Visual Analytics

2– D3: Scales & SVG

A.A. 2020/21

OUTLINE

- **Recap**
- **Scales**
- **Axes**
- **SVG Shapes**
- **Example**

RECAP: Selections

- A **selection** is an array of elements pulled from the current document.

#d3.**select**(selector)

- Selects the first element that matches the specified selector string, returning a single-element selection.

#d3.**selectAll**(selector)

- Selects all elements that match the specified selector.

```
<script>
    function changeColor(){
        var reds=d3.selectAll(".red")
            .attr("class","green");
        var blue=d3.select("#blue")
            .attr("fill","yellow");
    }
</script>
```

RECAP: Data

Data ↦ Elements

- D3 doesn't provide a single method for creating multiple elements.
- it provides a pattern for managing the mapping from data to elements. The way to create elements from scratch is a special case of the more generalized form.
- The **data** method does all the work. The **enter** and **exit** methods just return the subselections computed in the join.

```
var data = [1, 1, 2, 3, 5, 8];
var circle=svg.selectAll ("circle");
                .data(data)
```

RECAP: Enter, Exit

- **Enter**

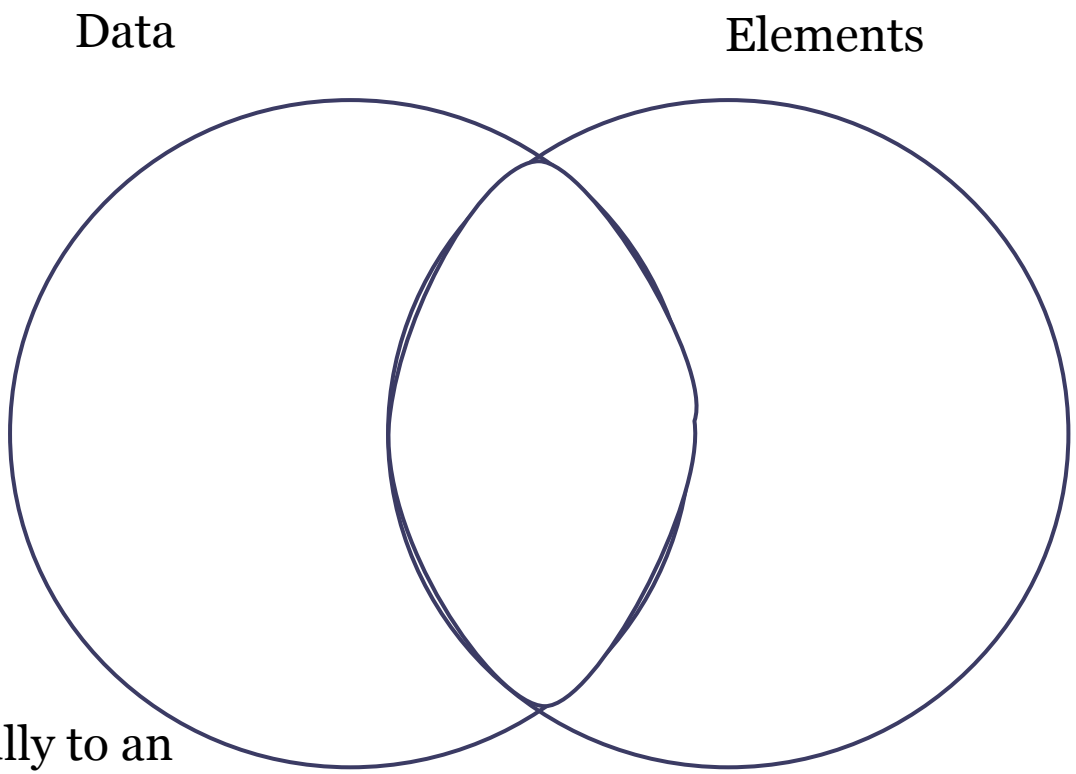
  New data, for which there were no existing elements.

- **Exit**

  Existing elements, for which there were no new data.

- **Update**

  New data that was joined successfully to an existing element.



RECAP: Append

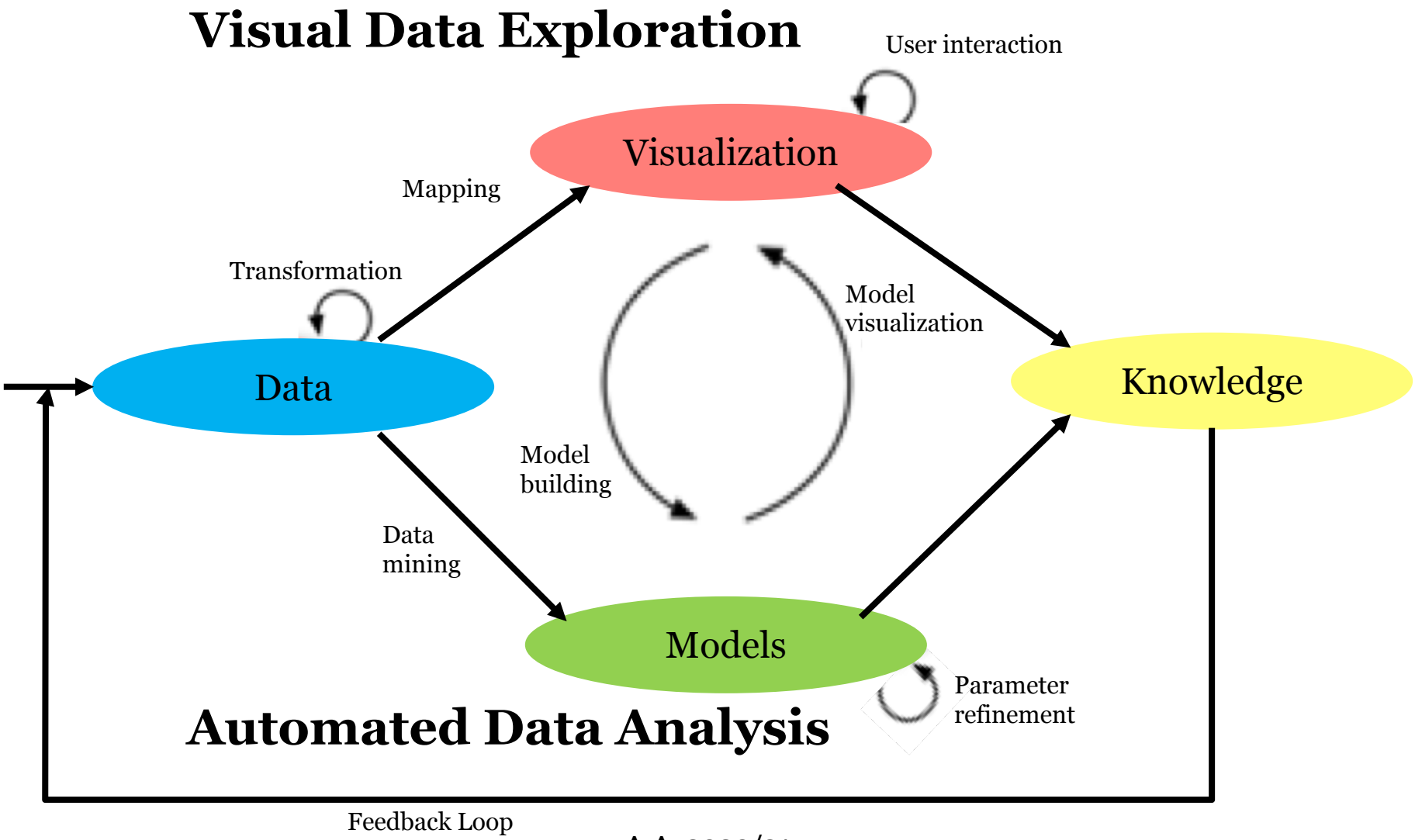
- With one element selected, adds one element:

```
var body = d3.select("body");
var h1 = body.append("h1");
h1.text("Hello!");
```

- With many elements selected, adds one element to each:

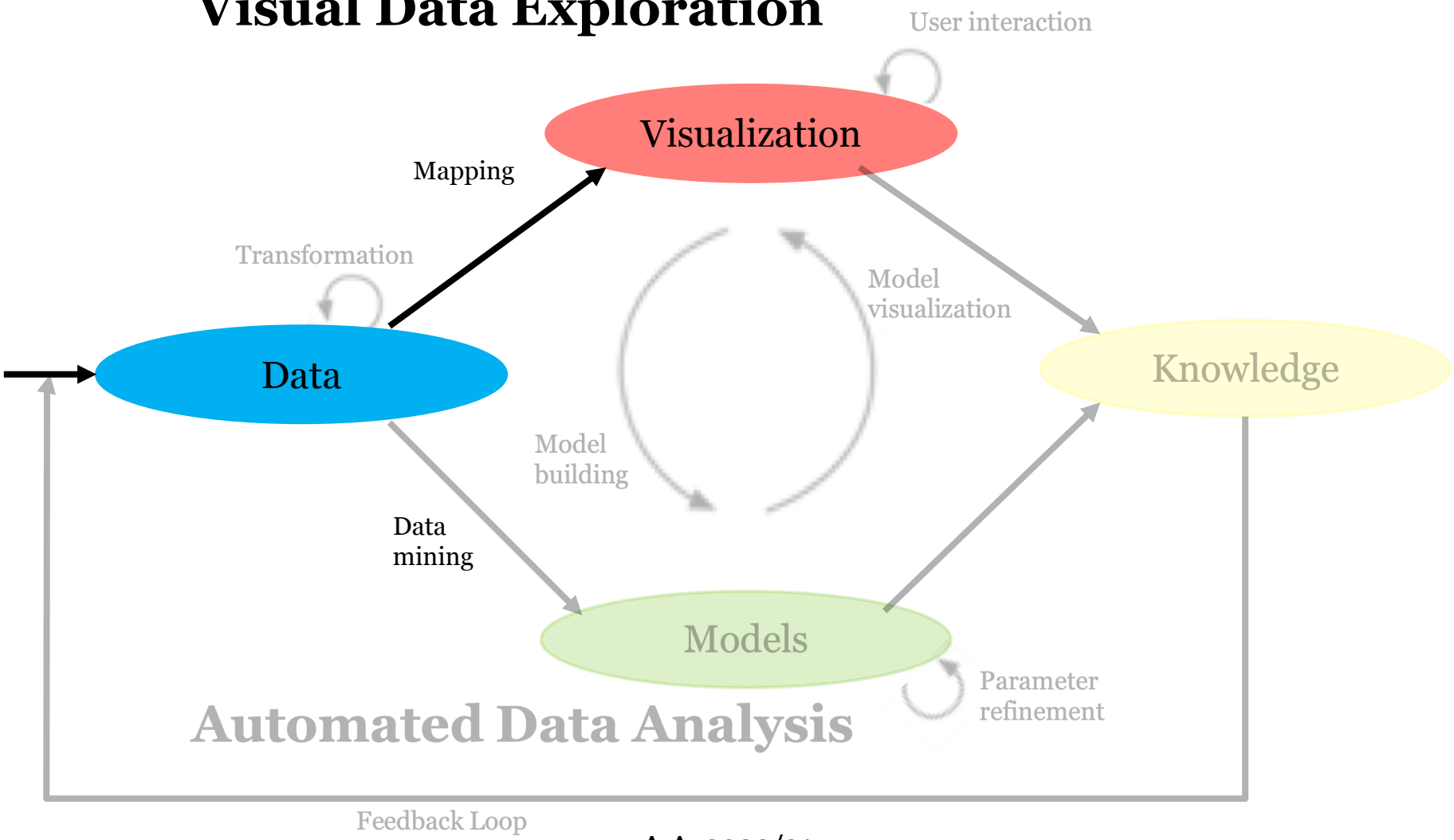
```
var section = d3.selectAll("section");
var h1 = section.append("h1");
h1.text("Hello!");
```

Visual Analytics



Towards Visual Analytics

Visual Data Exploration



OUTLINE

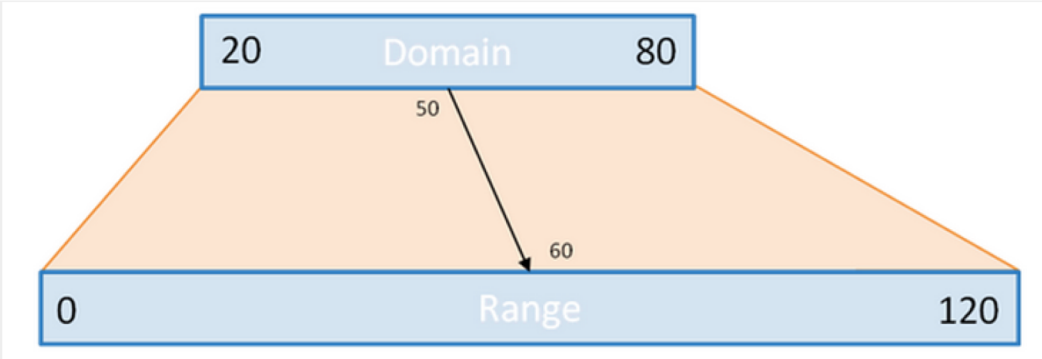
- Recap
- **Scales**
- Axes
- SVG Shapes
- Example

SCALES

Data ↦ Attributes

- **Attributes** (and styles) control position and appearance.
- We use the **data** join to maintain the mapping from data to elements, but what about the mapping from data to attributes?
- We need to compute attributes from data to generate visual encodings (position, color, *etc.*).
- **Scales** help with this.

SCALES



- The two key-concepts are the ones about **Domain** and **Range** of a Scale.
- Scales are functions that map from data-space (**Domain**) to visual-space (**Range**).
- Some scales can also go the other way and compute an inverse mapping from range to domain (the scale.invert method); this is useful for interaction.
- Despite the fact that Scales are optional (you can roll your own), they're a best practice in using D3 library.

SCALES: Quantitative

- A Quantitative scale maps a **continuous** (numeric) domain to a **continuous** range.
- **LINEAR scale**:

```
var x = d3.scaleLinear()        x(16); // 240
        .domain([12, 24])
        .range([0, 720]);
```

- **SQRT scale**:

```
var x = d3.scaleSqrt()          x(16); // 268.9056992603583
                .domain([12, 24])
                .range([0, 720]);
```

- **LOG scale**:

```
var x = d3.scaleLog()           x(16); // 298.82699948076737
          .domain([12, 24])
          .range([0, 720]);
```

SCALES: Domains & Ranges

- Typically, domains are derived from data while ranges are constant.

- Use d3.min and d3.max to compute the domain:

```
var x = d3.scaleLinear()
                    .domain([0, d3.max(numbers)])
                    .range([0, 720]);
```

- Use d3.extent to compute the min and max simultaneously:

```
var x = d3.scaleLog()
                    .domain(d3.extent(numbers))
                    .range([0, 720]);
```

SCALES: accessors

- If you have an array of objects as input data (in contrast with plain numbers…), use an **accessor function** to derive a numeric value for each single object:

1. Define the accessor for particular field. Because you're referring to data, use always the d parameter , a reference to the actual object processed:

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

2. Use the accessor function just defined in the definition of domain:

```
var x = d3.scaleLog()
            .domain(d3.extent(objects, value))
            .range([0, 720]);
```

SCALES: Interpolators

- Quantitative scales support multiple interpolators
- …and also transitions.

- Colors are detected automatically for RGB interpolation:

```
var x = d3.scaleLinear()
                  .domain([12, 24])
                  .range(["steelblue", "brown"]);

x(16); // #666586
```

- String interpolation matches embedded numbers; useful for CSS positions and sizes:

```
var x = d3.scaleLinear()
                  .domain([12, 24])
                  .range(["0px", "720px"]);

x(16); // 240px
```

SCALES: Compund scale

- Sometimes, you may want a compound (“polylinear”) scale.

```
var x = d3.scaleLinear()
                  .domain([-10, 0, 100])
                  .range(["red", "white", "green"]);

x(-5); // #ff8080
x(50); // #80c080
```

- The domain and range can have more than two values.

SCALES: Ordinal

- Map a discrete domain to a discrete range:

```
var x = d3.scaleOrdinal()
                    .domain(["A", "B", "C", "D"])
                    .range([0, 10, 20, 30]);

x("B"); // 10
```

- An ordinal scale is essentially an **explicit mapping.**

- Ordinal scales are often used to assign categorical colors:

```
var x = d3.scaleOrdinal(schemecategory20)
                    .domain(["A", "B", "C", "D"]);

x("B"); // #aec7e8
x("E"); // #2ca02c
x.domain(); // A, B, C, D, E
```

Unknown values are implicitly added to the domain.

SCALES: Ordinal

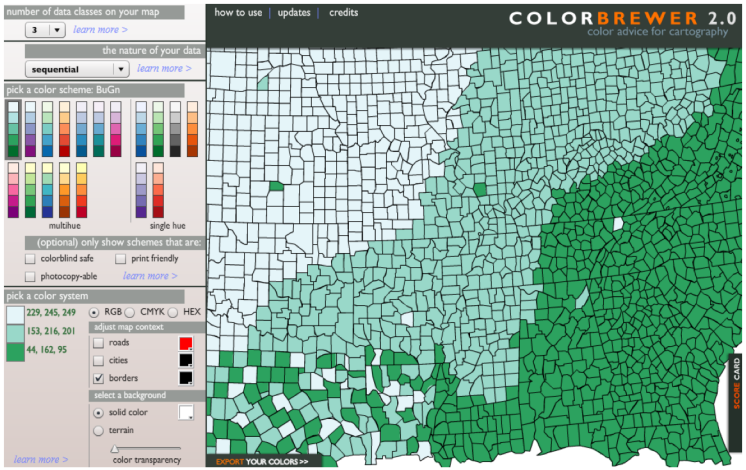
- A handful of color scales are built-in:

- For color scale and other issue
- regarding color spaces check also:

  http://colorbrewer2.org/

  - Color schemes
  - Colorblind
  - Others……

- Very useful export functions…



Sequential scales

d3.**scaleSequential**(*interpolator*)

- Constructs a new sequential scale with the given *interpolator* function.
- When the scale is applied, the interpolator will be invoked with a value typically in the range [0, 1], where 0 represents the start of the domain, and 1 represents the end of the domain.

Examples of predefined interpolators:

d3.**interpolateViridis**(*t*)

d3.**interpolateRainbow**(*t*)

SCALES: Ordinal

- Ordinal ranges can be derived from continuous ranges:

```
var x = d3.scalePoint()
                    .domain(["A", "B", "C", "D"])
                    .range([0, 720]);

x("B"); // 240
```

- The rangeRoundBands method is the same as rangeBands, except it rounds to whole pixels to avoid blurry anti-aliasing:

```
var x = d3.scaleBand()
                .domain(["A", "B", "C", "D"])
                .range([0, 720], .2);

x("B"); // 206, bar position
x.bandwidth(); // 137, bar width
```

Here the scale returns the position of the left-side of the bar, while the rangeBand method returns the bar width.

- Ordinal ranges are particularly useful for **bar charts.**

OUTLINE

- Recap
- Scales
- **Axes**
- SVG Shapes
- Example

AXES

- D3 provides convenient labeling for scales, in the form of **d3.axis** component.
- Create an axis for a given scale, and configure as desired:

```
var yAxis = d3.axisBottom(y);
```

```
d3.select(".axis")
.call(d3.axisBottom(x));
```

- Customize axis appearance via CSS:

```
.axis path, .axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
```

AXES: Ticks

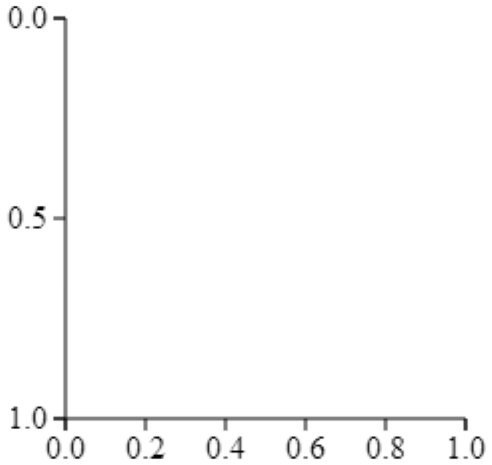
- There are a variety of other properties you might want to configure on an axis, such as the number of ticks or the tick format.

```
var x = d3.scaleLinear()
                    .domain([12, 24])
                    .range([0, 720]);

x.ticks(5); // [12, 14, 16, 18, 20, 22, 24]
```

- Keep in mind that the requested count is only a hint and not a strict directive (it depends on type and extension of domain)

- Particularly useful are the **d3.format** and **d3.time.format** for respectively general and time-focused tick formatting.

AXES: Example




```
var svg = d3.select("svg");

    cx = 40;
    y = 40;
    = 200;
    00;

                3.scale.linear()
                1])
                );

var yscal                linear()
    .domai
    .range([0

var xg = svg.append
    .attr("transform"                [cx,cy+ch] + ")");

var xAxis = d3.svg
    .scale(xscal
    .orient("b
    .ticks(

xAxis(x

va                nd("g")
                transform", "translate(" + [
                d3.svg.axis()
                (yscale)
                ient("left")
                tickValues([0, 0.5, 1])

Axis(yg);
```

OUTLINE

- Recap
- Scales
- Axes
- **SVG Shapes**
- Example

SVG SHAPES

- Absolute positioning; the origin ⟨0,0⟩ is the **top-left corner.**
  - Because the origin is in the top-left rather than bottom-left, you often see y-scales with an inverted range (e.g., [height, 0]).

- The "transform" attribute on <g> elements lets you define a new coordinate system. A translate lets you shift the origin based on margins; this approach is convenient because you isolate the margin definition to a single place in your code:


```
var svg = d3.select("body")
                    .append("svg")
                    .attr("width", outerWidth)
                    .attr("height", outerHeight);

var g = svg.append("g")
                     .attr("transform", "translate("
                   + marginLeft + ","
                   + marginTop + ")");
```

SVG SHAPES: basics

- Rect:

```
<rect
    x="0"
    y="0"
    width="0"
    height="0"
    rx="0"
    ry="0">
```

- Circle:

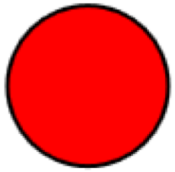
```
<circle
    cx="0"
    cy="0"
    r="0">
```

- Line:

```
<line
    x1="0"
    y1="0"
    x2="0"
    y2="0">
```

- Text:

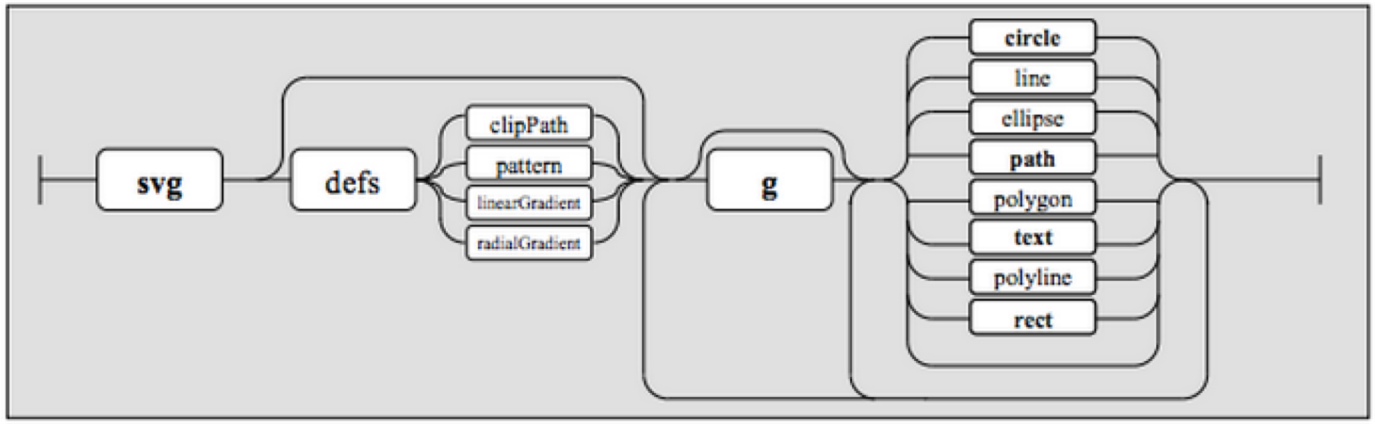
```
<text
    x="0"
    y="0"
    dx="0"
    dy="0"
    text-anchor="start">
```



- Note that position and size must be defined as attributes in SVG, rather than styles. Colors and opacity should be specified as CSS, however.

SVG SHAPES: basics

SVG



Positioning :

		svg	g	rect	text	line	path	polygon	polyline	circle	ellipse	style	attr
transform	"translate(x,y)"		●	●	●	●	●	●	●	●	●		●
x,y	number			●	●								●
x1,x2,y1,y2	number					●							●
d	(special)						●						●
points	"x,y x,y x,y"							●	●				●
cx,cy	number									●	●		●

Sizing:

		svg	g	rect	text	line	path	polygon	polyline	circle	ellipse	style	attr
transform	"scale(k)"		●	●	●	●	●	●	●	●	●		●
width, height	number	●		●									●
r	number									●			●
rx,ry	number										●		●

- Recap of the principal functions applicable to the different types of SVG shapes

SVG SHAPES: Paths

- As SVG specification, simply a sequence of directive+ control points:

```
<path d="M152.64962091501462,320.5600780855698L133.
88913955606318,325.4363177123538L134.9689095444304
6,330.3791763492199 6L131.19348249532786,331.1583936
14812L98.56681109628815,335.53933807857004L91.14450
799488135,333.79662025279L72.1880101321918,333.7473
3970068166L69.51723455785742,332.8569681440152L62.3
7313911354066,333.2100666843387L62.248334309137434,
```

- Paths require another mini-language; example: M152,300 means «move» to point with coordinates (152,300)

- It's occasionally useful to generate path strings by hand for simple and static shapes, but in general this can be a cumbersome process.

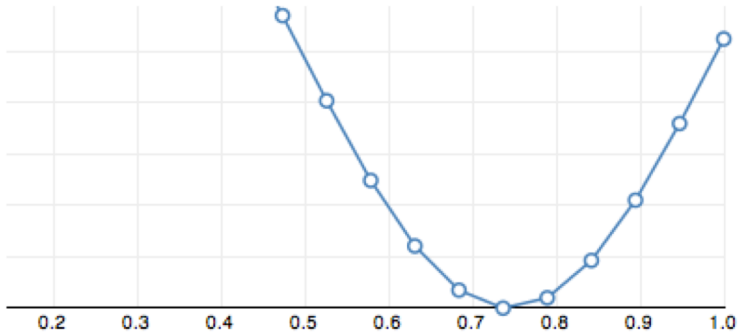
- D3 provides ways to construct common path types from data, in the form of **path generators**.

SVG SHAPES: Path Generators

- Configurable functions for generating paths from data.

d3.line:

- Define a path in terms of *x* and *y*.




```
var xcord = d3.scaleLinear(),
      ycord = d3.scaleLinear();

var line = d3.line()
     .x(function(d) { return xcord(d.x); })
     .y(function(d) { return ycord(d.y); });
```

- Compose scales with data accessors to define position (you're always working with data, NOT position)

SVG SHAPES: Path Generators

- Pass data to the line generator directly:

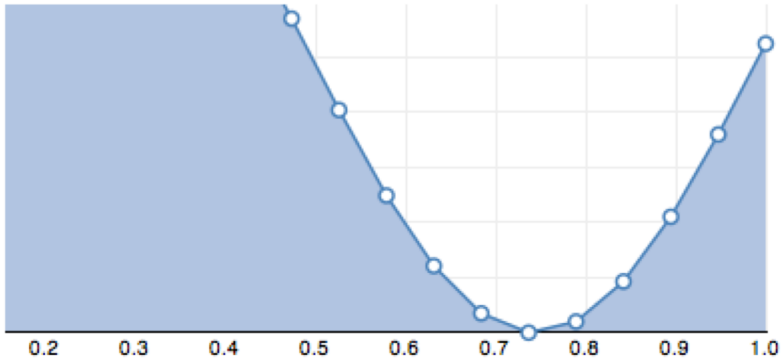
```
svg.append("path")
      .datum(objects)
      .attr("class", "line")
      .attr("d", line);
```

- The **datum** operator lets you bind data to elements without computing a data-join
- In this case, we're assigning a single data point to a single element.
- Nonetheless, this single data point is actually an array of objects.

SVG SHAPES: Path Generators

d3.area:

- Define a path in terms of x, y_0 and y_1.



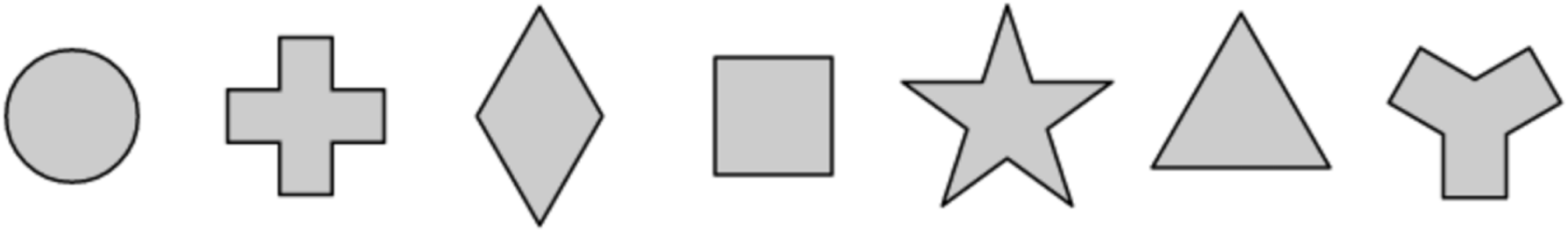

```
var x = d3.scaleLinear(),
      y = d3.scaleLinear();

var area = d3.area()
        .x(function(d) { return x(d.x); })
        .y0(height)
        .y1(function(d) { return y(d.y); });
```

- the top line is formed using the ***x***- and ***y1***-accessor functions, and proceeds from **left-to-right**

- the bottom line is added to this line, using the ***x***- and ***y0***-accessor functions, and proceeds from **right-to-left**

Symbols

- circle ↦ d3.symbolCircle
- cross ↦ d3.symbolCross
- diamond ↦ d3.symbolDiamond
- square ↦ d3.symbolSquare
- triangle-up ↦ d3.symbolTriangle
- star ↦ d3.symbolStar
- wye↦ d3.symbolWye



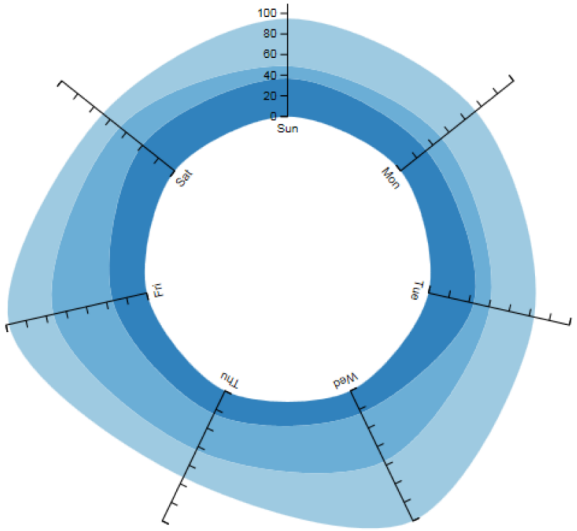
- Symbol types are passed to *symbol*.type in place of strings

SVG SHAPES: Path Generators

- **Radial variants**: similar to the linear ones, but expressed in polar coordinates:

- **d3.radialLine**

```
var line = d3.radialLine()
        .interpolate("cardinal-closed")
        .angle(function(d) { return angle(d.time); })
        .radius(function(d) { return radius(d.y0 + d.y); });
```



- **d3.radialArea**

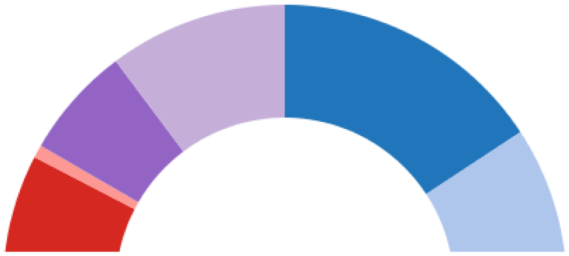
```
var area = d3.radialArea()
        .interpolate("cardinal-closed")
        .angle(function(d) { return angle(d.time); })
        .innerRadius(function(d) { return radius(d.y0); })
        .outerRadius(function(d) { return radius(d.y0 + d.y); });
```

SVG SHAPES: Path Generators

- **d3.arc:**

  A path generator for pie and donut charts, among other uses.

- SVG definition of arc:



```
var myArc = {
    "innerRadius": 0,
    "outerRadius": 360,
    "startAngle": 0, // 12 o'clock
    "endAngle": 1.2 // radians
};
```

- If you define the inner and outer radius as constants, just the remaining attributes need to be specified ({startAngle: 0, endAngle: 1.2})

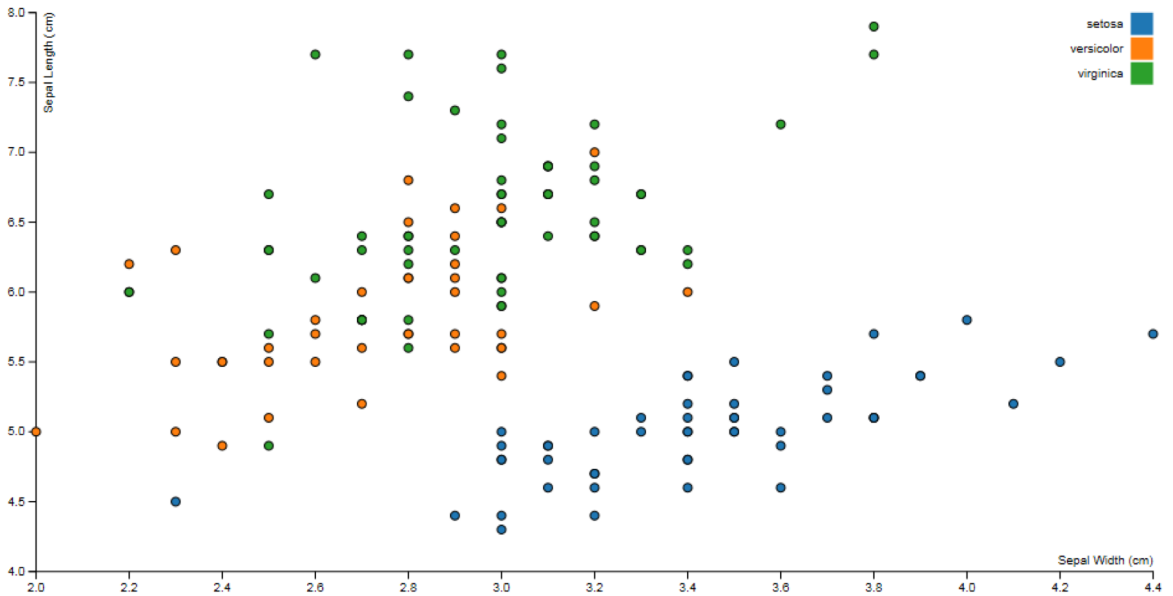
```
var arc = d3.arc()
        .innerRadius(0)
        .outerRadius(360);
```

OUTLINE

- Recap
- Scales
- Axes
- SVG Shapes
- **Example**

RECAP EXAMPLE: Scatterplot

- Now, put all the things seen in this lesson to create a real first visualization:



(click on the figure for code; open with Firefox)

NEXT LESSON

- **Working with data**
- **Layouts**
- **Examples….**