4

Let's Curve Things Up

In this chapter we will cover:

- Building a bubble chart
- Creating a pie chart
- Using a doughnut chart to show relationships
- Leveraging a radar
- Structuring a tree chart

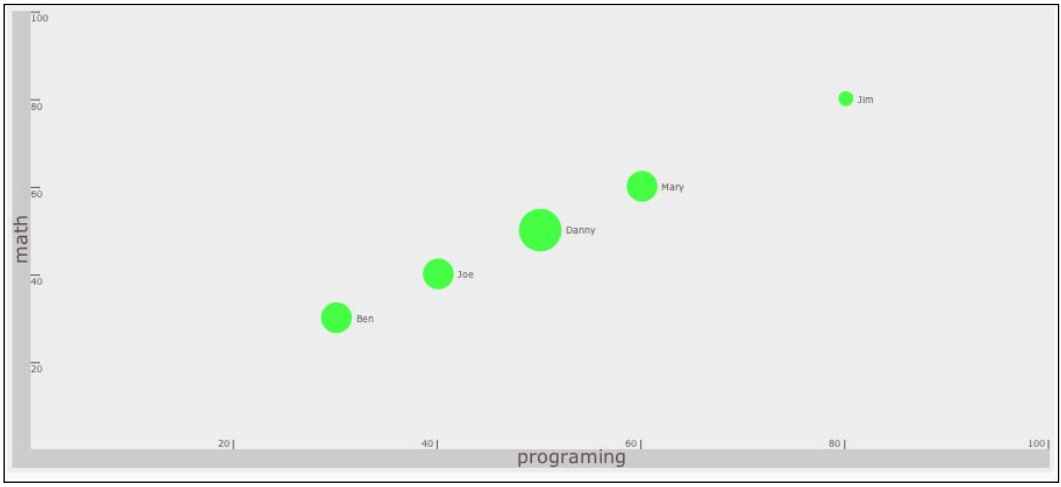
Introduction

In the last chapter, we built a component for linear graphs ranging through dots, lines, and bars. Most of the data we worked with was two-dimensional, while we ended our lesson with a four-dimensional chart. It was still represented using linear art. In this chapter, we will leverage the capability of creating non-linear data to represent data.

Building a bubble chart

Although many items in our chart will have correlations with earlier charts that we created in *Chapter 3*, *Creating Cartesian-based Graphs*, we will start from scratch. Our goal is to create a chart that has bubbles in it—the bubbles enable us to showcase data with three data points (x, y, and the size of the bubble). This type of chart is really ideal when animated as it can showcase changes over time (it could showcase many years in a few seconds).

A great demo of the powers of bubble charts can be seen in a TED presentation by Hans Rosling (`http://blog.everythingfla.com/2012/05/hans-rosling-data-vis.html`).



Getting ready

We will start up our project with a canvas setup and skip the HTML end. If you have forgotten how to create it please refer to the *Graphics with 2D Canvas* recipe in *Chapter 1*, *Drawing Shapes in Canvas*.

There are three major steps:

- Creating the data source
- Creating the background
- Adding the chart data info into the chart

How to do it...

Let's list the steps required to create a bubble chart:

1. The next data object should look familiar in an array that has objects within it with student scores in English, Math, and programming. Build the data object:

```
var students2001 = [{name:"Ben",
  math:30,
  english:60,
  programing:30},
  {name:"Joe",
  math:40,
```

```
  english:60,
  programing:40},
  {name:"Danny",
  math:50,
  english:90,
  programing:50},
  {name:"Mary",
  math:60,
  english:60,
  programing:60},
  {name:"Jim",
  math:80,
  english:20,
  programing:80}];
```

2. Create our chart information; contrary to previous charts, this chart has a third parameter for our bubble information. Define our chart rules:

```
var chartInfo= { y:{min:0, max:100,steps:5,label:"math"},
  x:{min:0, max:100,steps:5,label:"programing"},
  bubble:{min:0, max:100, minRaduis:3,
  maxRaduis:20,label:"english"}
};
```

3. The last data object will contain all the styling information that we might want to change in the future. Add a styling object:

```
var styling = { outlinePadding:4,
  barSize:16,
  font:"12pt Verdana, sans-serif",
  background:"eeeeee",
  bar:"cccccc",
  text:"605050"
};
```

4. We create an event callback when the document is ready to trigger `init`, so let's create the `init` function:

```
var wid;
var hei;
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");

  createOutline(context,chartInfo);
```

```
    addDots(context,chartInfo,students2001,
    ["math","programing","english"],"name");
}
```

5. We start creating our outline when we create our style object. Now it's time to draw everything into our canvas. So we start by setting up our base canvas style:

```
function createOutline(context,chartInfo){
  var s = styling;
  var pad = s.outlinePadding;
  var barSize = s.barSize;
  context.fillStyle = s.background;
  context.fillRect(0,0,wid,hei);
  context.fillStyle = s.bar;
  context.fillRect(pad,pad,barSize,hei-pad*2);
  context.font = s.font;
  context.fillStyle = s.text;
```

6. We need to save our current, canvas-based graphic layout information, change it to make it easier to position elements and then return it back to its original state:

```
context.save();
context.translate(17, hei/2 );
context.rotate(-Math.PI/2);
context.textAlign = "center";
context.fillText(chartInfo.y.label, 0, 0);
context.restore();

context.fillStyle = s.bar;
context.fillRect(pad+barSize,hei-pad-barSize,wid-pad*2-
barSize,barSize);
context.font = s.font;
context.fillStyle = s.text;
context.fillText(chartInfo.x.label,( wid-pad*2-barSize)/2, hei-
pad*2);

context.translate(pad+barSize,hei-pad-barSize);
context.scale(1, -1);
//SET UP CONSTANTS - NEVER CHANGE AFTER CREATED
styling.CHART_HEIGHT = hei-pad*2-barSize;
styling.CHART_WIDTH = wid-pad*2-barSize;
```

7. Now it is time to draw the outlines with the help of our `chartInfo` object:

```
var steps = chartInfo.y.steps;
var ratio;
chartInfo.y.range = chartInfo.y.max-chartInfo.y.min;
var scope = chartInfo.y.range;
```

```
context.strokeStyle = s.text;
var fontStyle = s.font.split("pt");
var pointSize = fontStyle[0]/2;
fontStyle[0]=pointSize;
fontStyle = fontStyle.join("pt");
context.font = fontStyle; // making 1/2 original size of bars
  for(var i=1; i<=steps; i++){
    ratio = i/steps;
    context.moveTo(0,ratio*styling.CHART_HEIGHT-1);
    context.lineTo(pad*2,ratio*styling.CHART_HEIGHT-1);
    context.scale(1,-1);

    context.fillText(chartInfo.y.min +
    (scope/steps)*i,0,(ratio*styling.CHART_HEIGHT-3 -
    pointSize)*-1);
    context.scale(1,-1);

  }

  steps = chartInfo.x.steps;
  chartInfo.x.range = chartInfo.x.max-chartInfo.x.min;
  scope = chartInfo.x.max-chartInfo.x.min;
  context.textAlign = "right";
  for(var i=1; i<=steps; i++){
    ratio = i/steps;
    context.moveTo(ratio*styling.CHART_WIDTH-1,0);
    context.lineTo(ratio*styling.CHART_WIDTH-1,pad*2);
    context.scale(1,-1);
    context.fillText(chartInfo.x.min +
    (scope/steps)*i,ratio*styling.CHART_WIDTH-pad,-
    pad/2);
    context.scale(1,-1);

  }

context.stroke();
}
```

8. Now it is time to add the data into our chart by creating the `addDots` method. The function `addDots` will take in the data with the definition of rules (keys) to be used, contrary to what we did in the earlier recipes.

```
function addDots(context,chartInfo,data,keys,label){
  var rangeX = chartInfo.y.range;
  var _y;
```

```
var _x;

var _xoffset=0;
var _yoffset=0;

if(chartInfo.bubble){
  var range = chartInfo.bubble.max-
  chartInfo.bubble.min;
  var radRange = chartInfo.bubble.maxRadius-
  chartInfo.bubble.minRadius;
  context.textAlign = "left";
}


for(var i=0; i<data.length; i++){
  _x = ((data[i][keys[0]] - chartInfo.x.min )/
  chartInfo.x.range) * styling.CHART_WIDTH;
  _y = ((data[i][keys[1]] - chartInfo.y.min )/
  chartInfo.y.range) * styling.CHART_HEIGHT;
  context.fillStyle = "#44ff44";


  if(data[i][keys[2]]){
    _xoffset = chartInfo.bubble.minRadius +
    (data[i][keys[2]]-chartInfo.bubble.min)/range
    *radRange;
    _yoffset = -3;
    context.beginPath();
    context.arc(_x,_y, _xoffset , 0, Math.PI*2, true);
    context.closePath();
    context.fill();

    _xoffset+=styling.outlinePadding;
  }else{
    context.fillRect(_x,_y,10,10);
  }

  if(label){
    _x+=_xoffset;
    _y+=_yoffset;
    context.fillStyle = styling.text;
    context.save();
    context.translate(_x,_y );
    context.scale(1,-1);
    context.fillText("Bluping",0,0);
```

```
            context.restore();

          }
        }
      }
```

This block of code, although redone from scratch, bears a lot of resemblance to the *Spreading data in a scatter chart* recipe in *Chapter 3*, *Creating Cartesian-based Graphs*, with modifications to enable the third level of data and the new charting format.

That's it. You should have a running bubble chart. Now when you run the application, you will see that the `x` parameter is showcasing the math score, the `y` parameter is showcasing the programming score, while the size of our bubble showcases the student's score in English.

How it works...

Let's start with the `createOutline` function. In this method, apart from the regular canvas drawing methods that we grow to love, we introduce a new style of coding where we manipulate the actual canvas to help us define our code in an easier way. The two important key methods here are as follows:

```
context.save();
context.restore();
```

We will be leveraging both the methods a few times. The `save` method saves the current view of the canvas while the `restore` method returns users to the last saved canvas:

```
context.save();
context.translate(17, hei/2 );
context.rotate(-Math.PI/2);
context.textAlign = "center";
context.fillText(chartInfo.y.label, 0, 0);
context.restore();
```

In the first use of this style, we are using it to draw our text by rotating it to the right. The `translate` method moves the `0, 0` coordinates of the canvas while the `rotate` method rotates the text using radians.

After drawing the external bars, it's time for us to use this new capability to our advantage. Most charts rely on a y coordinate that grows upwards, but this canvas has the y values growing from the top to the bottom of the canvas area. We can flip this relationship by adding some code before we loop through to add the range values.

```
context.translate(pad+barSize,hei-pad-barSize);
context.scale(1, -1);
```

In the preceding lines, we are first moving the `0,0` coordinates of our canvas to be exactly at the bottom-right range of our chart, and then we are flipping our canvas by switching the scale value. Note that from now on if we try to add text to the canvas, it will be upside down. Keep that in mind as we are now drawing in a canvas that is flipped upside down.

One thing to note in our first loop when we try to type in new text is that when we want to add text, we first undo our scale and then return back our canvas for it to be flipped:

```
context.scale(1,-1);
context.fillText(chartInfo.y.min + (scope/steps)*i,0,(ratio*styling.
CHART_HEIGHT-3 -pointSize)*-1);
context.scale(1,-1);
```

Note that we are multiplying our y coordinate by `*-1`. We are doing this because we actually want the value of our y coordinate to be negative as we have just flipped the screen.

The work around the x bar text is very similar; notice the main differences related to finding the x and y value calculations.

It's time to dig into the `addDots` function. The function will again look familiar if you've been following *Chapter 3*, *Creating Cartesian-based Graphs*, but this time we are working with a modified canvas.

We start with a few helper variables:

```
var rangeX = chartInfo.y.range;
var _y;
var _x;
var _xoffset=0;
var _yoffset=0;
```

We are adding the bubble effect dynamically, which means that this method can work even if there are only two points of information and not three. We continue by testing to see if our data object contains the bubble information:

```
if(chartInfo.bubble){
  var range = chartInfo.bubble.max-chartInfo.bubble.min;
  var radRange = chartInfo.bubble.maxRaduis-
  chartInfo.bubble.minRaduis;
  context.textAlign = "left";
}
```

If so, we add a few more variables and align our text to the left as we are going to use it in this example.

It's time for us to look through our data object and propagate the data on the chart.

```
for(var i=0; i<data.length; i++){
  _x = ((data[i][keys[0]] - chartInfo.x.min )/ chartInfo.x.range)
  * styling.CHART_WIDTH;
  _y = ((data[i][keys[1]] - chartInfo.y.min )/ chartInfo.y.range)
  * styling.CHART_HEIGHT;
  context.fillStyle = "#44ff44";
```

For each loop, we recalculate the `_x` and `_y` coordinates based on the current values.

If we have a third element, we are ready to develop a bubble. If we do not have it, we need to create a simple dot.

```
if(data[i][keys[2]]){
  _xoffset = chartInfo.bubble.minRaduis + (data[i][keys[2]]-
  chartInfo.bubble.min)/range *radRange;
  _yoffset = -3;
  context.beginPath();
  context.arc(_x,_y, _xoffset , 0, Math.PI*2, true);
  context.closePath();
  context.fill();
  _xoffset+=styling.outlinePadding;
}else{
   context.fillRect(_x,_y,10,10);
}
```

At this stage, we should have an active bubble/dot method. All that is left is for us to integrate our overlay copy.

Before we add a label, let's take a peek at the function signature:

```
function addDots(context,chartInfo,data,keys,label){}
```

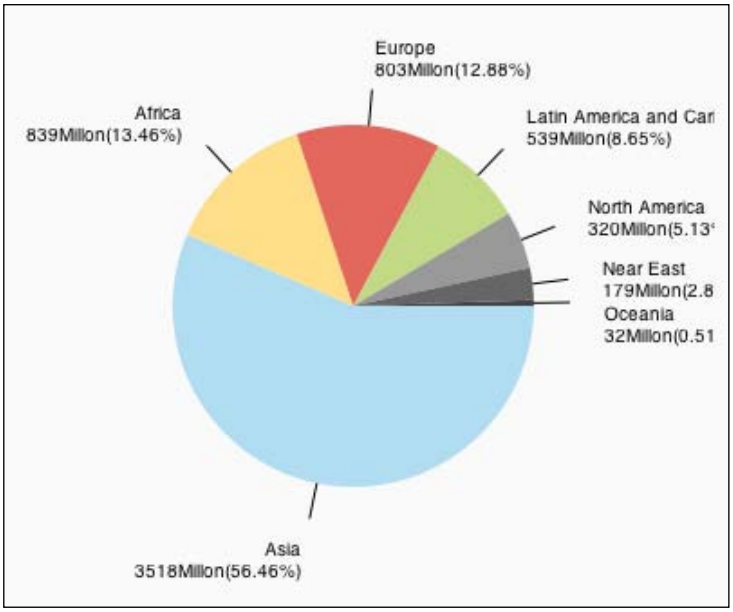
The `context` and `chartInfo` parameters are already a standard in our samples. The idea of keys was to enable us to switch what data will be tested dynamically. The keys' values are the array positions `0` and `1` that are correlated to the x and y coordinates, and position `2` is used for bubbles, as we've seen earlier. The `label` parameter enables us to send in a key value for the label. In this way, if the label is there we will add a label and if it is not there we will not.

```
if(label){
  _x+=_xoffset;
  _y+=_yoffset;
  context.fillStyle = styling.text;
  context.save();
  context.translate(_x,_y );
  context.scale(1,-1);
  context.fillText(data[i][label],0,0);
  context.restore();
 }
```

Then we add the preceding `if` statement. If our label is set, we position the style and create the text of the label.

Creating a pie chart

The steps to create a pie chart are relatively easy and short. Pie charts are ideal for showcasing a closed amount of data that we want to easily compare between data fields such as, in our example, dividing the number of people in the world into groups based on their region:



Getting ready

The first step will be to update our canvas size in the HTML area to be a rectangular area. In our sample, we will update the values to 400 x 400. That's about it; let's start building it.

How to do it...

In the following steps we will create our first pie chart. Let's get started:

1. Set up our data source and our global variables:

```
var data= [  {label:"Asia", value:3518000000,style:"#B1DDF3"},
  {label:"Africa", value:839000000,style:"#FFDE89"},
  {label:"Europe", value:803000000,style:"#E3675C"},
  {label:"Latin America and Caribbean", value:
```

```
  539000000,style:"#C2D985"},
  {label:"North America",
  value:320000000,style:"#eeeeee"},
  {label:"Near East", value:179000000,style:"#aaaaaa"},
  {label:"Oceania", value:32000000,style:"#444444"}
  ];
var wid;
var hei;
var radius = 100;
```

2. Prepare our canvas (from here on we are delving into the `init` function):

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");
…
```

3. Count the total data (world population):

```
var total=0;
for(var i=0; i<data.length; i++) total+=data[i].value;
```

4. Set up 360 degrees in radians and move our pivot point to `0,0`:

```
var rad360 = Math.PI*2;
context.translate(wid/2,hei/2);
```

5. Draw the pie chart by using the following code snippet:

```
var currentTotal=0;
  for(i=0; i<data.length; i++){
  context.beginPath();
  context.moveTo(0,0);
  context.fillStyle = data[i].style;
  context.arc( 0,0,radius,currentTotal/total*rad360,
  (currentTotal+data[i].value)/total*rad360,false);
  context.lineTo(0,0);
  context.closePath();
  context.fill();

  currentTotal+=data[i].value;
  }
}
```

That's it; we have just created a basic pie chart—I told you it would be easy!

How it works...

Our pie chart, as its name indicates, uses pies and always showcases 100 percent of data. As our arc method works based on radians, we need to convert these data points from percentile to radians.

After figuring out what the total of all the values is and the total radians in a circle (`2*PI`), we are ready to loop through and draw the slices.

```
var currentTotal=0;
  for(i=0; i<data.length; i++){
    context.beginPath();
    context.moveTo(0,0);
    context.fillStyle = data[i].style;
```

The logic is relatively simple; we loop through all the data elements, change the fill style based on the data object, and move our pointer to `0,0` (to the center of our screen as we have changed the pivot point of our canvas).

```
context.arc( 0,0,radius,currentTotal/total*rad360,(currentTotal+data
[i].value)/total*rad360,false);
context.lineTo(0,0);
context.closePath();
context.fill();

currentTotal+=data[i].value;
```

Now we draw the arc. Pay attention to the highlighted text; we start with where we left off our current total and through that we calculate the angle in radians:

```
currentTotal/total*rad360
```

We can turn this value into a percentage value that we can duplicate against the total radian of our circle. Our second parameter is very close, so we just add into it the current value of the current region we are in:

```
(currentTotal+data[i].value)/total*rad360
```

And the last point to note here is that we are setting the arc's last parameter to `false` (counter clockwise) as that works best for our calculations.

Last but not least, we update our `currentTotal` value to encompass the newly added region as that will be our starting point in the next round of our `for` loop.

There's more...

A pie chart without any information on its content is probably not going to work as well as a chart with information, but we can figure out the locations... well worry not; we are going to revisit our old friends `cos` and `sin` to help us locate the dots on our circle, to enable us to add textual information on our newly created pie.

Revisiting Math.cos() and Math.sin()

We will start with adding a new global variable to store the color of our lines and then we will call it `copyStyle`:

```
var copyStyle = "#0000000000";
```

Now that we are right back into our `init` function, let's add it into our `for` loop just before the last line:

```
currentTotal+=data[i].value;
```

As expected, we will first set our new `copyStyle` variable as our fill and stroke value:

```
context.strokeStype = context.fillStyle =  copyStyle;
```

Our next step is to locate where in our pie we would like to draw a line out so that we can add the text:

```
midRadian = (currentTotal+data[i].value/2)/total*rad360;
```

To accomplish this, we will use a new variable that will store the mid-value between the last total and the new value (the center of the new slice). So far so good. Now we need to figure out how to get the x and y positions of that point. Lucky for us, there is a very easy way of doing it in a circle by using the `Math.cos` (for the x) and `Math.sin` (for our y) functions:

```
context.beginPath();
context.moveTo(Math.cos(midRadian)*radius,Math.sin(midRadian)*radius);
context.lineTo(Math.cos(midRadian)*(radius+20),Math.
sin(midRadian)*(radius+20));
context.stroke();
```

Armed with our `midRadian` variable, we will get the value for a circle with a radius of `1`, so all that is left for us to do is duplicate that value by our real radius to find our starting point. As we want to draw a line in the same direction to the arc externally, we will find the points of an imaginary circle that is larger; so for that we are going to use the same formula, but instead upgrade our radius values by 20, creating a diagonal line that is correlative to the arc.

All that is left for us to do is figure out what text we would want to have within our chart, using the same arc point with a larger circle size:

```
context.fillText(data[i].label,Math.cos(midRadian)*(radius+40),Math.
sin(midRadian)*(radius+40));
```

Looks good... The only problem is that we don't have our values; let's add them and figure out the challenges involved with them.

Improving our bubbles' text format

In a real-world example, we would probably want to use a rollover if this was a live application (we will visit that idea in a later chapter), but let's try to figure out a way to create the chart capable of containing all the information. We stopped in our preceding line of code with a really large exterior circle (`radius+40`). Well that's because we wanted to slip in a new line of text right under, so let's do it:

```
context.fillText(formatToMillions(data[i].value) +
"(" +formatToPercent(data[i].value/total) + ")" ,Math.
cos(midRadian)*(radius+40),Math.sin(midRadian)*(radius+40) + 12);
```

It's a bit of a mouthful, but it's basically the same as the preceding line with a new line of text and an extra change, as we are shifting the y value by 12 pixels to account for the first line of text on the same area. To get this working, we are using two helper functions that format our text:

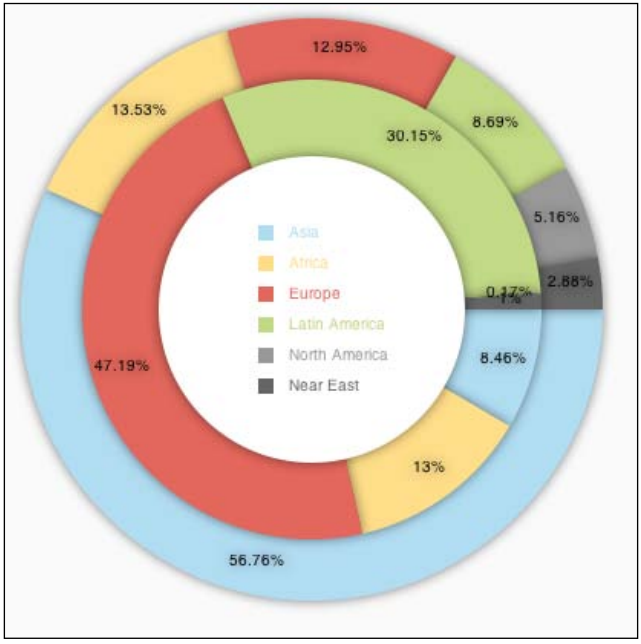
```
function formatToPercent(val){
  val*=10000;
  val = parseInt(val);
  val/=100;
  return val + "%"
}

function formatToMillions(val){
  val/=1000000;
  return val + "Million";
}
```

If you run the application in its current format, you will find that the text just doesn't look good on the page, and that is where the artist within you needs to sort things out. I've continued the sample in our source files until it felt right, so check it out or create your own variations from here on.

Using a doughnut chart to show relationships

The doughnut chart is a fancy pie chart. So if you haven't created a pie chart yet, I strongly encourage you to revisit the previous recipe, *Creating a pie chart*. A doughnut chart is a layered pie chart. This chart is ideal for condensing the comparable data between data types that would fit into a pie chart:



Getting ready

We are going to grab our code from the last example and adjust it to fit our needs. So we will start with the same HTML file and the same code from the last example.

How to do it...

Perform the following steps:

1. Let's update our data with some dummy data (we will create two data objects):

```
var data1= [ {label:"Asia", value:3518000000,style:"#B1DDF3"},
  {label:"Africa", value:839000000,style:"#FFDE89"},
  {label:"Europe", value:803000000,style:"#E3675C"},
  {label:"Latin America and Caribbean", value:
  539000000,style:"#C2D985"},
  {label:"North America",
  value:320000000,style:"#999999"},
  {label:"Near East", value:179000000,style:"#666666"}
];

var data2= [ {label:"Asia", value:151000,style:"#B1DDF3"},
  {label:"Africa", value:232000,style:"#FFDE89"},
  {label:"Europe", value:842000,style:"#E3675C"},
  {label:"Latin America and Caribbean", value:
  538100,style:"#C2D985"},
  {label:"North America", value:3200,style:"#999999"},
  {label:"Near East", value:17900,style:"#666666"}
];
```

2. Modify the `init` function by extracting all the pie-creating lines to a separate function and adding a new function `createHole` (for our doughnut):

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");
  context.translate(wid/2,hei/2);


  createPie(context,data1,190);
  createPie(context,data2,150);
  createHole(context,100);
}
```

3. Modify the pie creation to change the text layout to fit into a pie chart:

```
function createPie(context,data,radius){
  var total=0;
  for(var i=0; i<data.length;i++) total+=data[i].value;
```

```
  var rad360 = Math.PI*2;

  var currentTotal=0;
  var midRadian;
  var offset=0;
  for(i=0; i<data.length; i++){
    context.beginPath();
    context.moveTo(0,0);
    context.fillStyle = data[i].style;
    context.arc( 0,0,radius,currentTotal/total*rad360,
    (currentTotal+data[i].value)/total*rad360,false);
    context.lineTo(0,0);
    context.closePath();
    context.fill();

    context.strokeStype = context.fillStyle =  copyStyle;
    midRadian =
    (currentTotal+data[i].value/2)/total*rad360;
    context.textAlign = "center";
    context.fillText(formatToPercent(data[i].value/total)
    ,Math.cos(midRadian)*(radius-
    20),Math.sin(midRadian)*(radius-20) );

    currentTotal+=data[i].value;

  }

}
```

4. We need to create the method `createHole` (actually a simple circle):

```
function createHole(context,radius){
  context.beginPath();
  context.moveTo(0,0);
  context.fillStyle = "#ffffff";
  context.arc( 0,0,radius,0,Math.PI*2,false);
  context.closePath();
  context.fill();

}
```

That's it! We can now create an endless doughnut with as many layers as we would like by changing the radius, making it smaller each time we add a new layer.

How it works...

The core logic of the doughnut chart is the same as that of the pie chart. Our main focus is really about reformatting and rewiring the content to be outlined at the visual level. As such, part of our work is to delete the things that are not relevant and to make the needed updates:

```
context.fillText(formatToPercent(data[i].value/total),Math.
cos(midRadian)*(radius-20),Math.sin(midRadian)*(radius-20) );
```

The main thing to note is that we are hardcoding a value that is 20 less than the current radius. If we wanted our sample to work for every possible option, we would need to figure out a smarter way of generating this data as ideally we would want the text to be in between the doughnut area and rotated, but we have done things of that nature before so I'll leave that for you to explore.

There's more...

Although our doughnut is created and ready, it would help if we add some more information to it, such as outlines and a legend, as we extracted the majority of the text from the last example.

Adding an outline

We will use shadows to create a glow around our shapes. The easiest and quickest way to do it is to revisit the `init` function and add into it the shadow information to create this effect:

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");
  context.translate(wid/2,hei/2);

  context.shadowOffsetX = 0;
  context.shadowOffsetY = 0;
  context.shadowBlur    = 8;
  context.shadowColor   = 'rgba(0, 0, 0, 0.5)';

  createPie(context,data1,190);
  createPie(context,data2,150);
  createHole(context,100);

}
```

The key here is that we are setting our offset on both the x and y values to be `0`, and as such our shadow is being used as a glow. Every element that will be drawn from here on will have a shadow, and that works perfectly for us.

Creating a legend

Hey, since we have a huge hole in our doughnut, how about we put our legend right in the middle of everything? As sometimes the middle isn't exactly the best-looking thing, it will probably be best to manually figure out what is the perfect position after we create the legend.

```
context.shadowColor   = 'rgba(0, 0, 0, 0)';
context.translate(-35,-55);
createLegend(context,data1);
```

We start by removing our shadow, by setting its alpha to `0` and moving our pivot point. (I tweaked these numbers after the legend was created until I was happy.)

OK, we are ready to create our legend with the `createLegend` function:

```
function createLegend(context,data){
  context.textAlign="left";
  for(var i=0;i<data.length;i++){
    context.fillStyle=data[i].style;
    context.fillRect(0,i*20,10,10);
    context.fillText(data[i].label,20,i*20+8);
  }
}
```

We have completed a fully fledged doughnut chart with a legend.

See also

- The *Creating a pie chart* recipe

Leveraging a radar

Radars are very misunderstood charts but are really amazing. A radar enables us to showcase a really large amount of comparable data in a very condensed way. The radar chart is known as a spider chart as well.

Warning

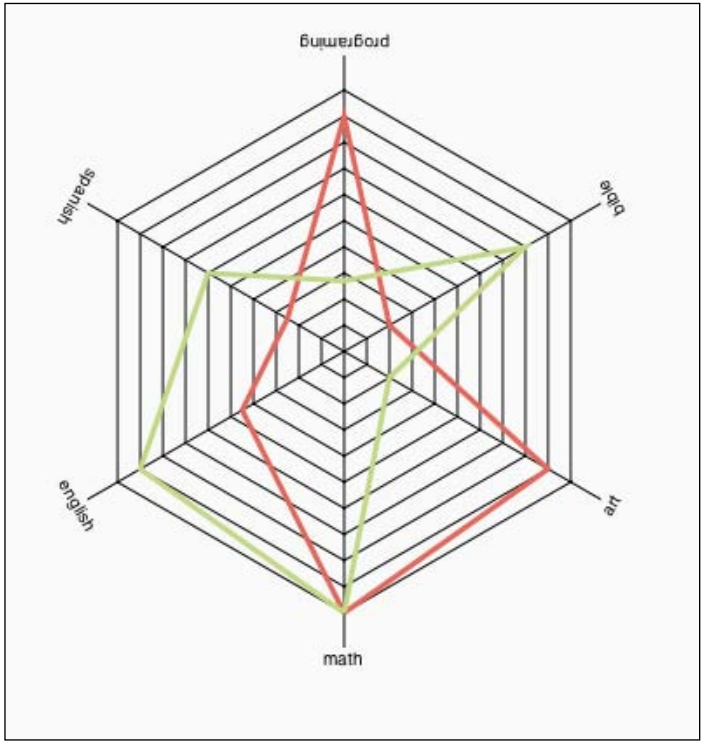
You really need to be friendly with the `Math.cos` and `Math.sin` functions, as we are going to use them plenty of times in this chart type. With that said, if you don't feel comfortable with them yet, it would be a good idea to start from the start of the chapter to refresh your memory on this.

Getting ready

As always, we are going to start with our base HTML page with an `init` callback.

Note

A radar chart is really a line chart wrapped up into a circular shape with a lot of different math involved; but it's the same idea—instead of spreading our data horizontally, we are spreading our data around a center point.



How to do it...

Let's see what are the steps involved in creating a radar chart:

1. Create/Organize the chart data and the actual data:

```
var data=[{label:"Ben", style:"#E3675C", math:90,english:45,spanis
h:25,programing:90,bible:20,art:90},
  {label:"Sharon", style:"#C2D985", math:100,english:90,spanish:60
,programing:27,bible:80,art:20}];

var chartInfo= {steps:10, max:100, types:["math","english","spanis
h","programing","bible","art"]};
```

2. Add a few helper variables and an `init` function:

```
var wid;
var hei;
var copyStyle = "#0000000000";
var radius = 180;
var radianOffset = Math.PI/2

function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");

  createSpider(context,chartInfo,data);
}
```

3. Now it is time to create the `createSpider` function:

```
function createSpider(context,chartInfo,data){
  drawWeb(context,chartInfo,radius);
  drawDataWeb(context,chartInfo,data,radius);

}
```

4. We split the creation of the radar web into two stages. The first is the lines coming out of the center of the web and the other is the actual webs that loop around this center point. Let's start with the first step and continue to the next part in the second loop:

```
function drawWeb(context,chartInfo,radius){
  chartInfo.stepSize = chartInfo.max/chartInfo.steps;
  var hSteps = chartInfo.types.length;
  var hStepSize = (Math.PI*2)/hSteps;
  context.translate(wid/2,hei/2);
```

```
    context.strokeStyle = "#000000";
      for(var i=0; i<hSteps; i++){
        context.moveTo(0,0);
        context.lineTo(Math.cos(hStepSize*i +
        radianOffset)*(radius+20),Math.sin(hStepSize*i +
        radianOffset)*(radius+20));
      }

      var stepSize = radius/chartInfo.steps;
      var cRad;

      for(var i=1; i<=chartInfo.steps; i++){
        cRad = i*stepSize;
        context.moveTo(Math.cos(radianOffset)*cRad,
        Math.sin(radianOffset)*cRad);

      for(var j=0;j<hSteps; j++){
        context.lineTo(Math.cos(hStepSize*j +
        radianOffset)*cRad,Math.sin(hStepSize*j +
        radianOffset)*cRad);
      }
      context.lineTo(Math.cos(radianOffset)*cRad,
      Math.sin(radianOffset)*cRad);

    }

    context.stroke();
}
```

5. Now it's time to integrate our data:

```
function drawDataWeb(context,chartInfo,data,radius){
  var hSteps = chartInfo.types.length;
  var hStepSize = (Math.PI*2)/hSteps;
  for(i=0; i<data.length; i++){
    context.beginPath();
    context.strokeStyle = data[i].style;
    context.lineWidth=3;
    cRad =
    radius*(data[i][chartInfo.types[0]]/chartInfo.max);
    context.moveTo(Math.cos(radianOffset)*
    cRad,Math.sin(radianOffset)*cRad);

    for(var j=1;j<hSteps; j++){
      cRad =
      radius*(data[i][chartInfo.types[j]]/chartInfo.max);
```

```
            context.lineTo(Math.cos(hStepSize*j +
            radianOffset)*cRad,Math.sin(hStepSize*j +
            radianOffset)*cRad);
          }
          cRad =
          radius*(data[i][chartInfo.types[0]]/chartInfo.max);
          context.lineTo(Math.cos(radianOffset)*
          cRad,Math.sin(radianOffset)*cRad);
          context.stroke();
        }

      }
```

Congratulations, you have just created a radar/spider chart.

How it works...

The radar chart is one of our more complicated chart types. So far it uses a lot of cos/sin functions, but the logic is very consistent and as such relatively simple.

Let's take a deeper look into the `drawWeb` method:

```
chartInfo.stepSize = chartInfo.max/chartInfo.steps;
var hSteps = chartInfo.types.length;
var hStepSize = (Math.PI*2)/hSteps;
context.translate(wid/2,hei/2);
context.strokeStyle = "#000000";
```

We start by creating a few helper variables and repositioning our pivot point to the center of the screen to help us with our calculations.

```
for(var i=0; i<hSteps; i++){
  context.moveTo(0,0);
  context.lineTo(Math.cos(hStepSize*i +
  radianOffset)*(radius+20),Math.sin(hStepSize*i +
  radianOffset)*(radius+20));
}
```

We then create our spikes based on the number of courses, as each course will be represented with a spike.

It's time to create the interwebs of our spider web now that we have our core building blocks (the spikes):

```
var stepSize = radius/chartInfo.steps;
  var cRad;
```

```
    for(var i=1; i<=chartInfo.steps; i++){
      cRad = i*stepSize;
      context.moveTo(Math.cos(radianOffset)*cRad,
      Math.sin(radianOffset)*cRad);

      for(var j=0;j<hSteps; j++){
        context.lineTo(Math.cos(hStepSize*j +
        radianOffset)*cRad,Math.sin(hStepSize*j +
        radianOffset)*cRad);
      }
      context.lineTo(Math.cos(radianOffset)*cRad,
      Math.sin(radianOffset)*cRad);

    }

  context.stroke();
```

In this multidimensional loop, we are running through step by step to draw lines from one dot on a circle to the next (from one spike point to the next), growing our radius each time we are done with creating a complete shape. Each shape we create here represents a growth by 10 in the students' score, as our students can only have scores between 0 and 100. We can ignore extreme cases in this sample. (You might need to adjust this code if your data range doesn't start at 0.)

While our `drawDataWeb` method changes, the radius based on the score assumes a range of 0 to 100. (If your ranges are not the same, you will need to modify this code, or modify your data sets to be between 0 and 100 when sent to the method.)

There's more...

Our radar isn't perfect as it could use a legend and some textual information around our radar so that we know what each bar represents. We will let you sort out a legend as we've done in the previous recipe *Using a doughnut chart to show relationships*.

Adding a rotated legend

To fix this issue and add our text, we will revisit our function `drawWeb` with our first loop in that function, and instead of updating the cos/sin values to find the rotation, we will just rotate our canvas and integrate our text at the edge each time:

```
function drawWeb(context,chartInfo,radius){
  chartInfo.stepSize = chartInfo.max/chartInfo.steps;
  var hSteps = chartInfo.types.length;
  var hStepSize = (Math.PI*2)/hSteps;
  context.translate(wid/2,hei/2);
  context.strokeStyle = "#000000";
```

```
    context.textAlign="center";
    for(var i=0; i<hSteps; i++){
    context.moveTo(0,0); context.lineTo(Math.cos(
    radianOffset)*(radius+20),Math.sin( radianOffset)*(radius+20));
    context.fillText(chartInfo.types[i],Math.cos(
    radianOffset)*(radius+30),Math.sin( radianOffset)*(radius+30));
    context.rotate(hStepSize);
  }
```

The logic here is a bit simpler as we are just rotating our canvas each time and using the exact same code over and over until the rotation comes to a full circle.

Structuring a tree chart

There are many types of trees in the virtual world, although the most intuitive one is a family tree. A family tree is a bit more complex than a basic data tree such as a class inheritance tree, as for the most part classes have only one parent while family trees usually have two.

We will build an inheritance tree for the display objects of ActionScript 3.0.

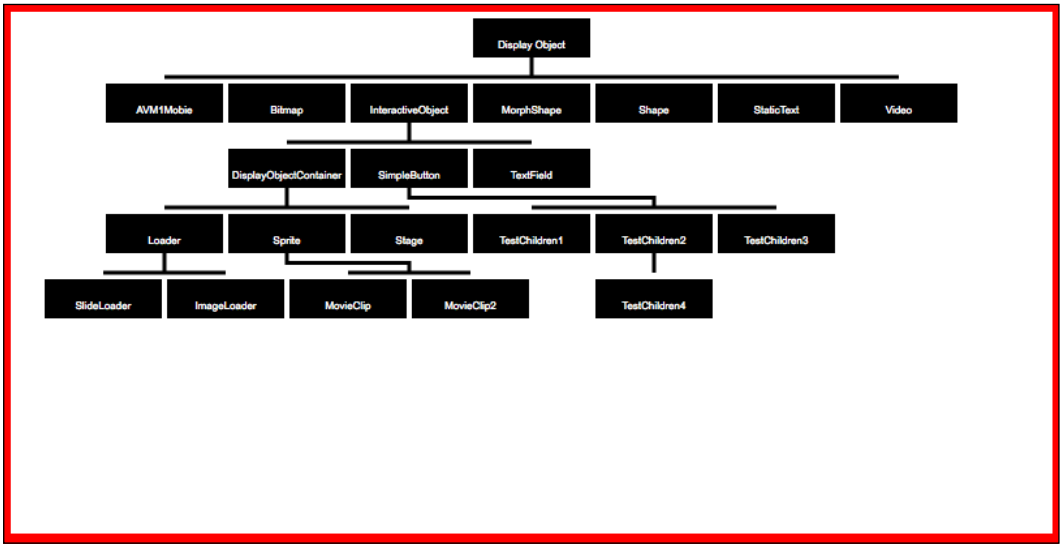
Getting ready

Please note that this sample is cutting edge in HTML5. One of the new features that no one really knows whether will get adopted or not is E4X. It's been embraced by Firefox, but not all browsers have implemented it (it is fully supported in Flash as well).

ECMAScript for XML (**E4X**) is a programming language extension that adds native XML support to ECMAScript. It has replaced the DOM interface and is implemented as a primitive (such as numbers and Booleans), making it faster and more optimized.

As we are mainly working locally, we are going to save our XML document directly in our JavaScript to avoid sandbox security issues.

To help us space out our elements, we will make our canvas area much larger in this sample (800 x 400). Alright, let's start implementing our tree sample created with E4X.



How to do it...

Perform the following steps:

1. We will start by creating our XML object that contains our class tree (please note that this will only work on an up-to-date version of Firefox as at the time of this book being written):

```
var xml = <node name ="Display Object">
<node name="AVM1Mobie" />
  <node name="Bitmap" />
  <node name="InteractiveObject" >
  <node name="DisplayObjectContainer">
  <node name="Loader" />
  <node name="Sprite" >
  <node name="MovieClip"/>
</node>
<node name="Stage" />
</node>
<node name="SimpleButton" />
  <node name="TextField" />
</node>
  <node name="MorphShape" />
  <node name="Shape" />
```

```
  <node name="StaticText" />
  <node name="Video" />
</node>;
```

2. Then create our standard helper and styling objects:

```
var wid;
var hei;
var style = {boxWidth:90,boxHeight:30, boxColor:"black",boxCopy:"w
hite", boxSpace:4, lines:"black",lineSpace:30 };
```

3. We will implement our `init` function and then call the `drawTree` function:

```
function init(){
  var can = document.getElementById("bar");
  wid = can.width;
  hei = can.height;
  var context = can.getContext("2d");
  context.textAlign = "center";
  context.font = "6pt Arial";
  drawTree(context,wid/2,20, xml );
}
```

4. Time to implement the `drawTree` function (our recursive function):

```
function drawTree(context,_x,_y,node){
  context.fillStyle=style.boxColor;
  context.fillRect(_x-style.boxWidth/2,_y-
  style.boxHeight/2,style.boxWidth,style.boxHeight);
  context.fillStyle=style.boxCopy;
  context.fillText(node.@name,_x,_y+8);

  if(node.hasComplexContent()){
    var nodes = node.node;
    var totalWidthOfNewLayer = nodes.length()*
    style.boxWidth;
    if(nodes.length()>1)totalWidthOfNewLayer += (
    nodes.length()-1)* style.boxSpace;
    var startXPoint = _x-totalWidthOfNewLayer/2 +
    style.boxWidth/2;
    var currentY = _y+style.boxHeight/2;

    context.beginPath();
    context.strokeStyle ="#000000";
    context.lineWidth=3;
    context.moveTo(_x,currentY);
    currentY+=style.lineSpace/2;
    context.lineTo(_x,currentY);
```

```
            context.moveTo(startXPoint,currentY);
            context.lineTo(startXPoint+totalWidthOfNewLayer-
            style.boxWidth,currentY);
            context.stroke();

            for(var i=0; i<nodes.length();i++){
              drawTree(context,startXPoint + i*(style.boxWidth +
              style.boxSpace) ,_y+50,nodes[i]);
            }
          }
        }
```

Tah Dah! We just created our first tree.

How it works...

For more information on how E4X works, I recommend checking out some online resources such as `http://goo.gl/jLWYd` and `http://goo.gl/dsHD4`.

Let's take a deeper look at how our recursive `drawTree` works. The basic idea of `createTree` is to create the current node in focus and to check if the node has children; if it does, to send them to the `drawTree` and have them recursively continue until all the children are created and done. One of the most critical things you need to worry about when creating a recursive function (a function that calls itself) is to make sure that it doesn't end up being endless, and as our scenario has a very defined end that is based on the XML structure, we are safe.

We start by creating the current node in focus, based on the point values that were sent over in our function's parameters:

```
context.fillStyle=style.boxColor;
context.fillRect(_x-style.boxWidth/2,_y-style.boxHeight/2,style.
boxWidth,style.boxHeight);
context.fillStyle=style.boxCopy;
context.fillText(node.@name,_x,_y+8);
```

Right after these lines is where it starts getting really interesting. If our node is complex, we are going to assume that it has children, as that's our base rule in creating our XML object; and if so, it's time for us to draw the children:

```
if(node.hasComplexContent()){
```

We start by drawing a visualizer bar to help us see what the children of the current element are, and create a few helper variables in the process:

```
var nodes = node.node;
var totalWidthOfNewLayer = nodes.length()* style.boxWidth;
if(nodes.length()>1)
```

```
    totalWidthOfNewLayer += ( nodes.length()-1)* style.boxSpace;

    var startXPoint = _x-totalWidthOfNewLayer/2 +
    style.boxWidth/2;
    var currentY = _y+style.boxHeight/2;

    context.beginPath();
    context.strokeStyle ="#000000";
    context.lineWidth=3;
    context.moveTo(_x,currentY);
    currentY+=style.lineSpace/2;
    context.lineTo(_x,currentY);
    context.moveTo(startXPoint,currentY);
    context.lineTo(startXPoint+totalWidthOfNewLayer-
    style.boxWidth,currentY);
  context.stroke();
```

After creating our outline helper lines, it's time for us to loop through the children and send them to `drawTree` with their new positions:

```
  for(var i=0; i<nodes.length();i++){
    drawTree(context,startXPoint + i*(style.boxWidth +
    style.boxSpace) ,_y+50,nodes[i]);
    }
  }
```

That covers all the logic. At this stage, the logic will start all over again for each element, one at a time.

There's more...

In a perfect world our work with our tree would be done by now, but many a time in real-world scenarios we would encounter issues. If we play with our current tree enough, we will discover visual issues, such as if a child node has more than one child, its children will overlap the other tree branches. For example, if we update our `Loader` class to have two new children (two dummy classes just for the sake of our example):

```
  var xml = <node name ="Display Object">
    <node name="AVM1Mobie" />
    <node name="Bitmap" />
    <node name="InteractiveObject" >
    <node name="DisplayObjectContainer">
    <node name="Loader">
    <node name="SlideLoader"/>
    <node name="ImageLoader"/>
    </node>
```

```
    <node name="Sprite" >
    <node name="MovieClip"/>
    <node name="MovieClip2"/>
    </node>
    <node name="Stage" />
    </node>
    <node name="SimpleButton" />
    <node name="TextField" />
    </node>
    <node name="MorphShape" />
    <node name="Shape" />
    <node name="StaticText" />
    <node name="Video" />
</node>;
```

If you refresh your browser (currently only Firefox), you will see that our elements are overlapping as we didn't take into account the option of children that have children. If we review our code more deeply, we will see that in the current logic format there is no way to solve the problem as the creation of the children is happening separately. We will need to figure out a way to manage lines so that our elements will have a way to know that they are about to overlap.

To solve this problem, we will need to make our recursive function more complex, as it will need to keep track of its children's x position so that it can offset whenever there is an overlap. Please review the modified code (changes marked in bold):

```
function drawTree(context,_x,_y,node,nextChildX){
  context.fillStyle=style.boxColor;
  context.fillRect(_x-style.boxWidth/2,_y-
  style.boxHeight/2,style.boxWidth,style.boxHeight);
  context.fillStyle=style.boxCopy;
  context.fillText(node.@name,_x,_y+8);

  if(node.hasComplexContent()){
    var nodes = node.node;
    var totalWidthOfNewLayer = nodes.length()* style.boxWidth;
    if(nodes.length()>1)totalWidthOfNewLayer += ( nodes.length()-
    1)* style.boxSpace;
    var startXPoint = _x-totalWidthOfNewLayer/2 +
    style.boxWidth/2;
    var currentY = _y+style.boxHeight/2;

    context.beginPath();
    context.strokeStyle ="#000000";
    context.lineWidth=3;
    context.moveTo(_x,currentY);
```

```
    if(nextChildX>startXPoint){
      currentY+=style.lineSpace/4;
      context.lineTo(_x,currentY);
      context.lineTo(_x + (nextChildX-startXPoint),currentY);

      currentY+=style.lineSpace/4;
      context.lineTo(_x + (nextChildX-startXPoint),currentY);
      startXPoint = nextChildX; // offset correction value
    }else{
      currentY+=style.lineSpace/2;
      context.lineTo(_x,currentY);
    }
    context.moveTo(startXPoint,currentY);
    context.lineTo(startXPoint+totalWidthOfNewLayer-
    style.boxWidth,currentY);
    context.stroke();
    var returnedNextChildX=0;
    for(var i=0; i<nodes.length();i++){
      returnedNextChildX = drawTree(context,startXPoint +
      i*(style.boxWidth + style.boxSpace)
      ,_y+50,nodes[i],returnedNextChildX);
    }
    return startXPoint + i*(style.boxWidth + style.boxSpace);
  }

  return 0;
}
```

Wow that looks complicated—it's because it is! So let's break the logic down.

The idea is simple, but as for every simple idea, sometimes it's harder to visualize after it's implemented. The idea is that every time we create a new tree element, we will return `0` if it has no children, and if it has children, we will send back the next free position for future children. We added a fourth parameter to the function as well, and we sent that information each time we looped through children. That way each child is aware of where the last child left off. If an element's real position can't be worked out, we draw a redirect line as per the amount of the offset and update `startXPoint`. Take a deeper look at this (so far my favorite code in the cookbook), which was fun to figure out!