Of course, you could name `w` and `h` something else, like `svgWidth` and `svgHeight`. Use whatever is most clear to you. JavaScript programmers, as a group, are fixated on efficiency, so you'll often see single-character variable names, code written with no spaces, and other hard-to-read, yet programmatically efficient, syntax.

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

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

To recap, this inserts a new `<svg>` element just before the closing `</body>` tag, and assigns the SVG a width and height of 500 by 100 pixels. This statement also puts the result into our new variable called `svg`, so we can easily reference the new SVG without having to reselect it later using something like `d3.select("svg")`.

Next, instead of creating `divs`, we generate `rects` and add them to `svg`:

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

This code selects all `rects` inside of `svg`. Of course, there aren't any yet, so an empty selection is returned. (Weird, yes, but stay with me. With D3, you always have to first select whatever it is you're about to act on, even if that selection is momentarily empty.)

Then, `data(dataset)` sees that we have 20 values in the dataset, and those values are handed off to `enter()` for processing. `enter()`, in turn, returns a placeholder selection for each data point that does not yet have a corresponding `rect`—which is to say, all of them.

For each of the 20 placeholders, `append("rect")` inserts a `rect` into the DOM. As we learned in Chapter 3, every `rect` must have `x`, `y`, `width`, and `height` values. We use `attr()` to add those attributes onto each newly created `rect`.

Beautiful, no? Okay, maybe not. All of the bars are there (check the DOM of *13_making_a_bar_chart_rects.html* with your web inspector), but they all share the same `x`, `y`, `width`, and `height` values, with the result that they all overlap (see Figure 6-14). This isn't a visualization of data yet.

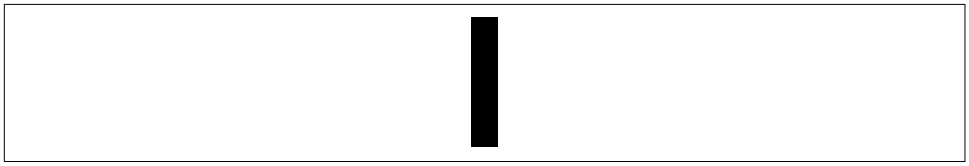

Figure 6-14. One lonely bar

Let's fix the overlap issue first. Instead of an `x` of `0`, we'll assign a dynamic value that corresponds to `i`, or each value's position in the dataset. So the first bar will be at `0`, but subsequent bars will be at `21`, then `42`, and so on. (In a later chapter, we'll learn about D3's *scales*, which offer a better, more flexible way to accomplish this same feat.)

```
.attr("x", function(d, i) {
    return i * 21;  //Bar width of 20 plus 1 for padding
})
```

See that code in action with *14_making_a_bar_chart_offset.html* and the result in Figure 6-15.

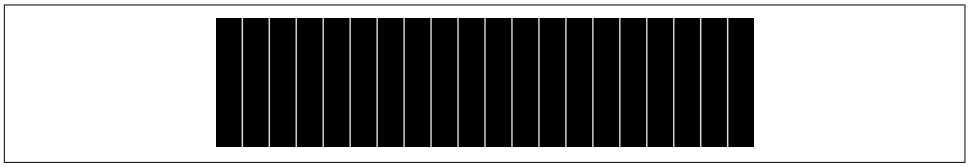

Figure 6-15. Twenty bars

That works, but it's not particularly flexible. If our dataset were longer, then the bars would just run off to the right, past the end of the SVG! Because each bar is 20 pixels wide, plus 1 pixel of padding, a 500-pixel wide SVG can only accommodate 23 data points. Note how the 24th bar gets clipped in Figure 6-16.

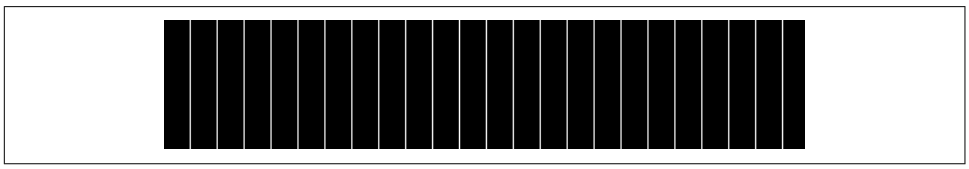

Figure 6-16. Twenty-four bars

It's good practice to use flexible, dynamic coordinates—heights, widths, x values, and y values—so your visualization can scale appropriately along with your data.

As with anything else in programming, there are a thousand ways to achieve that end. I'll use a simple one. First, I'll amend the line where we set each bar's x-position:

```
    .attr("x", function(d, i) {
        return i * (w / dataset.length);
    })
```

Notice how the x value is now tied directly to the width of the SVG (`w`) and the number of values in the dataset (`dataset.length`). This is exciting because now our bars will be evenly spaced, whether we have 20 data values, as in Figure 6-17…

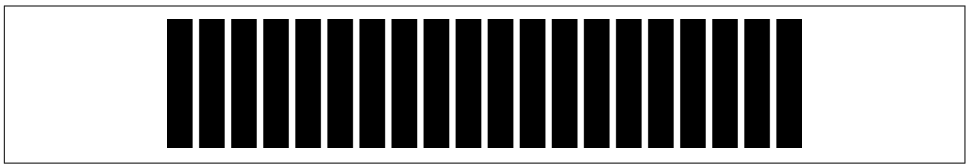

Figure 6-17. Twenty evenly spaced bars

…or only five, as in Figure 6-18.

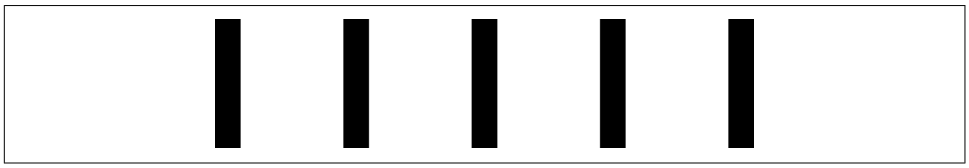

Figure 6-18. Five evenly spaced bars

See that code so far in *15_making_a_bar_chart_even.html*.

Now we should set the bar *widths* to be proportional, too, so they get narrower as more data is added, or wider when there are fewer values. I'll add a new variable near where we set the SVG's width and height:

```
//Width and height
var w = 500;
var h = 100;
var barPadding = 1;  // <-- New!
```

and then reference that variable in the line where we set each bar's `width`. Instead of a static value of `20`, the width will now be set as a fraction of the SVG width and number of data points, minus a padding value:

```
    .attr("width", w / dataset.length - barPadding)
```

It works! (See Figure 6-19 and *16_making_a_bar_chart_widths.html*.)

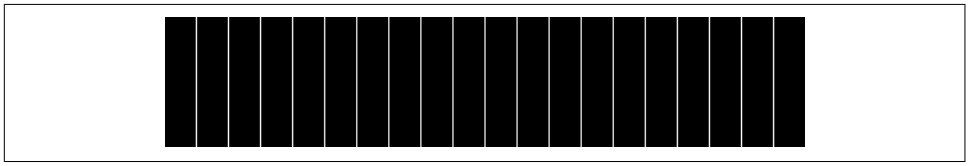

Figure 6-19. Twenty evenly spaced bars with dynamic widths

The bar widths and x-positions scale correctly whether there are 20 points, only 5 (see Figure 6-20), or even 100 (see Figure 6-21).

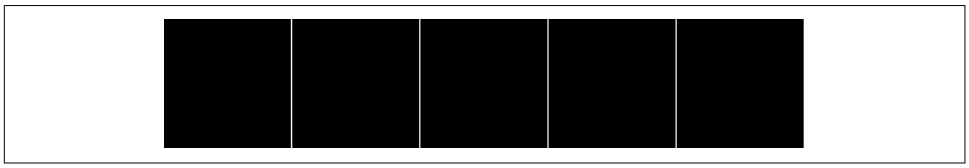

Figure 6-20. Five evenly spaced bars with dynamic widths

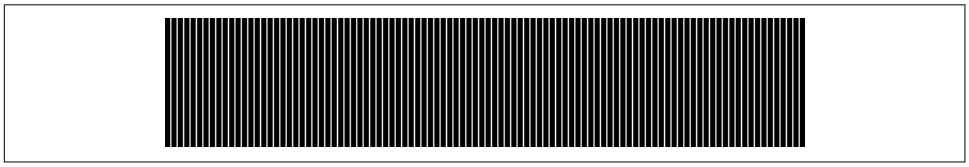

Figure 6-21. One hundred evenly spaced bars with dynamic widths

Finally, we encode our data as the *height* of each bar. You would hope it were as easy as referencing the `d` data value when setting each bar's `height`:

```
.attr("height", function(d) {
    return d;
});
```

Hmm, the chart in Figure 6-22 looks funky.

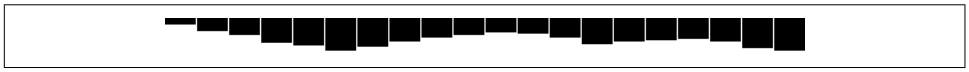

Figure 6-22. Dynamic heights

Maybe we can just scale up our numbers a bit?

```
.attr("height", function(d) {
    return d * 4;  // <-- Times four!
});
```

Alas, it is not that easy! We want our bars to grow upward from the bottom edge, not down from the top, as in Figure 6-23—but don't blame D3, blame SVG.

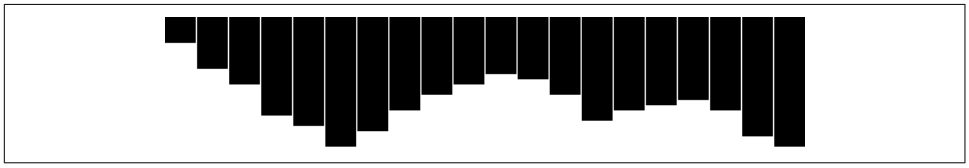

Figure 6-23. Dynamic heights, magnified

You'll recall that, when drawing SVG `rect`s, the x and y values specify the coordinates of the *upper-left corner*. That is, the origin or reference point for every `rect` is its top left. For our purposes, it would be soooooo much easier to set the origin point as the bottom-left corner, but that's just not how SVG does it, and frankly, SVG is indifferent about our feelings on the matter.

Given that our bars do have to "grow down from the top," then where is "the top" of each bar in relationship to the top of the SVG? Well, the top of each bar could be expressed as a relationship between the height of the SVG and the corresponding data value, as in:

```
.attr("y", function(d) {
    return h - d;  //Height minus data value
})
```

Then, to put the "bottom" of the bar on the bottom of the SVG (see Figure 6-24), each `rect`'s height can be just the data value itself:

```
.attr("height", function(d) {
    return d;  //Just the data value
});
```

Figure 6-24. Growing down from above

Let's scale things up a bit by changing `d` to `d * 4`, with the result shown in Figure 6-25. (Just as with the bar placements, we could do this more properly using D3 scales, but we're not there yet.)

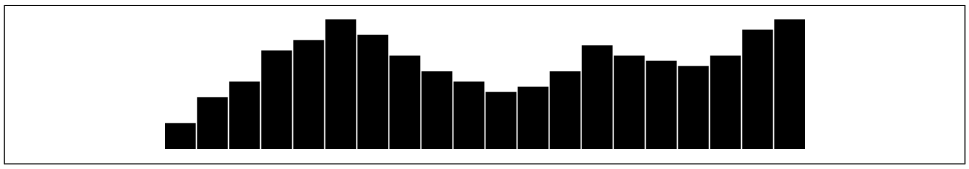

Figure 6-25. Growing bigger from above

The working code for our growing-down-from-above, SVG bar chart is in *17_making_a_bar_chart_heights.html*.

Color

Adding color is easy. Just use `attr()` to set a `fill`:

```
    .attr("fill", "teal");
```

Find the all-teal bar chart shown in Figure 6-26 in *18_making_a_bar_chart_teal.html*.

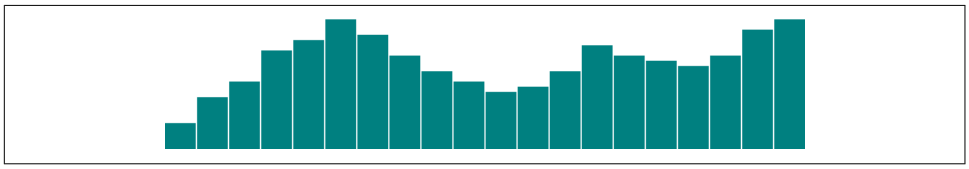

Figure 6-26. Teal bars

Teal is nice, but you'll often want a shape's color to reflect some quality of the data. That is, you might want to *encode* the data values as color. (In the case of our bar chart, that makes a *dual encoding*, in which the same data value is encoded in two different visual properties: both height and color.)

Using data to drive color is as easy as writing a custom function that again references `d`. Here, we replace `"teal"` with a custom function, resulting in the chart in Figure 6-27.

```
    .attr("fill", function(d) {
        return "rgb(0, 0, " + Math.round(d * 10) + ")";
    });
```

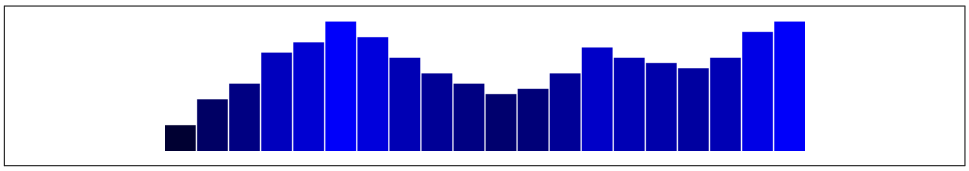

Figure 6-27. Data-driven blue bars

See the code in *19_making_a_bar_chart_blues.html*. This is not a particularly useful visual encoding, but you can get the idea of how to translate data into color. Here, `d` is multiplied by 10, and then rounded to the nearest whole number with `Math.round()`. The resulting number is used as the blue value in an `rgb()` color definition. So the greater values of `d` (taller bars) will be more blue. Smaller values of `d` (shorter bars) will be less blue (closer to black). The red and green components of the color are fixed at zero.

Exercise

Try manipulating these RGB values on your own to get a feel for how they work.

Labels

Visuals are great, but sometimes you need to show the actual data values as text within the visualization. Here's where value labels come in, and they are very easy to generate with D3.

You'll recall from the SVG primer that you can add `text` elements to an SVG element. Let's start with:

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

Look familiar? Just as we did for the `rect`s, here we do for the `text`s. First, select what you want, bring in the data, enter the new elements (which are just placeholders at this point), and finally append the new `text` elements to the DOM.

We'll extend that code to include a data value within each `text` element by using the `text()` method:

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

and then extend it further, by including x and y values to position the text. It's easiest if I just copy and paste the same x/y code we previously used for the bars:

```
   .attr("x", function(d, i) {
       return i * (w / dataset.length);
   })
   .attr("y", function(d) {
       return h - (d * 4);
   });
```

Aha! Value labels! But some are getting cut off at the top (see Figure 6-28).

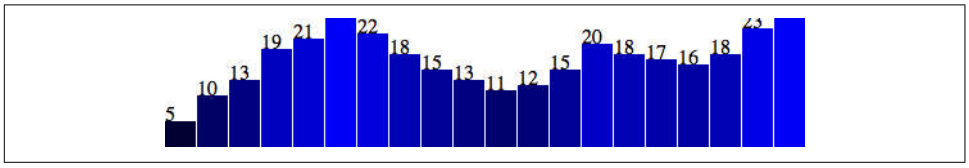

Figure 6-28. Baby value labels!

Let's try moving them down, inside the bars, by adding a small amount to the x and y calculations:

```
.attr("x", function(d, i) {
    return i * (w / dataset.length) + 5;  // +5
})
.attr("y", function(d) {
    return h - (d * 4) + 15;  // +15
});
```

The chart in Figure 6-29 is better, but not legible.

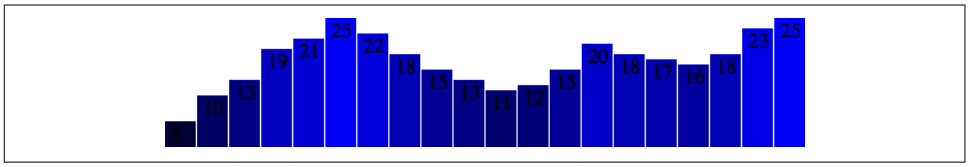

Figure 6-29. In-bar value labels

Fortunately, we can fix that:

```
.attr("font-family", "sans-serif")
.attr("font-size", "11px")
.attr("fill", "white");
```

Fantasti-code! See *20_making_a_bar_chart_labels.html* for the brilliant visualization shown in Figure 6-30.

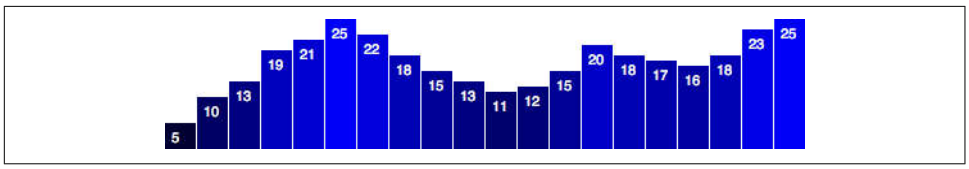

Figure 6-30. Really nice value labels

If you are not typographically obsessive, then you're all done. If, however, you are like me, you'll notice that the value labels aren't perfectly aligned within their bars. (For example, note the "5" in the first column.) That's easy enough to fix. Let's use the SVG `text-anchor` attribute to center the text horizontally at the assigned x value:

```
.attr("text-anchor", "middle")
```

Then, let's change the way we calculate the x-position by setting it to the left edge of each bar *plus* half the bar width:

```
.attr("x", function(d, i) {
    return i * (w / dataset.length) + (w / dataset.length - barPadding) / 2;
})
```

And I'll also bring the labels up one pixel for perfect spacing, as you can see in Figure 6-31 and *21_making_a_bar_chart_aligned.html*:

```
        .attr("y", function(d) {
            return h - (d * 4) + 14;  //15 is now 14
        })
```

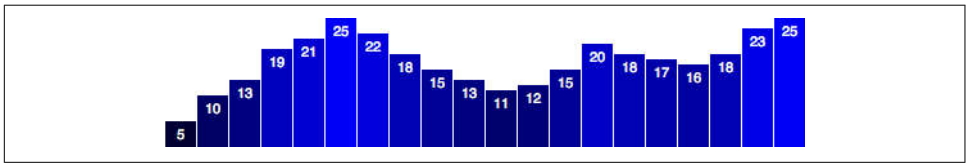

Figure 6-31. Centered labels

Making a Scatterplot

So far, we've drawn only bar charts with simple data—just one-dimensional sets of numbers.

But when you have two sets of values to plot against each other, you need a second dimension. The scatterplot is a common type of visualization that represents two sets of corresponding values on two different axes: horizontal and vertical, x and y.

The Data

As you saw in Chapter 3, you have a lot of flexibility around how to structure a dataset. For our scatterplot, I'm going to use an array of arrays. The primary array will contain one element for each data "point." Each of those "point" elements will be another array, with just two values: one for the x value, and one for y:

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

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

We could rewrite our dataset with more whitespace so it's easier to read:

```
var dataset = [
                  [  5,   20 ],
                  [ 480,  90 ],
                  [ 250,  50 ],
                  [ 100,  33 ],
                  [ 330,  95 ],
                  [ 410,  12 ],
                  [ 475,  44 ],
```

```
            [  25,   67 ],
            [  85,   21 ],
            [ 220,   88 ]
        ];
```

Now you can see that each of these 10 rows will correspond to one point in our visualization. With the row `[5, 20]`, for example, we'll use `5` as the x value, and `20` for the y.

The Scatterplot

Let's carry over most of the code from our bar chart experiments, including the piece that creates the SVG element:

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

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

```
svg.selectAll("circle")  // <-- No longer "rect"
   .data(dataset)
   .enter()
   .append("circle")     // <-- No longer "rect"
```

Also, instead of specifying the `rect` attributes of `x`, `y`, `width`, and `height`, our `circles` need `cx`, `cy`, and `r`:

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

See the working scatterplot code that recreates the result shown in Figure 6-32 in *22_scatterplot.html*.

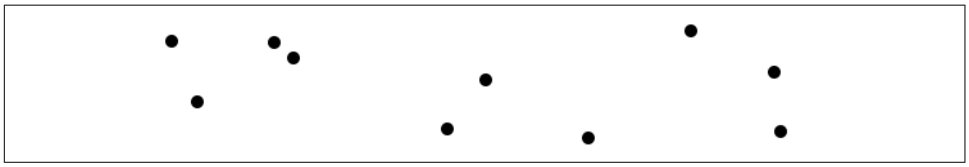

Figure 6-32. Simple scatterplot

Notice how we access the data values and use them for the `cx` and `cy` values. When using `function(d)`, D3 automatically hands off the current data value as `d` to your

function. In this case, the current data value is one of the smaller subarrays in our larger `dataset` array.

When each single datum `d` is itself an array of values (and not just a single value, like `3.14159`), you need to use bracket notation to access its values. Hence, instead of `return d`, we use `return d[0]` and `return d[1]`, which return the first and second values of the array, respectively.

For example, in the case of our first data point `[5, 20]`, the first value (array position `0`) is `5`, and the second value (array position `1`) is `20`. Thus:

```
d[0] returns 5
d[1] returns 20
```

By the way, if you ever want to access any value in the larger dataset (outside of D3, say), you can do so using bracket notation. For example:

```
dataset[5] returns [410, 12]
```

You can even use multiple sets of brackets to access values within nested arrays:

```
dataset[5][1] returns 12
```

Don't believe me? Take another look at the scatterplot page *22_scatterplot.html*, open your JavaScript console, type in **`dataset[5]`** or **`dataset[5][1]`**, and see what happens.

Size

Maybe you want the circles to be different sizes, so each circle's area corresponds to its y value. As a general rule, when visualizing quantitative values with circles, make sure to encode the values as *area*, not as a circle's *radius*. Perceptually, humans interpret the overall amount of "ink" or pixels (the area) to reflect the data value. A common mistake is to map the value to the radius, which would vastly overrepresent the data and distort the relative relationship between values. (For that matter, humans are not so great at accurately comparing *areas*, either, but that's another discussion.) Mapping to the radius is easier to do, as it requires less math, but the result will visually distort your data.

Yet when creating SVG circles, we can't specify an `area` value; we have to calculate the radius `r` and then set that. So, starting with a data value as area, how do we get to a radius value?

You might remember that the area of a circle equals π times the radius squared, or $A = \pi r^2$.

To solve for *r*, we can rework the equation like so:

```
A = π r^2          //Original equation for area
A / π = r^2        //Divide both sides by pi
```

```
sqrt ( A / n ) = r  //Take the square root of both sides
r = sqrt ( A / n )  //Flip the equation around for legibility
```

So our solution for r is $r = \sqrt{\frac{A}{\pi}}$. As long as we know the area A, we just divide by pi, then take the square root in order to get the radius.

For us, the area of each circle is driven by a data value—`d[1]`, in this case. Actually, let's subtract that value from `h`, so the circles at the top are larger. So our *area* value A is `h - d[1]`. (Admittedly, it is not a meaningful to include `h` here; please just bear with me for the sake of the example. I promise to illustrate a cleaner and more meaningful approach using scales in Chapter 7.) We could update our equation as pseudocode as follows:

```
r = sqrt ( ( h - d[1] ) / n )
```

The D3 code equivalent is:

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

That said, we can actually omit the pi part, resulting in the simpler:

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

"That's not possible," you say. "Archimedes's equation $A = \pi r^2$ is sacred! You can't arbitrarily change it to $A = r^2$!"

You are right, of course! If the "area" here were an actual area value of an actual, measured circle—such as that of an 18-inch pizza (254 in^2 or 1.77 ft^2)—then we should divide by pi. But since the "areas" of our circles are just arbitrary data values and not real-life measurements, dividing by pi merely reduces each number to about a third of its original value. What matters here is not the *actual* circle areas, but the *relative* areas. The actual areas will vary greatly, anyway, when your chart is viewed on different devices and displays. For purposes of honest visual representation, dividing by pi is not necessary; it has the effect of simply making all circles equally smaller.

See *23_scatterplot_sqrt.html* for the code that results in the scatterplot shown in Figure 6-33.

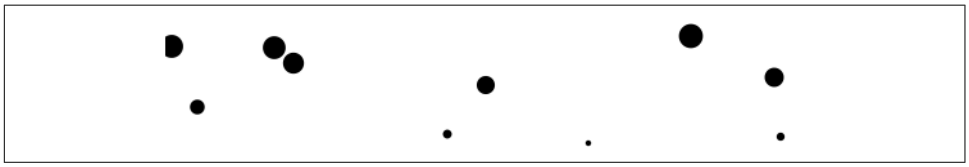

Figure 6-33. Scatterplot with sized circles

After arbitrarily subtracting the datum's y value `d[1]` from the SVG height `h`, and then taking the square root, we see that circles with greater y values (those circles lower down) have smaller areas (and shorter radii).

This particular use of circle area as a visualization tool isn't necessarily useful. I simply want to illustrate how you can use `d`, along with bracket notation, to reference an individual datum, apply some transformation to that value, and use the newly calculated value to *return* a value back to the attribute-setting method (a value used for `r`, in this case).

Labels

Let's label our data points with `text` elements. I'll adapt the label code from our bar chart experiments, starting with the following:

```
svg.selectAll("text")  // <-- Note "text", not "circle" or "rect"
   .data(dataset)
   .enter()
   .append("text")     // <-- Same here!
```

This looks for all `text` elements in the SVG (there aren't any yet), and then appends a new `text` element for each data point. Then we use the `text()` method to specify each element's contents:

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

This looks messy, but bear with me. Once again, we're using `function(d)` to access each data point. Then, within the function, we're using *both* `d[0]` *and* `d[1]` to get both values within that data point array.

The plus `+` symbols, when used with strings, such as the comma between quotation marks `","`, act as *append* operators. So what this one line of code is really saying is this: get the values of `d[0]` and `d[1]` and smush them together with a comma in the middle. The end result should be something like `5,20` or `25,67`.

Next, we specify *where* the text should be placed with x and y values. For now, let's just use `d[0]` and `d[1]`, the same values that we used to specify the `circle` positions:

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

Finally, add a bit of font styling with:

```
        .attr("font-family", "sans-serif")
        .attr("font-size", "11px")
        .attr("fill", "red");
```

The result in Figure 6-34 might not be pretty, but we got it working! See *24_scatterplot_labels.html* for the latest.

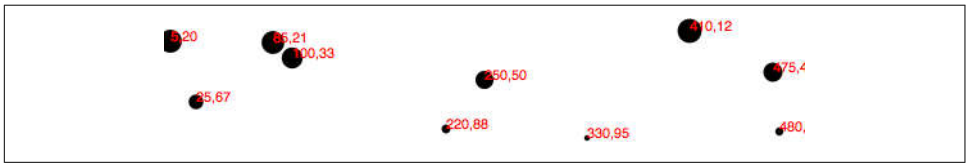

Figure 6-34. Scatterplot with labels

Next Steps

Hopefully, some core concepts of D3 are becoming clear: loading data, generating new elements, and using data values to derive attribute values for those elements.

Yet the image in Figure 6-34 is barely passable as a data visualization. The scatterplot is hard to read, and the code doesn't use our data flexibly.

Not to worry: generating a shiny, interactive chart involves taking our D3 skills to the next level. To use data flexibly, we'll learn about D3's *scales* in the next chapter. And to make our scatterplot easier to read, we'll learn about *axis generators* and axis labels.

This would be a good time to take a break and stretch your legs. Maybe go for a walk, or grab a coffee or a sandwich. I'll hang out here (if you don't mind), and when you get back, we'll jump into D3 scales!