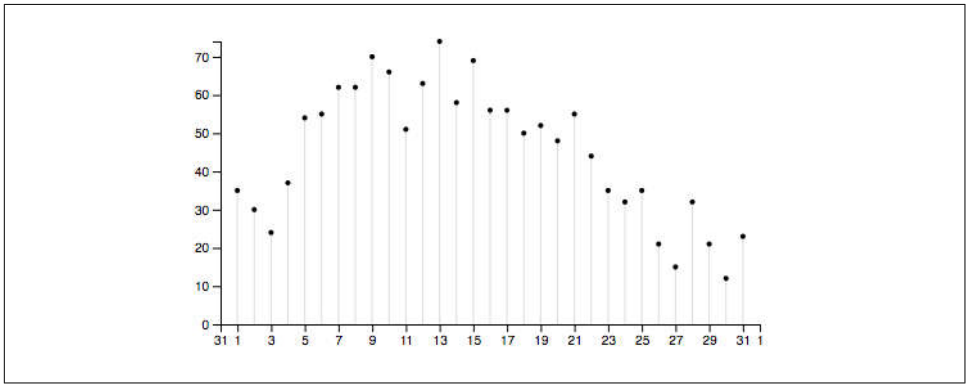

Figure 8-13. Time series, cleaned up