# D3.js Scales

## The Goal

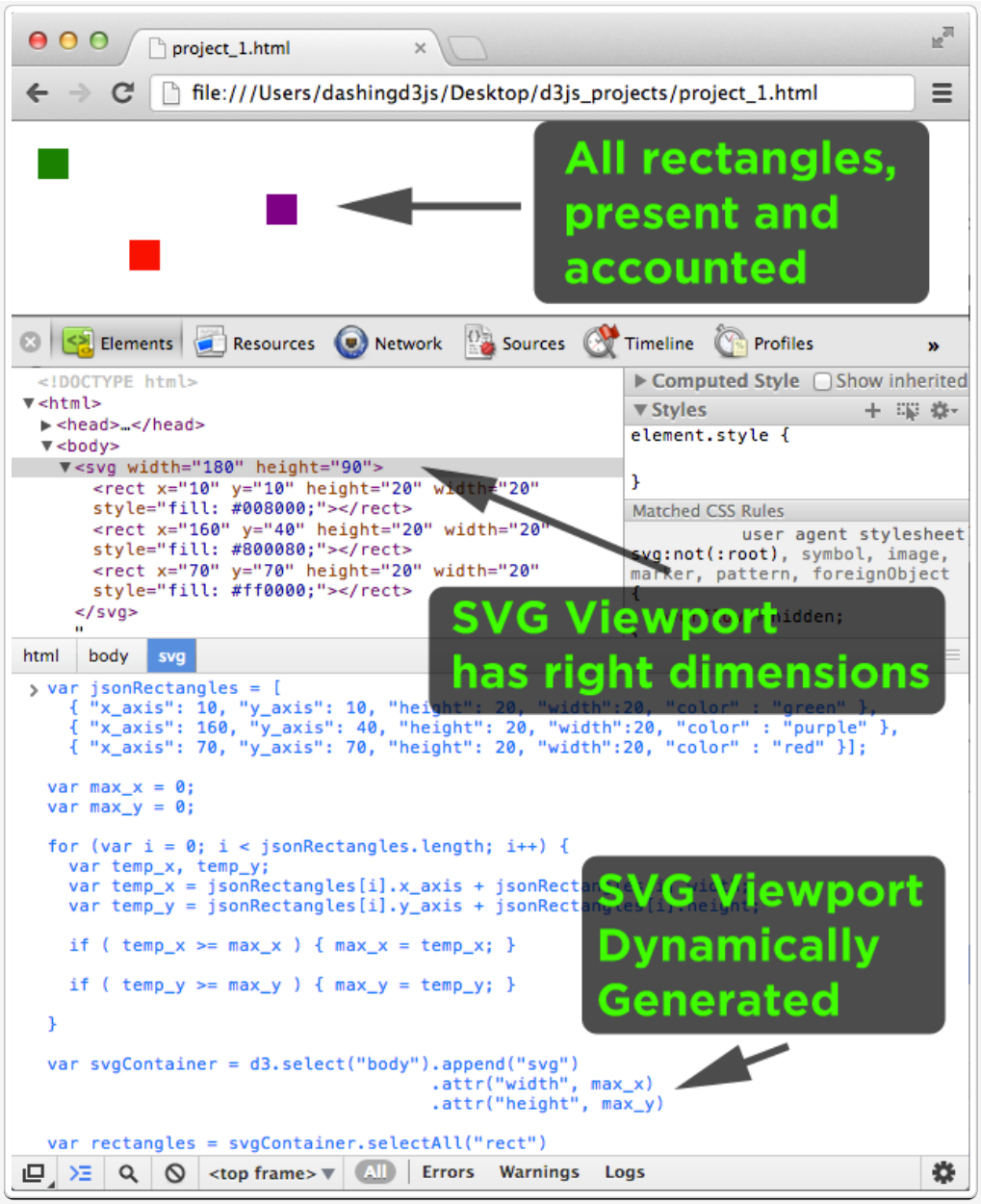
In this section, we will cover D3.js Scales so that instead of resizing our SVG Coordinate space to our data, we can resize our data to fit into our pre-defined SVG Coordinate Space.

## Previous Example of Three Rectangles

At the end of the Dynamic SVG Coordinate Space section, we created, styled, and updated our SVG coordinate space to draw three rectangles using D3.js

```
var jsonRectangles = [
  { "x_axis": 10, "y_axis": 10, "height": 20, "width":20, "color" : "green" },
  { "x_axis": 160, "y_axis": 40, "height": 20, "width":20, "color" : "purple" },
  { "x_axis": 70, "y_axis": 70, "height": 20, "width":20, "color" : "red" }];

var max_x = 0;
var max_y = 0;

for (var i = 0; i < jsonRectangles.length; i++) {
  var temp_x, temp_y;
  var temp_x = jsonRectangles[i].x_axis + jsonRectangles[i].width;
  var temp_y = jsonRectangles[i].y_axis + jsonRectangles[i].height;

  if ( temp_x >= max_x ) { max_x = temp_x; }

  if ( temp_y >= max_y ) { max_y = temp_y; }
}

var svgContainer = d3.select("body").append("svg")
                                    .attr("width", max_x)
                                    .attr("height", max_y)

var rectangles = svgContainer.selectAll("rect")
                             .data(jsonRectangles)
                             .enter()
                             .append("rect");

var rectangleAttributes = rectangles
                          .attr("x", function (d) { return d.x_axis; })
                          .attr("y", function (d) { return d.y_axis; })
                          .attr("height", function (d) { return d.height; })
                          .attr("width", function (d) { return d.width; })
                          .style("fill", function(d) { return d.color; });
```

Which gave us:




Which was great - we scaled our SVG Coordinate Space up to include the data.

However, what if our data attributes suddenly quadrupled. And then, quadrupled again. And then....etc.

As the data attributes grow, our SVG Coordinate Space will grow as well.

This is a problem once the SVG Viewport/Coordinate Space is bigger than the browser window.

At some point, the size is too big to scroll through and it becomes incredibly easy to get lost.

Imagine if our new data became:

```
var jsonRectangles = [
  { "x_axis": 10, "y_axis": 10, "height": 20, "width":20, "color" : "green" },
  { "x_axis": 16000000000, "y_axis": 40, "height": 20, "width":20, "color" : "purple" },
  { "x_axis": 70, "y_axis": 70000000000000, "height": 20, "width":20, "color" : "red" }];
```

The purple rectangle would be so far to the right that it would be practically impossible to see.

The red rectangle would be so far down that it would be practically impossible to see as well.

In order to keep our SVG Viewport within the browser window, we can scale our data to fit into the space allotted.

Which is where D3.js Scales come in.

## D3.js Scales

D3.js provides functions to perform data transformations.

These functions map an input **domain** to an output **range**.

Said another way, these functions take an interval and transform it into a new interval.

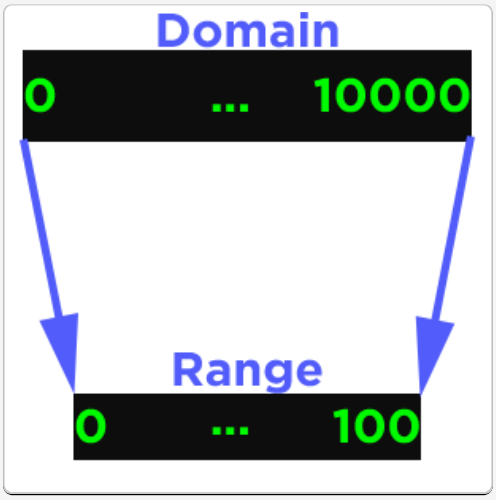
Because the D3.js Scales are functions, not only can we map one input domain to an output range, the functions can convert a number in the domain to an output in the range.

## A Numerical Example

Let us say that our input data will be some number between (and including) 0 and 10,000.

We want to transform this into an interval of numbers between (and including) 0 and 100.

Visually, we want to do the following:




We take an interval (called Domain by D3.js) and transform it into a new interval (called Range by D3.js).

One possible reason for doing this is that we might have an SVG Viewport whose width is 100, so we need to scale our data down to fit into the window.

To do the transformation, we can follow this algorithm:

1. Figure out what the largest number in the original interval is ( -> 10000 )
2. Figure out what the smallest number in the original interval is ( -> 0)
3. Figure out the difference between the two original interval numbers ( -> 10000 - 0 = 10000 )
4. Figure out what the largest number in the new interval is ( -> 100 )
5. Figure out what the smallest number in the new interval is ( -> 0 )
6. Figure out the difference between the two new interval numbers ( -> 100 - 0 = 100 )
7. Divide the original interval difference between the new interval difference ( -> 10000 / 100 = 100 )
8. This tells us that 100 units of the original interval are equal to 1 unit of the new interval
9. This is called a linear scaling (y = mx + b , where b=0 and m = 1/100)

In our example, if x = 10000, we divide it by 100 to get y = 100

Per above, our initial data comes in as follows:

```
//Initial Data
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];
```

To transform it to the new interval, we can manually divide each number by 100:

```
//New Data, scaled by 100 (initial data with every element manually divided by 100)
var scaledByOneHundredData = [0, 10, 30, 20, 50, 40, 70, 60, 90, 80, 100];
```

It is painful to divide by 100 and then type out the new answer every time.

So we can code it up in JavaScript:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var scaledByOneHundredData = [];

for (var i = 0; i < initialScaleData.length; i++) {
  scaledByOneHundredData[i] = initialScaleData[i] / 100;
}

scaledByOneHundredData;
//[0, 10, 30, 20, 50, 40, 70, 60, 90, 80, 100]
```

Which is all well and good, except for one thing: **lots of manual work**.

What we would rather do is tell the computer that the initial interval (domain) is 0 to 10,000 and that the new interval (range) should be 0 to 100.

Then when a data point comes in, we want the computer to automatically convert it for us.

## D3.js Scale Linear

D3.js can do this automatically for us using the D3.js Scale Linear Function:

```
var identityScale = d3.scale.linear();
```

This constructs a new linear scale with the default domain [0,1] to range [0,1] which produces a mapping of 1:1.

This default linear scale acts like the identity function for numbers

```
var identityScale = d3.scale.linear();

identityScale(1);
// 1

identityScale(2);
// 2
```

We can override the default domain by specifying the domain using chained syntax.

```
var domainOnlyScale = d3.scale.linear()
                              .domain([0,10000]);
```

As we did not change the range, we are mapping the domain of 0 to 10000 onto 0 to 1.

Which shows up as:

```
var domainOnlyScale = d3.scale.linear()
                                .domain([0,10000]);

domainOnlyScale(1);
// 0.0001

domainOnlyScale(2);
// 0.0002
```

We can override the default range by specifying the range using chained syntax.

```
var linearScale = d3.scale.linear()
                          .domain([0,10000])
                          .range([0,100]);
```

Which shows up as:

```
var linearScale = d3.scale.linear()
                           .domain([0,10000])
                           .range([0,100]);

linearScale(1);
//0.01

linearScale(10);
// 0.1

linearScale(100);
// 1

linearScale(1000);
// 10

linearScale(10000);
// 100
```

We can then code it into JavaScript:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var newScaledData = [];

var linearScale = d3.scale.linear()
                    .domain([0,10000])
                    .range([0,100]);

for (var i = 0; i < initialScaleData.length; i++) {
  newScaledData[i] = linearScale(initialScaleData[i]);
}

newScaledData;
//[0, 10, 30, 20, 50, 40, 70, 60, 90, 80, 100]
```

Bingo! **Much less manual work!**

We were able to bypass the step of figuring out how to map our initial interval (domain) onto our new interval (range).

## D3.max

In the initial data:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];
```

We already knew that the max possible value of the data was 10,000.

This allowed us to enter the domain as [0,10000].

**What if** we didn't know what the max value of our data was going to be?

D3.js has a function (D3.max) which calculates the maximum value of an array.

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var maxInitialData = d3.max(initialScaleData);
// 10000
```

Which means that we could re-write the JavaScript above to be:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var newScaledData = [];

var linearScale = d3.scale.linear()
                          .domain([0,d3.max(initialScaleData)])
                          .range([0,100]);

for (var i = 0; i < initialScaleData.length; i++) {
  newScaledData[i] = linearScale(initialScaleData[i]);
}

newScaledData;
//[0, 10, 30, 20, 50, 40, 70, 60, 90, 80, 100]
```

Now, regardless of how big the biggest data point ends up being, we know that the max value will help us with the domain mapping.

## D3.min

In the initial data:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];
```

We already knew that the min possible value of the data was 0.

This allowed us to enter the domain as [0,10000].

**What if** we didn't know what the min value of our data was going to be?

D3.js has a function (D3.min) which calculates the minimum value of an array.

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var minInitialData = d3.min(initialScaleData);
// 0
```

Which means that we could re-write the JavaScript above to be:

```
var initialScaleData = [0, 1000, 3000, 2000, 5000, 4000, 7000, 6000, 9000, 8000, 10000];

var newScaledData = [];
var minDataPoint = d3.min(initialScaleData);
var maxDataPoint = d3.max(initialScaleData);

var linearScale = d3.scale.linear()
                          .domain([minDataPoint,maxDataPoint])
                          .range([0,100]);

for (var i = 0; i < initialScaleData.length; i++) {
  newScaledData[i] = linearScale(initialScaleData[i]);
}

newScaledData;
//[0, 10, 30, 20, 50, 40, 70, 60, 90, 80, 100]
```

Now, regardless of how small the smallest data point ends up being, we know that the min value will help us with the domain mapping.

## Other D3.js Scales

D3.js comes with Quantitative Scales (one of which we've already covered - Linear) and Ordinal Scales.

The Quantitative scales have a continuous domain such as dates, times, real numbers, etc...

The Ordinal scales are for discrete domains - like names, categories, colors, etc...

The D3.js scales are:

1. Identity: a special kind of linear scale, 1:1, good for pixel values. input == output
2. Linear: transforms one value in the domain interval into a value in the range interval
3. Power and Logarithmic scales: sqrt, pow, log – used for exponentially increasing values
4. Quantize and Quantile scales: for discrete sets of unique possible values for inputs or outputs
5. Ordinal: for non quantitative scales, like names, categories, etc.

We will stick with the Linear Scale for a while as we cover the next few sections.

Using the D3.js Linear Scales, we will now be able to resize our data to fit into our pre-defined SVG Coordinate Space, rather than resizing our SVG Coordinate space to fit our data.



← Dynamic SVG Coordinate Space

SVG Group Element and D3.js →

**Learn D3.js**

- D3 Tutorial
- D3 Screencasts
- D3 Mapping Training
- D3 Introductory Training
- D3 Intermediate Training
- D3 Advanced Training
- D3 Corporate Training

**DashingD3js.com**

- Blog
- About
- Hire Me
- D3 Examples
- D3 Resources
- D3 & Data Viz Newsletter Archive

**Data Visualization & D3.js Weekly Newsletter**

Get D3.js and Data Visualization news, articles, jobs and more delivered to your inbox every Tuesday:

E-mail

Get the Newsletter

Did you sign up for the newsletter? :)