

IN4MATX 133: User Interface Software

Lecture 7:
TypeScript &
Data Visualization Tools

Professor Daniel A. Epstein
TA Lucas de Melo Silva
TA Jong Ho Lee

Today's goals

By the end of today, you should be able to...

- Write code which follows object-oriented principles in TypeScript
- Explain the advantages and disadvantages of using TypeScript
- Describe the concepts of threshold and ceiling in software tools and what tool designers should be striving to create
- Explain the relative threshold and ceilings of visualization tools like Protopis, D3, and Vega-Lite
- Describe common visualization primitives like marks, axes, and scales
- Implement simple visualizations with Vega-Lite

Using npm to install TypeScript

- `npm install -g typescript`
- Installs `tsc`, the TypeScript transpiler
- Could also install via HomeBrew (mac) or Chocolatey (pc)

<https://www.typescriptlang.org/#download-links>

So what is TypeScript?

About TypeScript

- Released by Microsoft in 2012
- Originally only supported in Visual Studio
 - Now in Eclipse, Sublime, Webstorm, Atom, etc.
- Latest version is TypeScript 3.6, released in April 2019



About TypeScript

- Introduces static types, type checking, and objects
 - These are all optional!
- A strict superset of JavaScript
 - Syntactically-correct JavaScript will also run in TypeScript
 - Means TypeScript can use popular JavaScript libraries
- “Transcompiles” or “Transpiles” to JavaScript
 - Takes TypeScript code and converts it to equivalent JavaScript code



Typing

- hello.ts

```
var courseNumber:number = 133;
```

```
console.log('Hello, IN4MATX ' + courseNumber + '!');
```

- Transpiling: tsc hello.ts

- Generates hello.js

```
var courseNumber = 133;
```

```
console.log('Hello, IN4MATX ' + courseNumber + '!');
```

Typing

- Pre-defined types
 - Number
 - Boolean
 - String
 - Void (generally a function return type)
 - Null
 - Undefined
 - Any

Typing

- Typing is optional

```
var courseNumber: any = 133;
```

```
var courseNumber = 133;
```

```
console.log('Hello, IN4MATX ' + courseNumber + '!');
```

Typing

- Functions can specify argument types and return types

```
function area(shape: string, width: number, height: number)
{
    var area = width * height;
    return "I'm a " + shape + " of area " + area + " cm^2.";
}

// "better" function
function area(shape: string, width: number,
height: number):string {
    var area:number = width * height;
    return "I'm a " + shape + " of area " + area + " cm^2.";
}
```

Typing

- Types enable error checking

```
// "better" function  
function area(shape: string, width: number, height:  
number):string {  
    var area:number = width * height;  
    return "I'm a " + shape + " of area " + area + "  
cm2.";  
}
```

```
document.body.innerHTML = area(15, 15, 'square');
```

```
error: Argument of type '15' is not assignable to parameter of type 'string'
```

TypeScript demo



```
Elements    Console    Sources
top    Filter

hello my name is theresa
hello my name is theresa
hello my name is theresa
in4matx-133
in4matx-133
two2too2to
two2too2to
two2too2to
two2too2to
> |
```

Question



Which TypeScript files would transpile to

```
function pythagorean(a, b) {  
    var cSq = a * a + b * b;  
    return Math.sqrt(cSq);  
}
```

- A 1
- B 2
- C 3
- D 1 and 2
- E 1, 2, and 3

```
1 function pythagorean(a:number, b:number):number {  
    var cSq = a*a + b*b;  
    return Math.sqrt(cSq);  
}  
  
2 function pythagorean(a:number, b:number) {  
    var cSq = a*a + b*b;  
    return Math.sqrt(cSq);  
}  
  
3 function pythagorean(a, b) {  
    var cSq = a * a + b * b;  
    return Math.sqrt(cSq);  
}
```

Question



Which TypeScript files would transpile to

```
function pythagorean(a, b) {  
    var cSq = a * a + b * b;  
    return Math.sqrt(cSq);  
}
```

- A 1
- B 2
- C 3
- D 1 and 2
- E 1, 2, and 3

```
1 function pythagorean(a:number, b:number):number {  
    var cSq = a*a + b*b;  
    return Math.sqrt(cSq);  
}  
  
2 function pythagorean(a:number, b:number) {  
    var cSq = a*a + b*b;  
    return Math.sqrt(cSq);  
}  
  
3 function pythagorean(a, b) {  
    var cSq = a * a + b * b;  
    return Math.sqrt(cSq);  
}
```

Classes

- Also just like in Java, with a constructor and methods

```
class Shape {  
    area: number;  
    color: string;  
    name: string;  
  
    constructor (name: string, width: number, height: number) {  
        this.name = name  
        this.area = width * height;  
        this.color = "pink";  
    };  
    shoutout() {  
        return "I'm " + this.color + " " + this.name + " with an area of " +  
this.area + " cm squared."  
    }  
}  
  
var square = new Shape("square", 30, 30);
```

Classes

- Will make a function() with prototype methods when transpiled

```
//shape.js
var Shape = /** @class */ (function () {
    function Shape(name, width, height) {
        this.name = name;
        this.area = width * height;
        this.color = "pink";
    }
    ;
    Shape.prototype.shoutout = function () {
        return "I'm " + this.color + " " + this.name + " with an area of
" + this.area + " cm squared.";
    };
    return Shape;
}());
var square = new Shape("square", 30, 30);
```

tsconfig.json

- Indicates a TypeScript project
 - Indicates what files or folders to compile
 - Pick compiler options, such as whether to remove comments
 - Specify what version of ECMAScript to compile to
 - `tsc --project tsconfig.json`

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

Benefits of TypeScript

- Type checking
 - Transpiler can show warnings or errors *before* code is executed
 - Because your editor knows the types, it can autocomplete methods and API features
 - Easier to refactor code
- Object-oriented
 - Easier to manage/maintain large code bases
 - Simple enough to use; same principles as Java and other OO languages

Drawbacks of TypeScript

- Compiling is occasionally a pain
 - Can be slow for large projects
- Might not work with new JavaScript libraries out of the gate
 - It took a little while for TypeScript to interface nicely with Angular and React, for example
 - Now it's the default for Angular

Other noteworthy JavaScript transpilers

- Dart
 - Developed by Google
 - Introduces typing and similar object-oriented practices
 - Transpiles to JavaScript with dart2js
- CoffeeScript
 - Open-source development
 - “Python-like” variable assignment, loops, and conditionals
 - Mostly meant to make JavaScript syntax prettier/cleaner



Software Tools and Visualization

**Today is a *very* narrow slice
of visualization**

If you want more, take IN4MATX 143

Sequential programs (command line)

- Program takes control, prompts for input
- Person waits on the program
- Program says when it is ready for more input, which the person then provides



Sequential programs (command line)

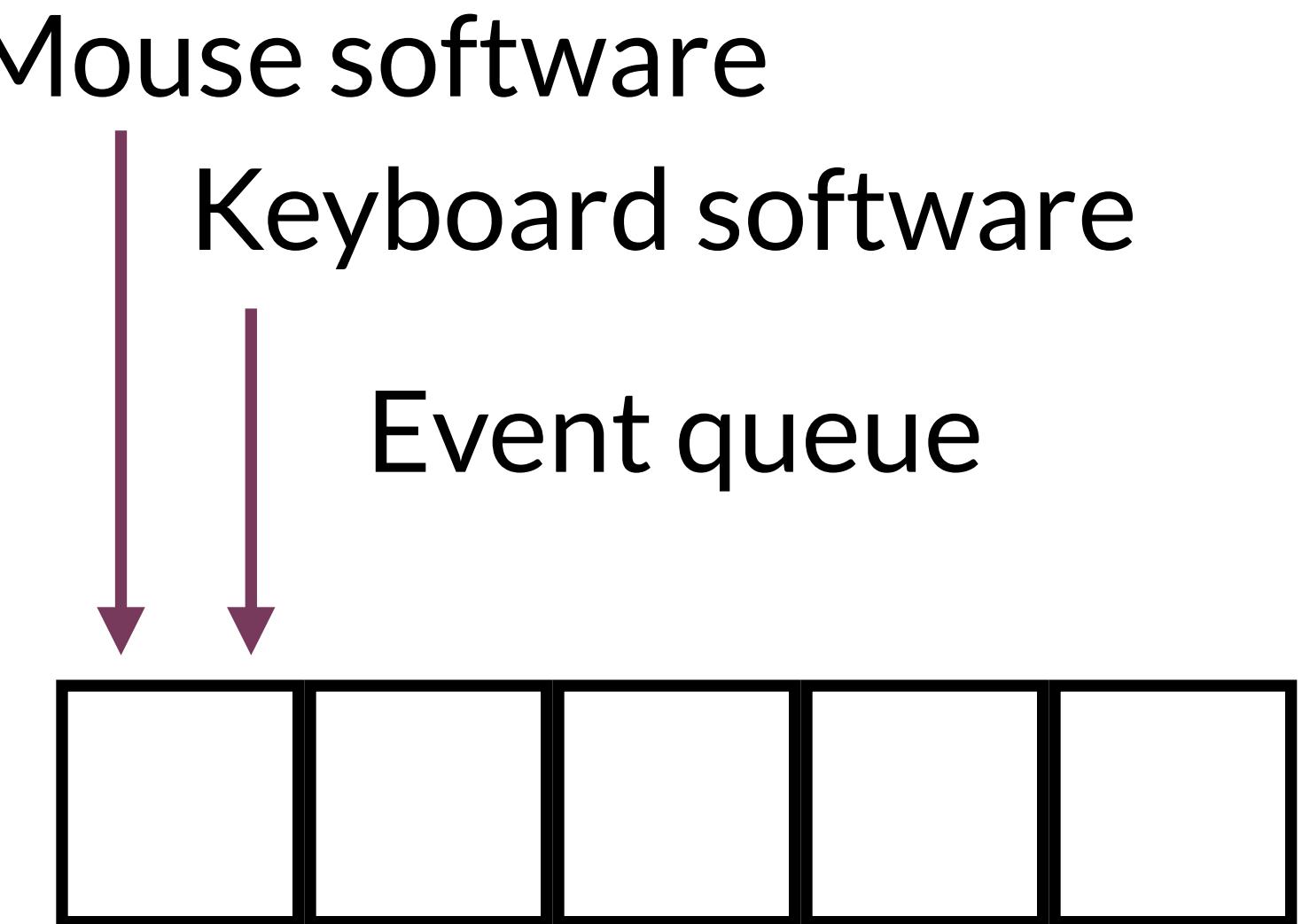
```
while true {  
    print "Prompt for Input"  
    input = readline_of_text()  
    output = do_work()  
    print output  
}
```

- Person is literally modeled as a file



Event-driven programming

- A program waits for a person to provide input
- All communication is done via events
 - Mouse down, item drag, key up
- All events go in a queue
 - Ensures events are handled in order
 - Hides specifics from applications



Basic interactive software loop

- All interactive software has this loop somewhere

```
do {  
    e = read_event();      ← Input  
    dispatch_event(e);   ← Processing  
    if (damage_exists())  
        update_display(); ← Output  
} while (e.type != WM_QUIT);
```

Basic interactive software loop

- Maybe if you've made a game, you've built this loop
- But imagine you had to write this loop every time you wanted to write a webpage, desktop app, or mobile app
- Instead, we rely on tools to handle common operations

```
do {  
    e = read_event();  
    dispatch_event(e);  
    if (damage_exists())  
        update_display();  
} while (e.type != WM_QUIT);
```

Example: a button

- What's behind a button?
 - Set X and Y boundaries
 - Check if mouse down is within those boundaries
 - Check if mouse up is *also* within those boundaries
 - If so, then fire an event
- What if you had to program this sequence every time you wanted to add a button to your website?



Understanding tools

What is a user interface tool?

- Software or libraries which help you build a user interface
 - Bootstrap is a user interface tool, designed to help make interfaces responsive
 - Angular, React, etc. are all user interface tools

Understanding tools

We use tools because they...

- Identify common or important practices
- Package those practices in a framework
- Make it easy to follow those practices
- Make it easier to focus on the application we're building

Understanding tools

Tools enable...

- Faster and more iterative design
- Better implementation than without the tool
- Consistency across applications using the same tool

Understanding tools

Why is designing tools difficult?

- Need to understand the core practices and problems
- Those are often evolving with technology and design
- The tasks people are trying to solve change quickly, so tools struggle to keep up

Understanding tools

Key terms

- Threshold: How hard to get started
- Ceiling: How much can be achieved
- Path of least resistance: Tools influence what interfaces are created
- Moving targets: Changing needs make tools obsolete

Threshold

How hard to get started

- Some tools are harder to pick up
- Depends on what a person knows already
 - A new programming language adds to the threshold
 - If a tool borrows concepts from another popular tool, it will be easier for many people to pick up

Ceiling

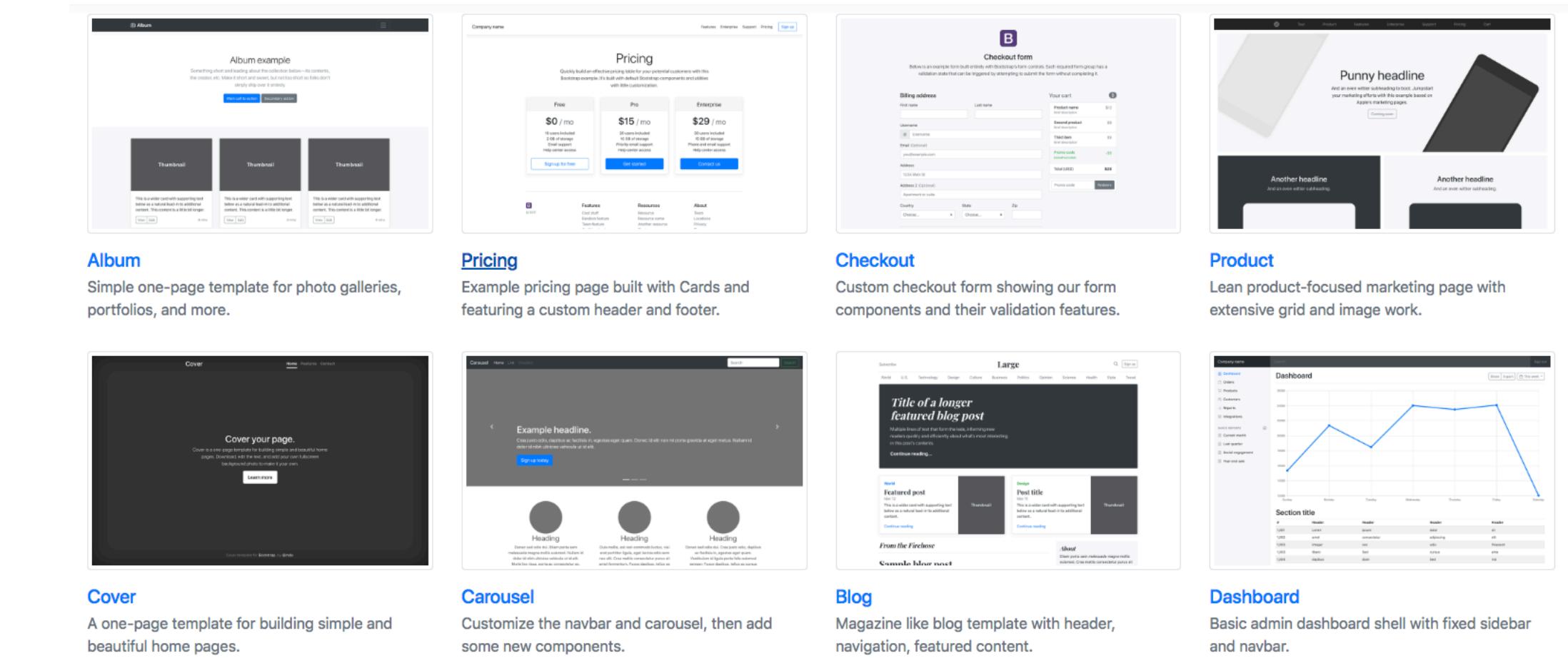
How much can be achieved

- Tools restrict what's possible
 - Your program could do much more if it had direct access to the bits on your computing device

Path of least resistance

Tools influence what interfaces are created

- Remember the concern that all Bootstrap pages look similar?
- Sapir-Whorf Hypothesis
 - Roughly, some thoughts in one language cannot be expressed or understood in another language
- Our tools frame how we think about interaction and design

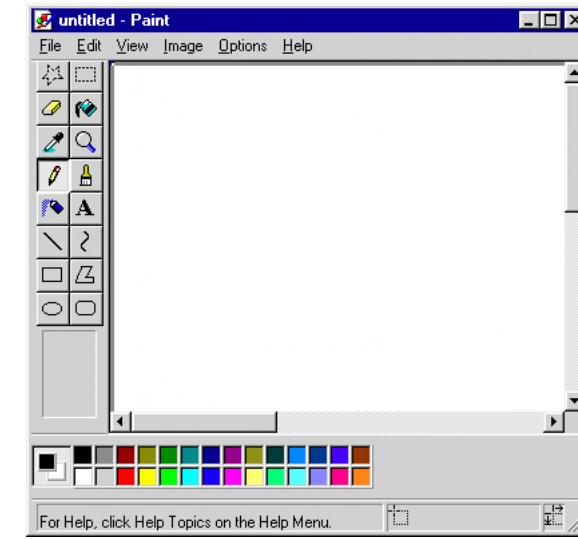


Moving targets

Changing needs make tools obsolete

- Codification eventually constrains design
 - Our understanding of how people interact with technology improves
 - New technology comes along to change the needs of tools
 - Example: Virtual reality has wildly different interactions and tool needs

Question

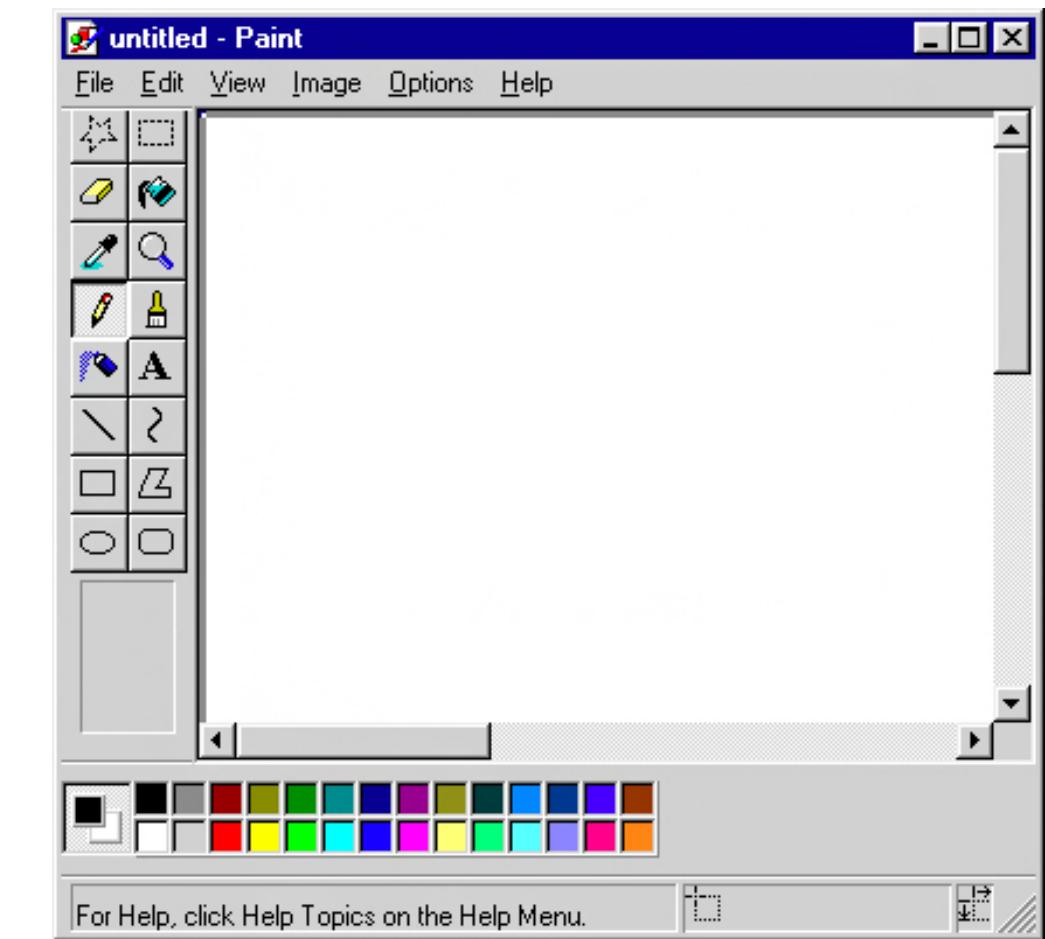


Relative to each other, what are the *threshold* and *ceiling* of Adobe Photoshop and Microsoft Paint?
(They're tools for making images, but the principles apply)

- A Photoshop: high ceiling, high threshold; Paint: low ceiling, low threshold
- B Photoshop: high ceiling, low threshold; Paint: low ceiling, high threshold
- C Photoshop: low ceiling, high threshold; Paint: high ceiling, low threshold
- D Photoshop: low ceiling, low threshold; Paint: high ceiling, high threshold
- E I'm confused, just clicking in to get participation credit

Threshold and ceiling

- It's all relative; no absolute measure
- Tools should be *low threshold*
 - Easy to pick up
- But tools should also be *high ceiling*
 - Can do a lot
- The best tools are both
 - Photoshop introduces tutorials, etc. to lower the threshold



**Ok, so what does any of this
have to do with visualization?**

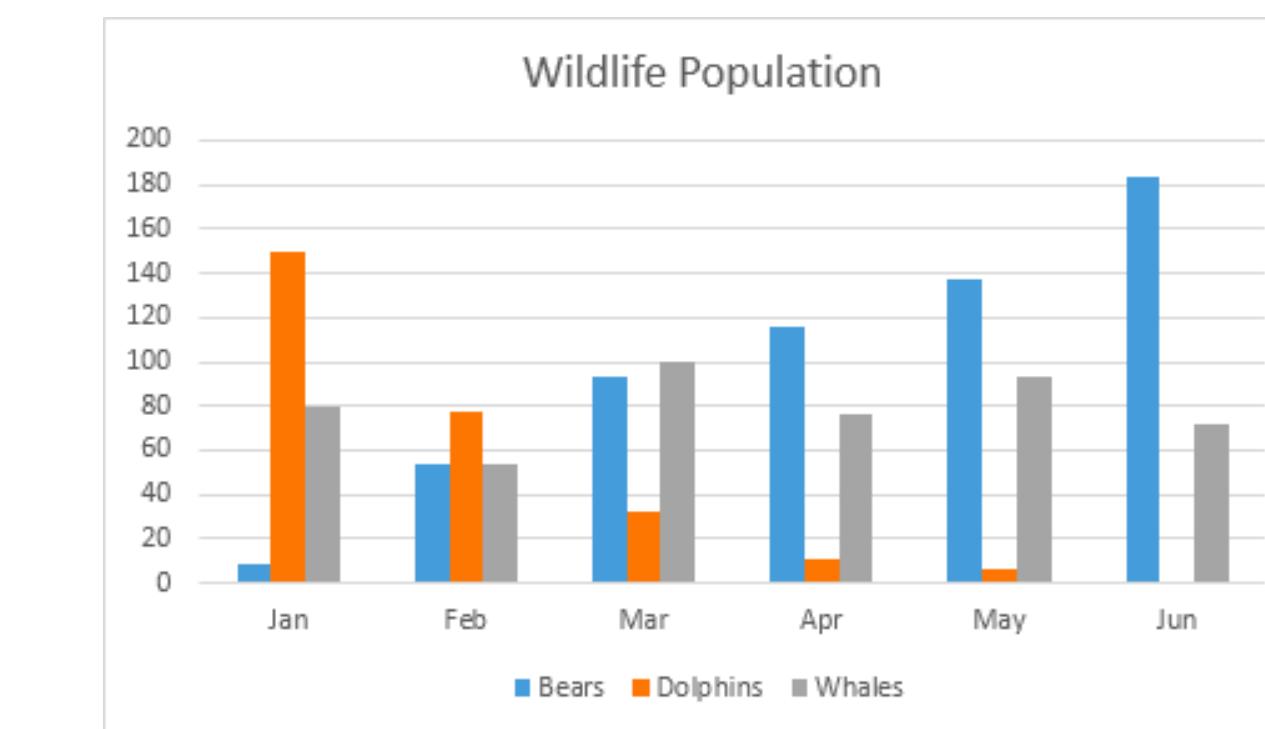
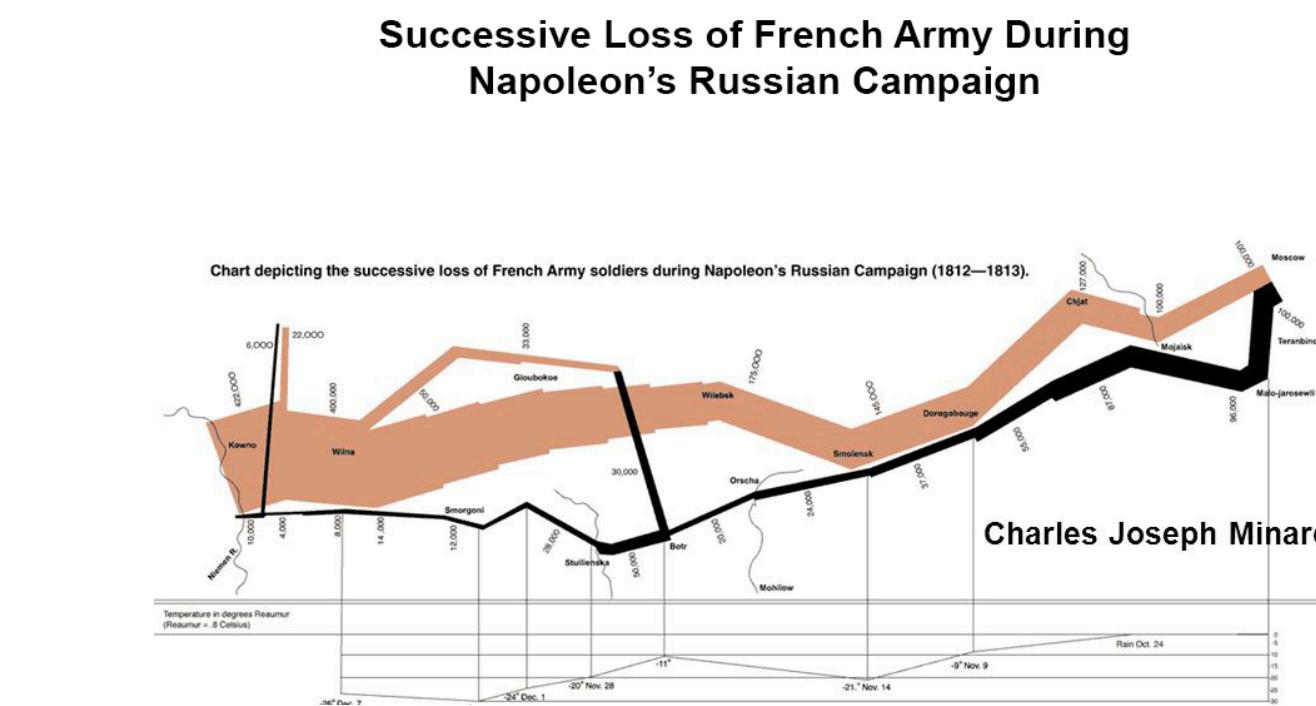
Scaleable Vector Graphics (SVG)

- XML format for specifying graphics
 - Looks somewhat like HTML
 - Most browsers can render them
 - Composed of lines, circles, rectangles, etc.

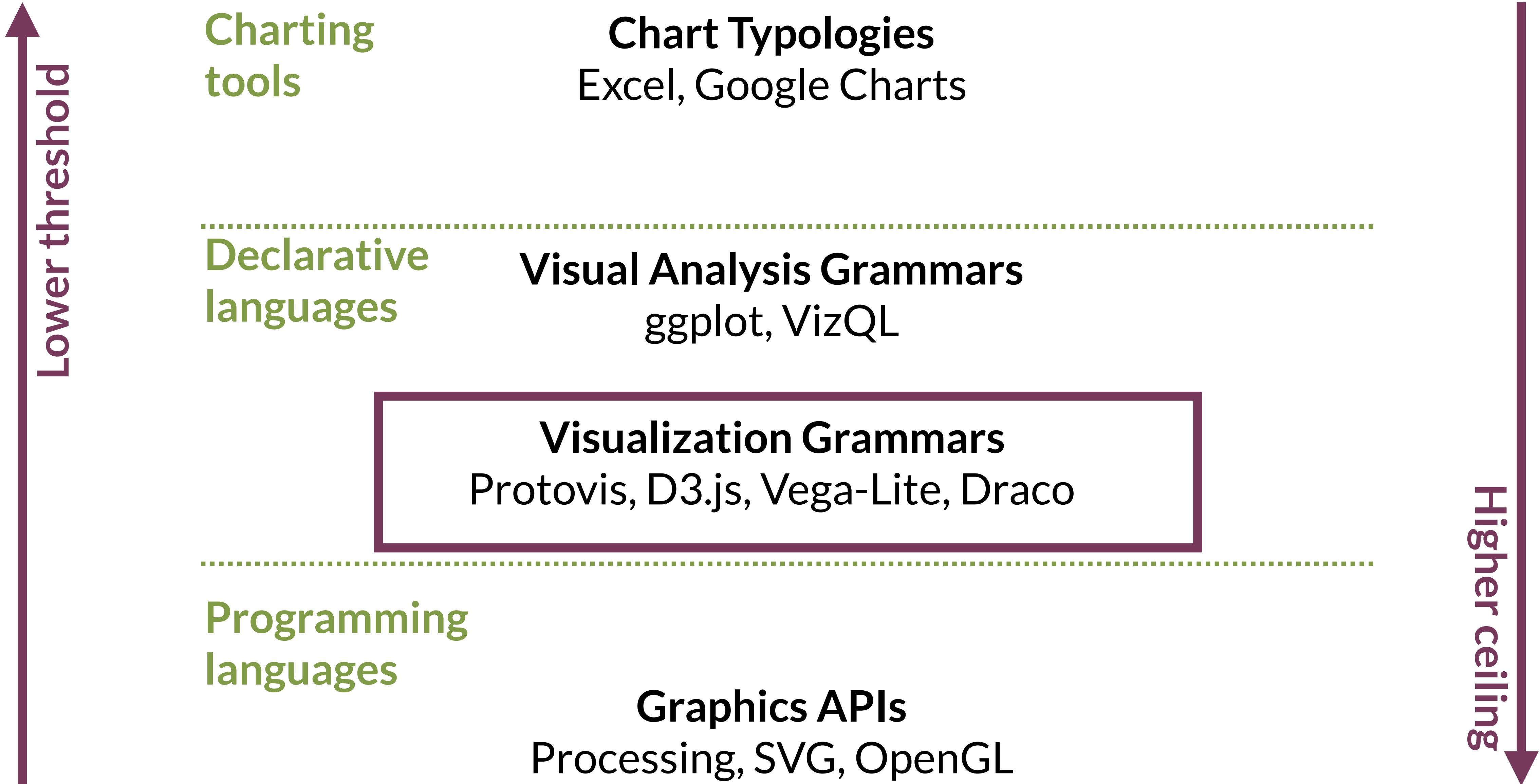
```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="green" stroke-
width="4" fill="yellow" />
</svg>
```

Visualization tools

- Are governed by the same principles
- Scalable vector graphics (svg)
 - High ceiling, but high threshold
- Microsoft excel
 - Low threshold, but low ceiling



<https://www.edwardtufte.com/tufte/posters>



Declarative languages

- Programming by describing *what*, not *how*
- Separate specification (*what you want*) from execution (*how it should be computed*)
- Contrasts to **imperative** programming, where you must give explicit steps

Declarative languages



Markup
language



Styling
language



Programming
language

Declarative languages

HTML



Declarative
language

CSS



Declarative
language

JS



Imperative
language

Declarative languages

HTML



```
<main class="container">
  <div class="row">
    <div class="col-3">A</div>
    <div class="col-6">B</div>
    <div class="col-4">C</div>
    <div class="col">D</div>
    <div class="col">E</div>
  </div>
</main>
```

What should be rendered, but not how

JS



```
var array = [ '1', 'fish', 2, 'blue' ];
array[5] = 'dog';
array.push('2');
array[2] = array[array.length - 1] - 4;
array[0] = typeof array[2];
array[4] = array.indexOf('blue');

console.log(array.join('*'));
```

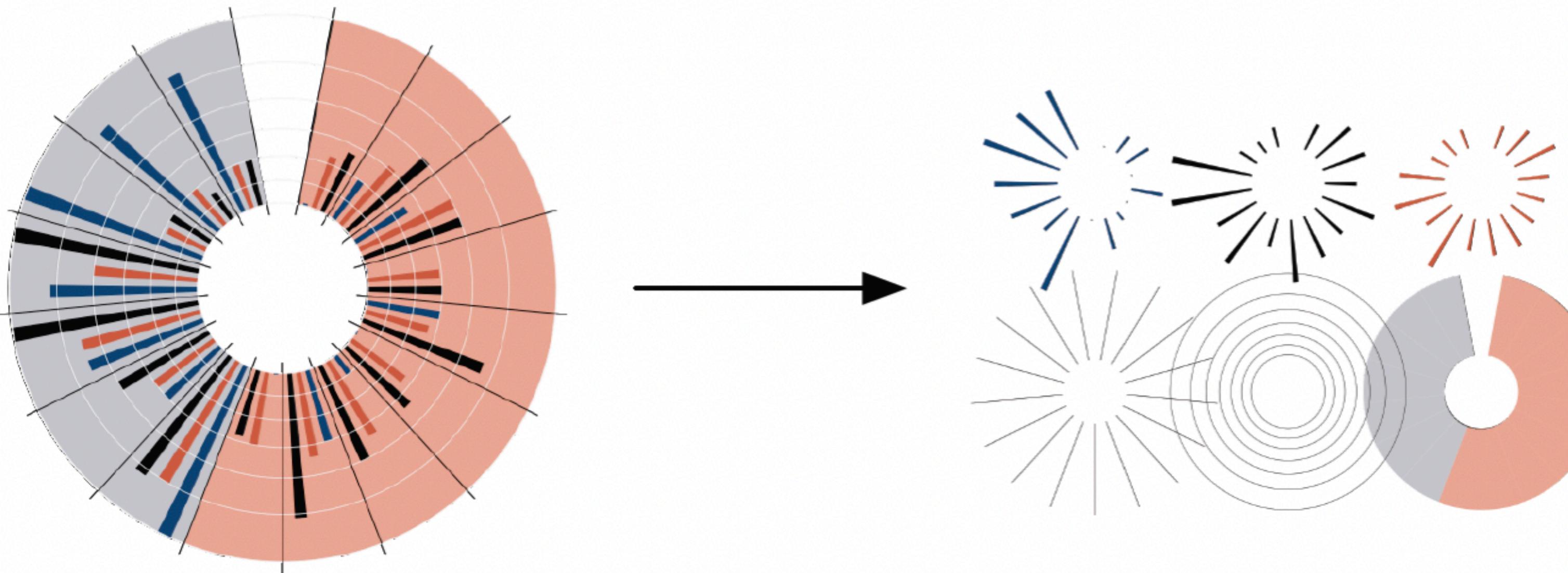
Step-by-step

Why declarative languages?

- Faster iteration, less code
 - Easier to pick up; lower threshold
 - Can be generated programmatically

Protopis

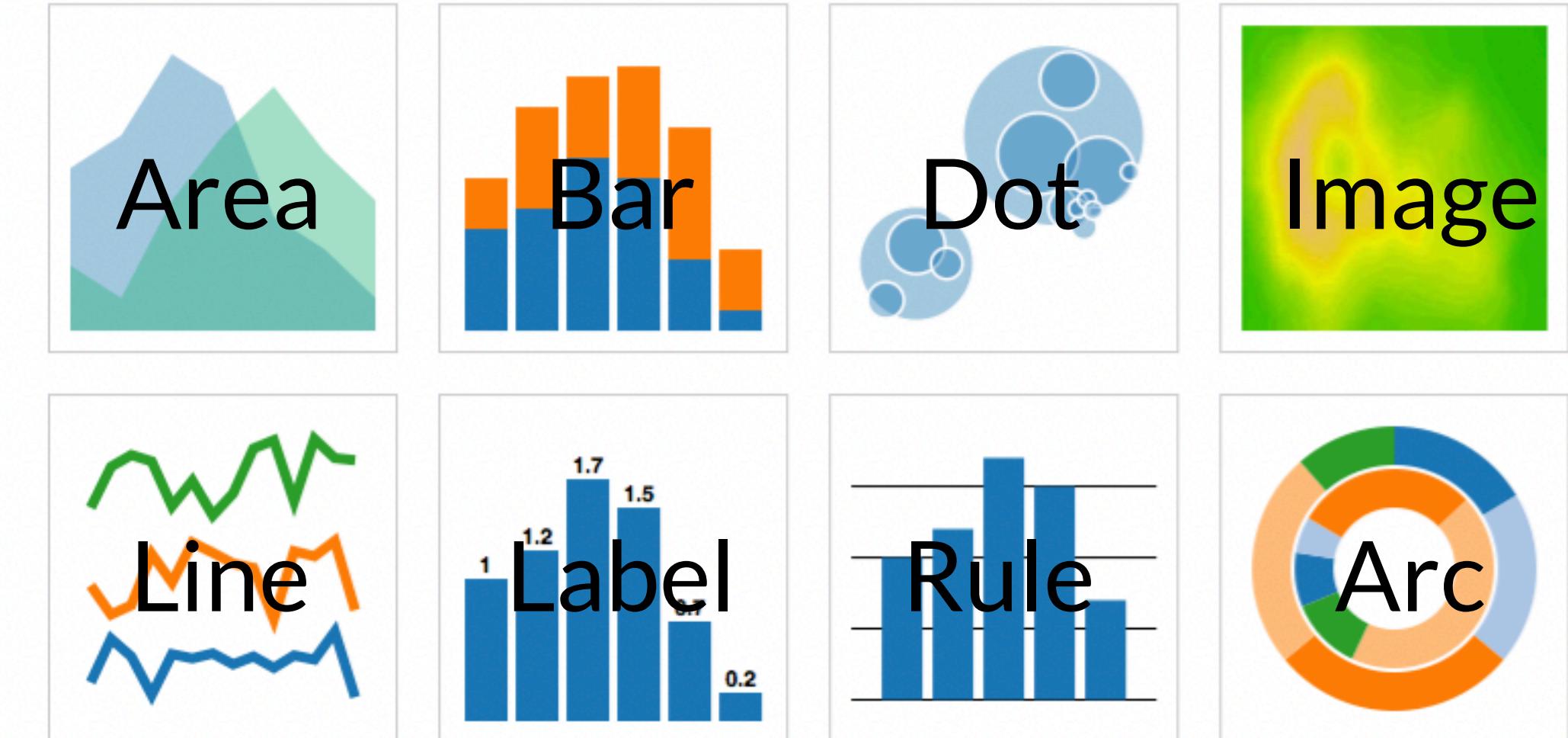
- Initial grammar for visualization
- A composition of data-representative marks
 - Self-contained JavaScript model (doesn't export to SVG or anything else)



Michael Bostock, Jeffrey Heer. IEEE Vis, 2009. Protopis: A Graphical Toolkit for Visualization.
<https://doi.org/10.1109/TVCG.2009.174>

Protopis

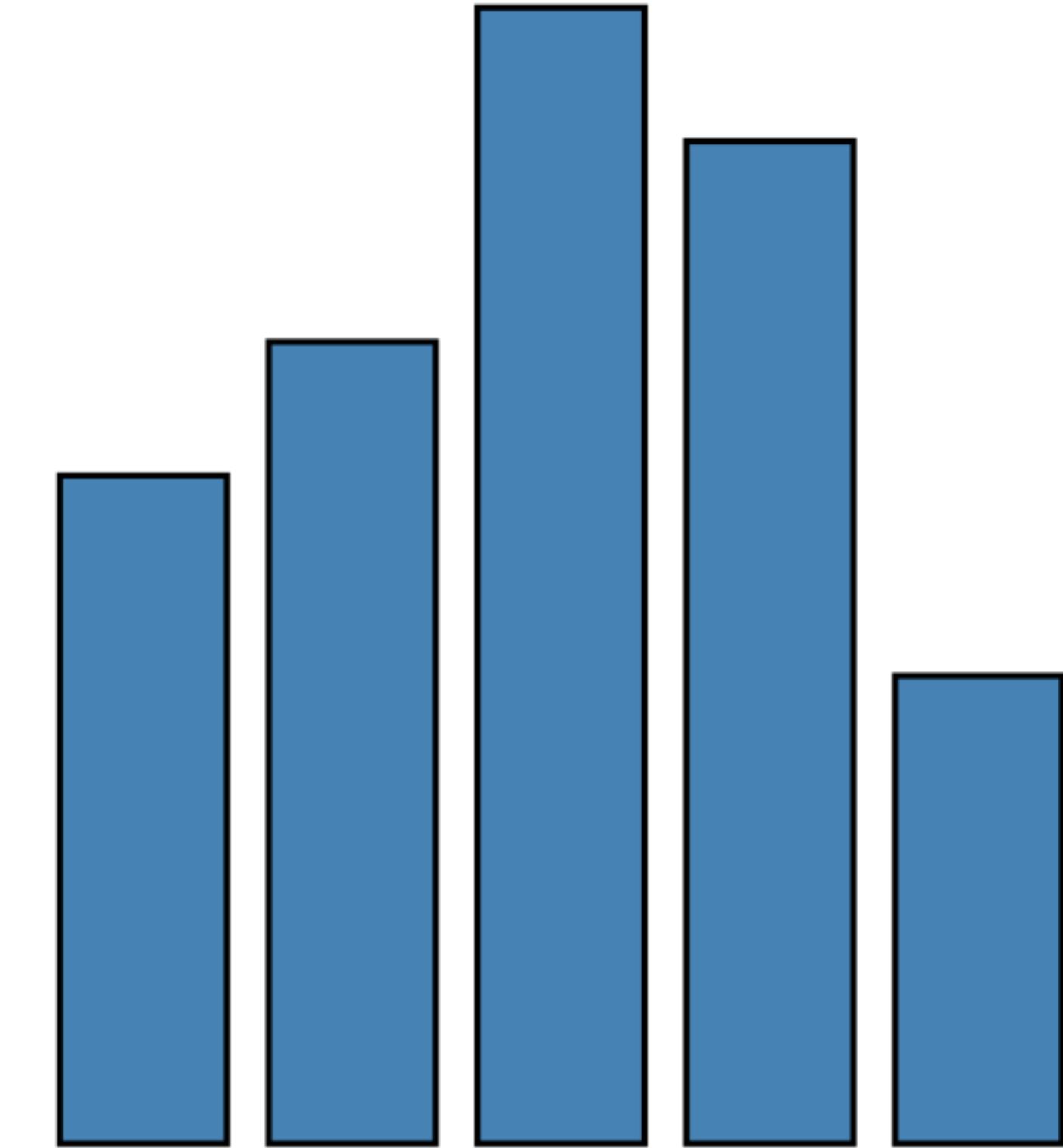
- Marks: graphical primitives
 - Marks specify how content should be rendered



Protopis

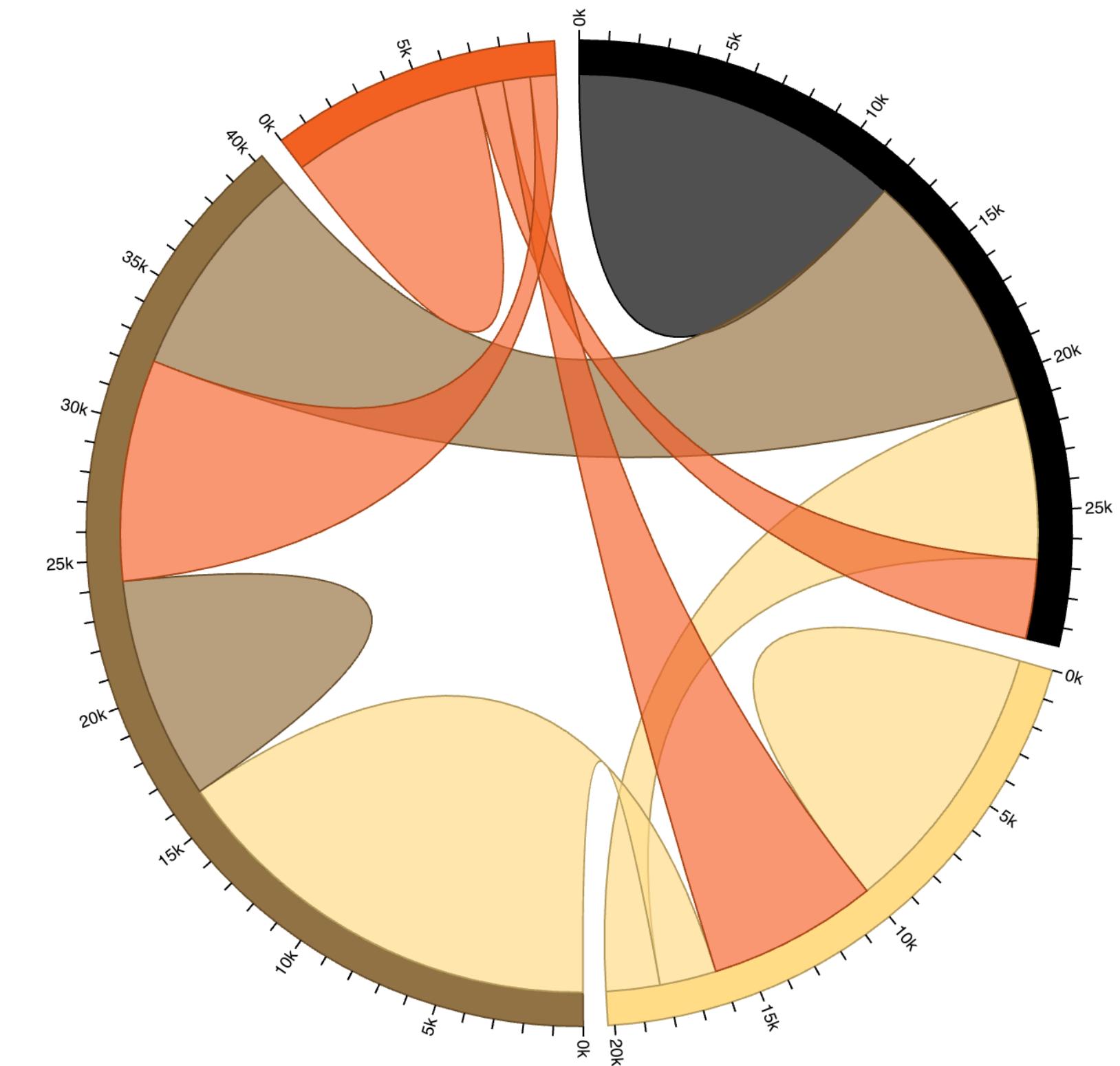
```
var vis = new pv.Panel();
vis.add(pv.Bar)←Mark
  .data([1, 1.2, 1.7, 1.5, 0.7])
  .visible(true)
  .left((d) => this.index * 25)
  .bottom(0)
  .width(20)
  .height((d) => d * 80) ←
  .fillStyle("blue")
  .strokeStyle("black")
  .lineWidth(1.5);
vis.render();
```

Literally specifies
which pixel each bar
should start at and
how many pixels tall
it should be



D3

- Binds data directly to a web page's DOM by editing a SVG
 - More expressive! Can make anything an SVG can make
 - Enables interactivity, can access mouse & keyboard events through the same tools as a browser
 - Much more complex...



D3

```
var svg = d3.select(DOM.svg(width,  
height));
```

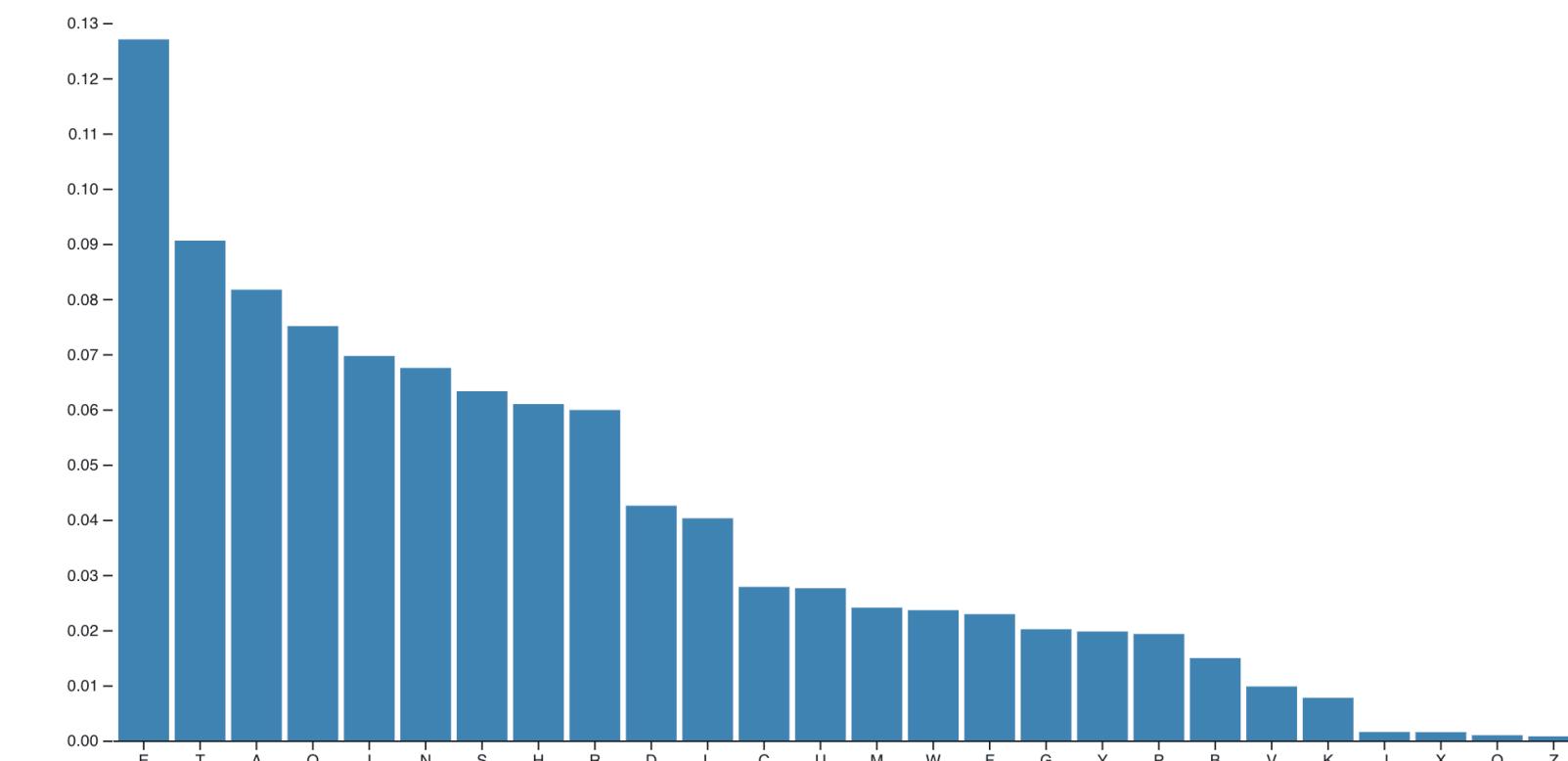


Find SVG in the DOM

```
svg.append("g")  
  .attr("fill", "steelblue")  
  .selectAll("rect").data(data).enter()  
  .append("rect") ← No more mention of marks!  
    .attr("x", d => x(d.name))  
    .attr("y", d => y(d.value))  
    .attr("height", d => y(0) -  
y(d.value))  
    .attr("width", x.bandwidth());
```

```
svg.append("g")  
  .call(xAxis);
```

```
svg.append("g")  
  .call(yAxis);
```



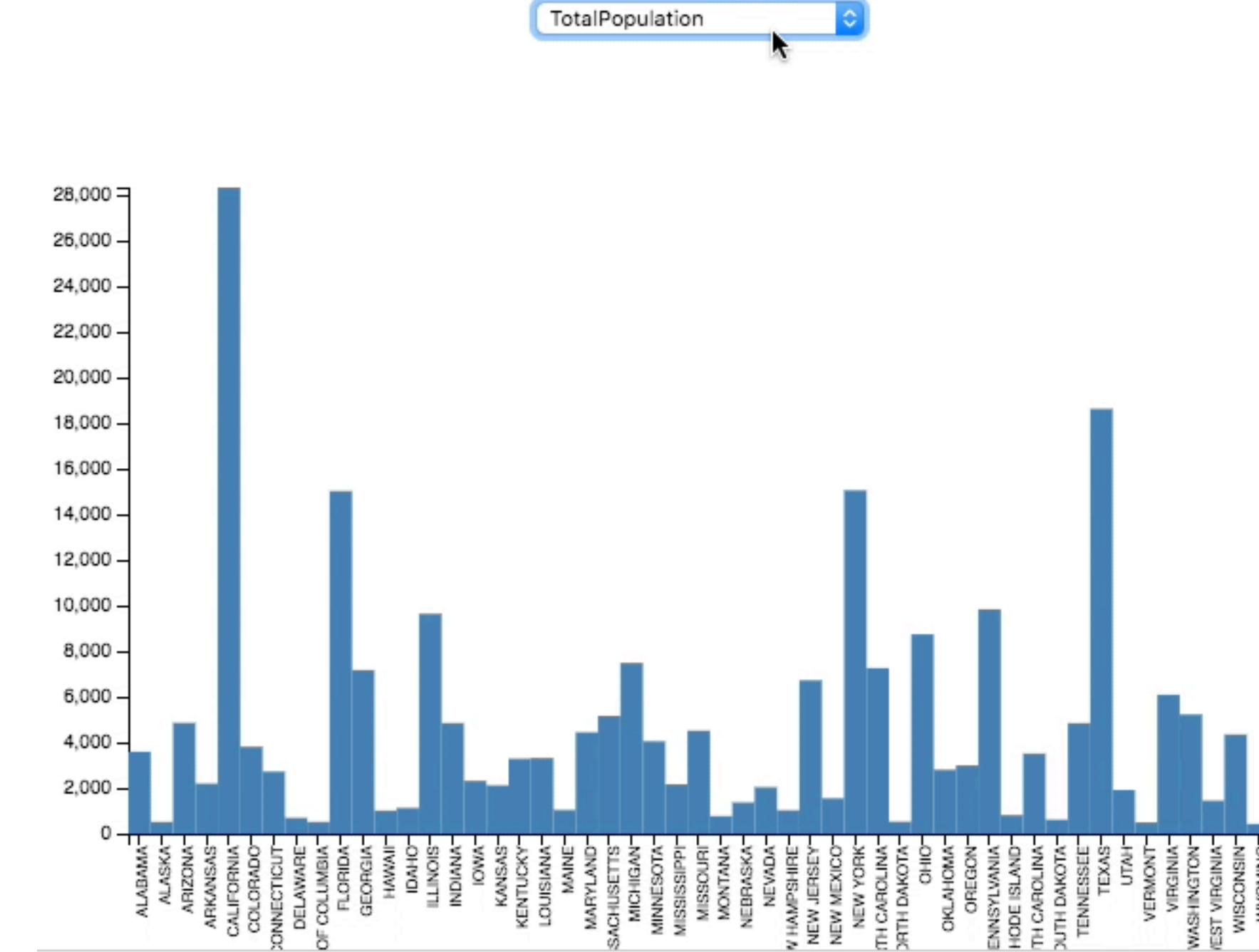
D3

~118 lines of code,
plus data in a separate file

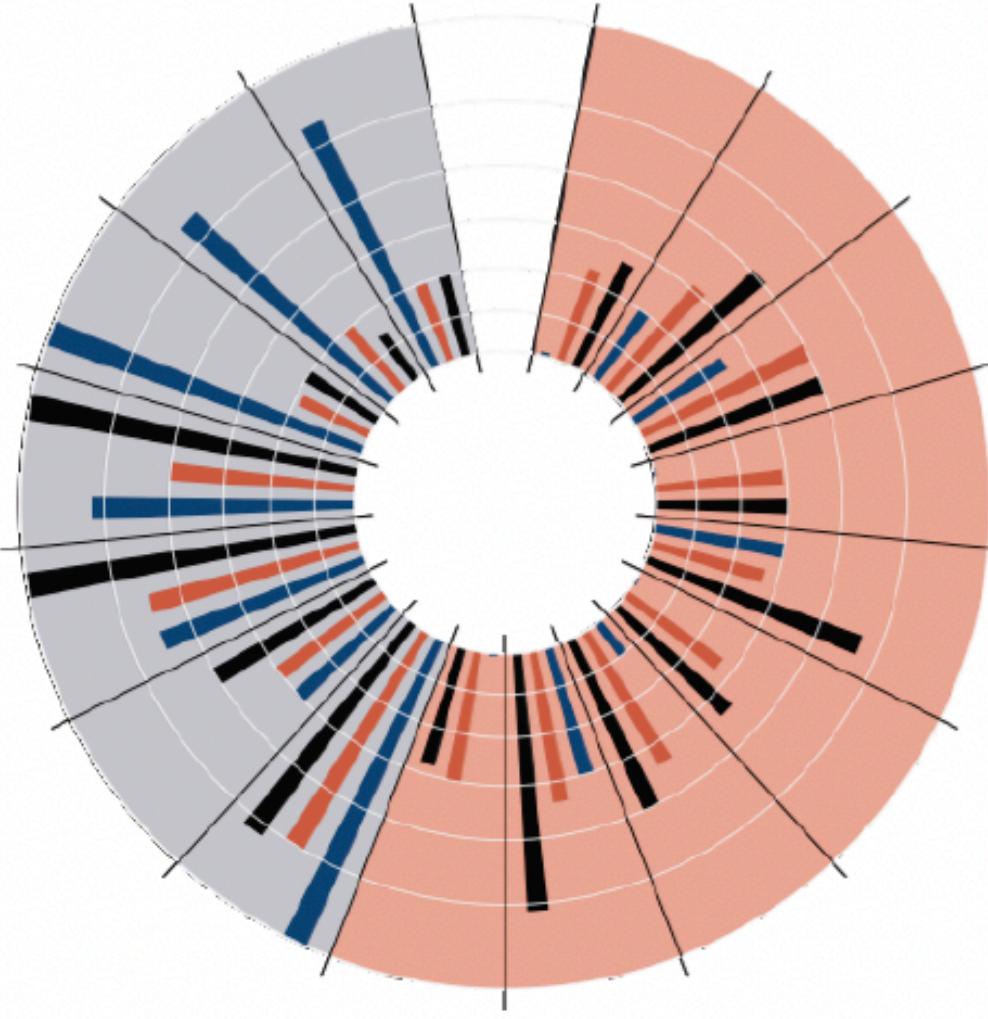
```

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
d3.tsv("VotingInformation.tsv", function(error, data){
  // filter year
  var data = data.filter(function(d){return d.Year == '2012'});
  // filter out column values
  var elements = Object.keys(data[0])
    .filter(function(d){return !(d == "Year") & (d != "State")});
  var selection = elements[0];
  var y = d3.scale.linear()
    .domain([0, d3.max(data, function(d){return d[selection]}))
    .range([height, 0]);
  var x = d3.scale.ordinal()
    .domain(data.map(function(d){ return d.State; }));
    .rangeBands([0, width]);
  var xAxis = d3.svg.axis()
    .scale(x)
    .orient('bottom');
  var yAxis = d3.svg.axis()
    .scale(y)
    .orient('left');
  svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")");
  .call(xAxis);
  .selectAll("text")
    .style("font-size", "8px")
    .style("text-anchor", "end")
    .attr("dx", "-.8em")
    .attr("dy", "-.55em")
    .attr("transform", "rotate(-90) ");
  svg.append("g")
    .attr("class", "y axis")
    .call(yAxis);
  svg.selectAll(".rectangle")
    .data(data)
    .enter()
    .append("rect")
    .attr("class", "rectangle")
    .attr("width", width / data.length)
    .attr("height", function(d){return height - y(+d[selection]);})
    .attr("x", function(d, i){return (width / data.length) * i;})
    .attr("y", function(d){return y(+d[selection]);})
    .append("title")
    .text(function(d){return d.State + " : " + d[selection];});
  var selector = d3.select("#drop")
    .append("select")
    .attr("id", "dropdown")
    .on("change", function(d){
      selector = document.getElementById("dropdown");
      y.domain([0, d3.max(data, function(d){return +d[selection.value]}));
      yAxis.scale(y);
      d3.selectAll(".rectangle")
        .transition()
        .attr("height", function(d){return height - y(+d[selection.value]);})
        .attr("x", function(d, i){return (width / data.length) * i;})
        .attr("y", function(d){return y(+d[selection.value]);})
        .ease("linear")
        .select("title")
        .text(function(d){return d.State + " : " + d[selection.value];});
      d3.selectAll(".x,y.axis")
        .transition()
        .call(xAxis);
    });
  selector.selectAll("option")
    .data(elements)
    .enter()
    .append("option")
    .attr("value", function(d){return d;})
    .text(function(d){return d;})
  });
});

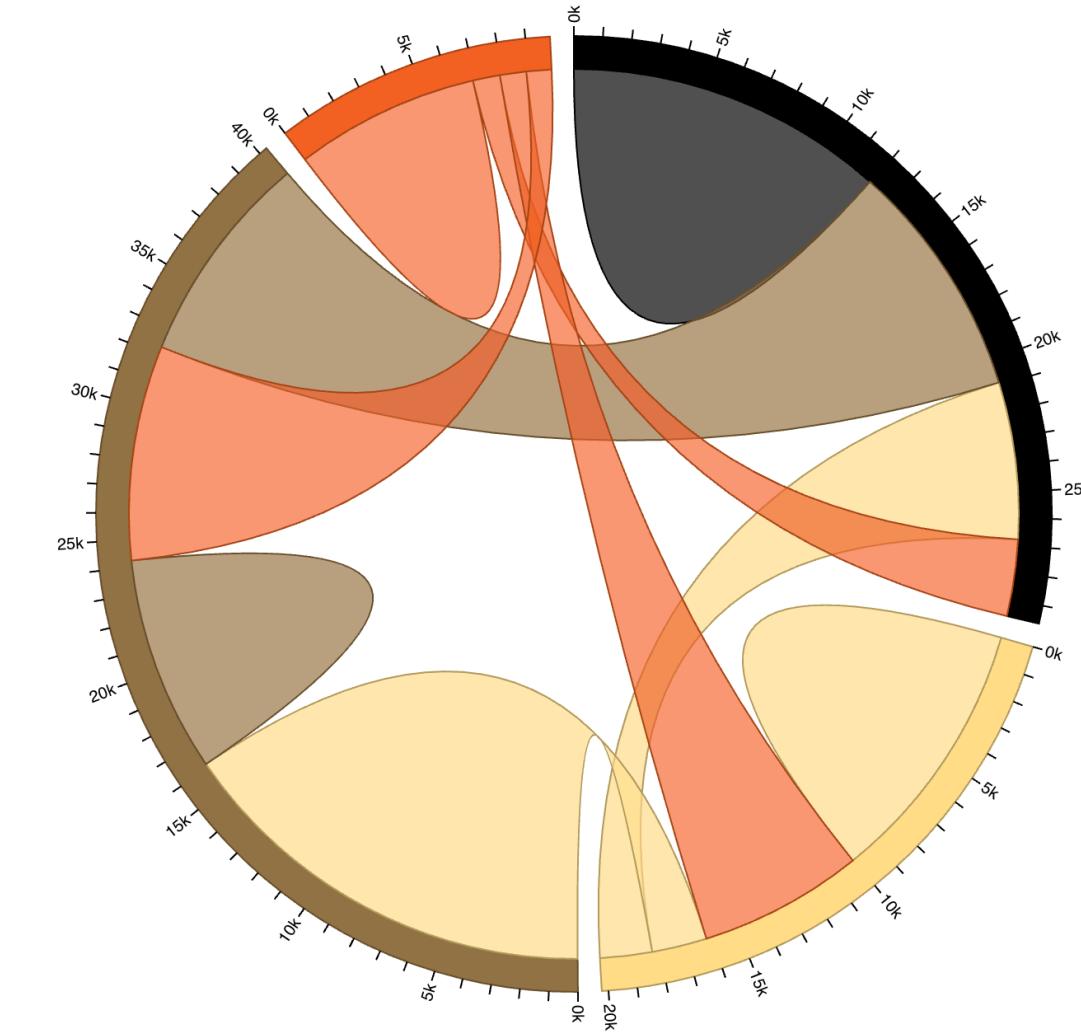
```



Protopvis & D3



Protopvis
Low(er) ceiling



D3
High(er) ceiling

Compared to excel, etc., both have a high ceiling
But both have a pretty high threshold!

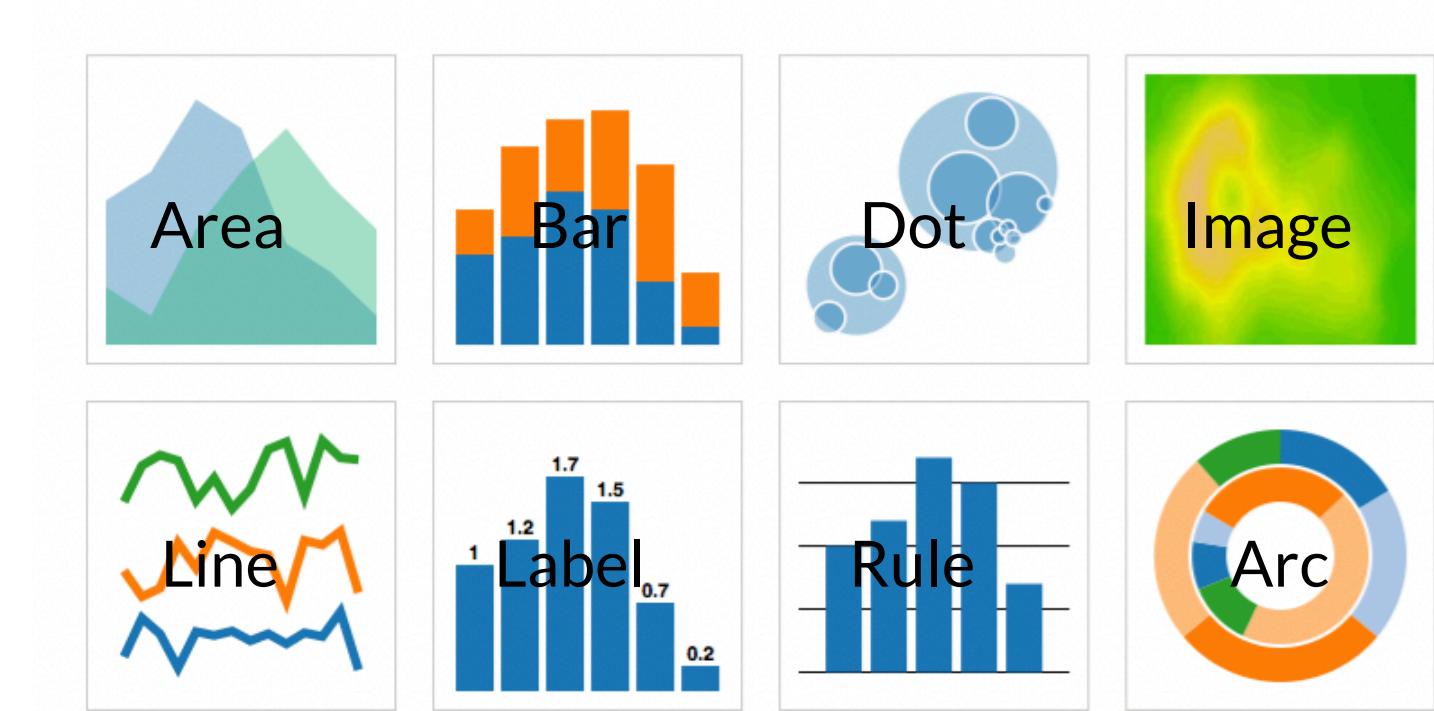
Vega-Lite: lowering the threshold

Lowering the threshold

- Goal: “create an *expressive* (high ceiling) yet *concise* (low threshold) declarative language for specifying visualizations”

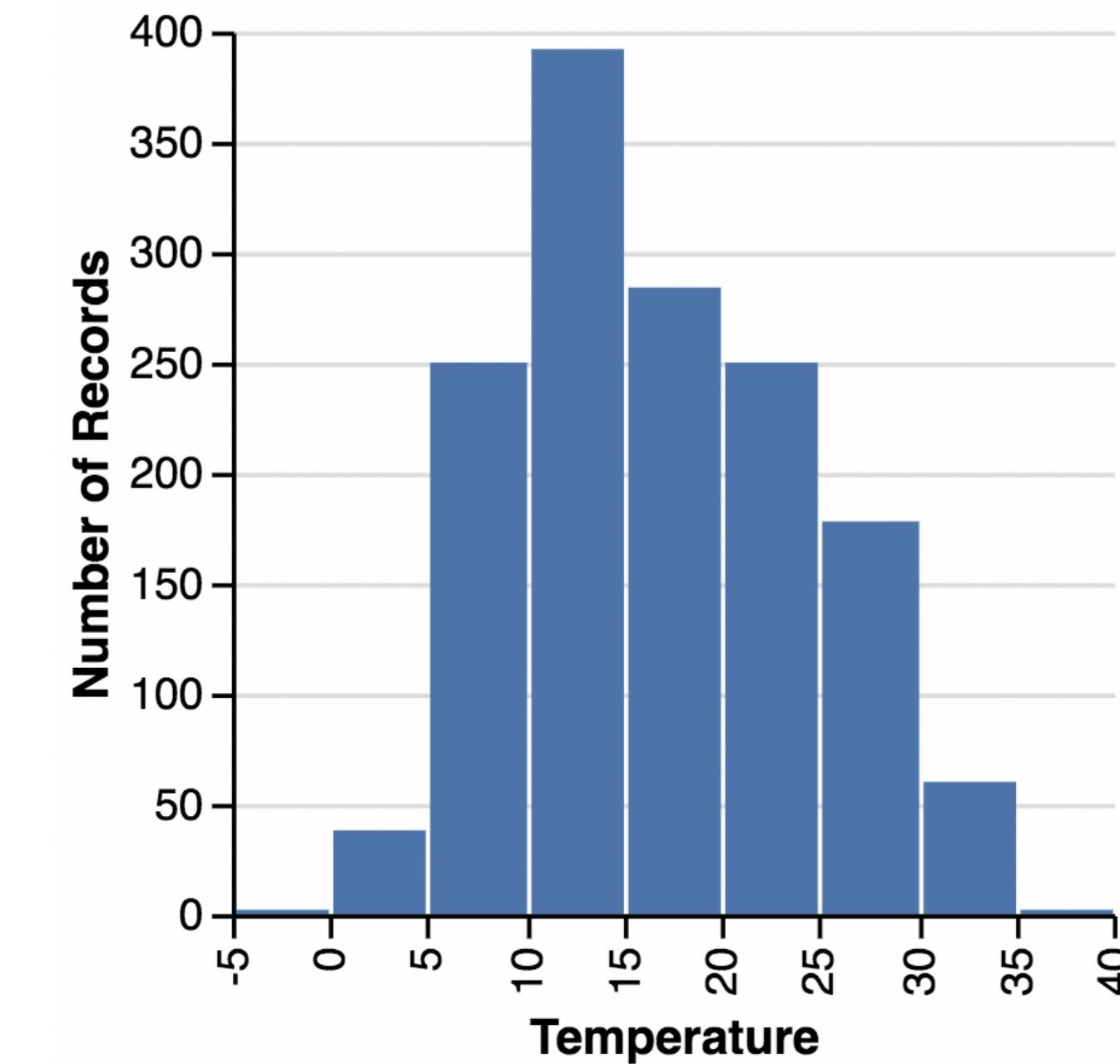
Vega-Lite

- Grammar of graphics
 - Data: input data to visualize
 - Mark: Data-representative graphics
 - Transform: whether to filter, aggregate, bin, etc.
 - Encoding: mapping between data and mark properties
 - Scale: map between data values and visual values
 - Guides: axes & legends that visualize scales



Vega-lite

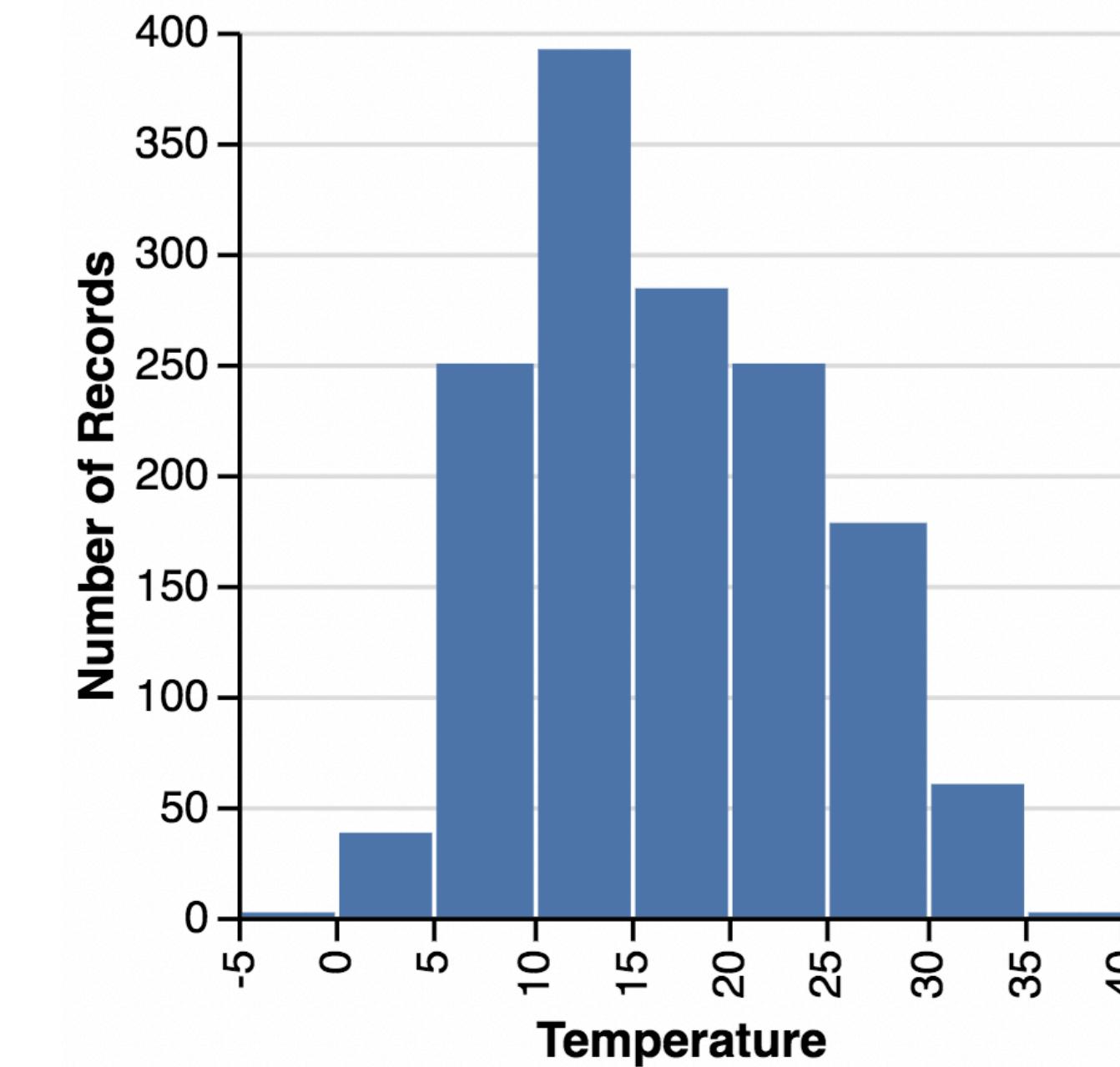
Making a histogram



Vega-lite

Histogram = (Bar with x=binned field, y=count)

- Bin records by their temperature
- Count how many records fall into each bin
- Render those bins as vertical bars

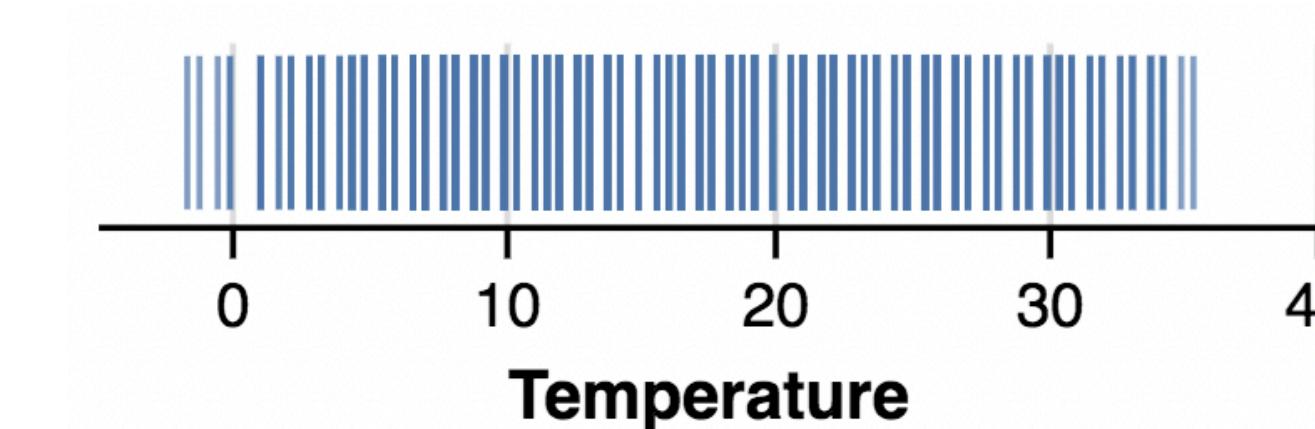


Vega-lite

Histogram = (Bar with x=binned field, y=count)

```
{  
  data: {url: "weather-seattle.json"},  
  mark: "tick", ← Set mark as a tick  
  encoding: {  
    x: {  
      field: "temperature", ← Encode x according to the  
      type: "quantitative" "temperature" field  
    }  
  }  
}
```

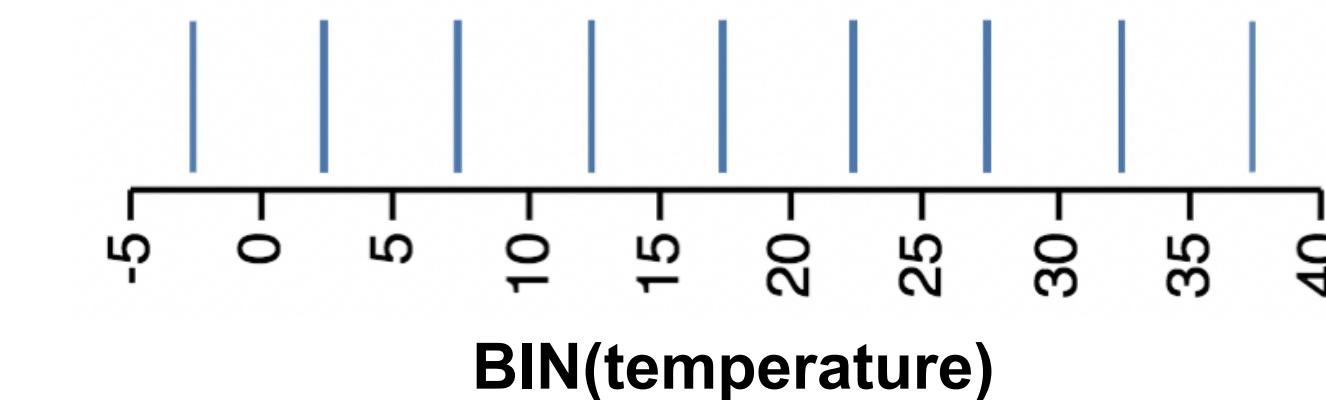
↑
Four types:
quantitative (numerical)
temporal (time)
ordinal (ordered)
nominal (categorical)



Vega-lite

Histogram = (Bar with x=binned field, y=count)

```
{  
  data: {url: "weather-seattle.json"},  
  mark: "tick",  
  encoding: {  
    x: {  
      bin: true, ←Bin values by x dimension  
      field: "temperature",  
      type: "quantitative"  
    }  
  }  
}
```

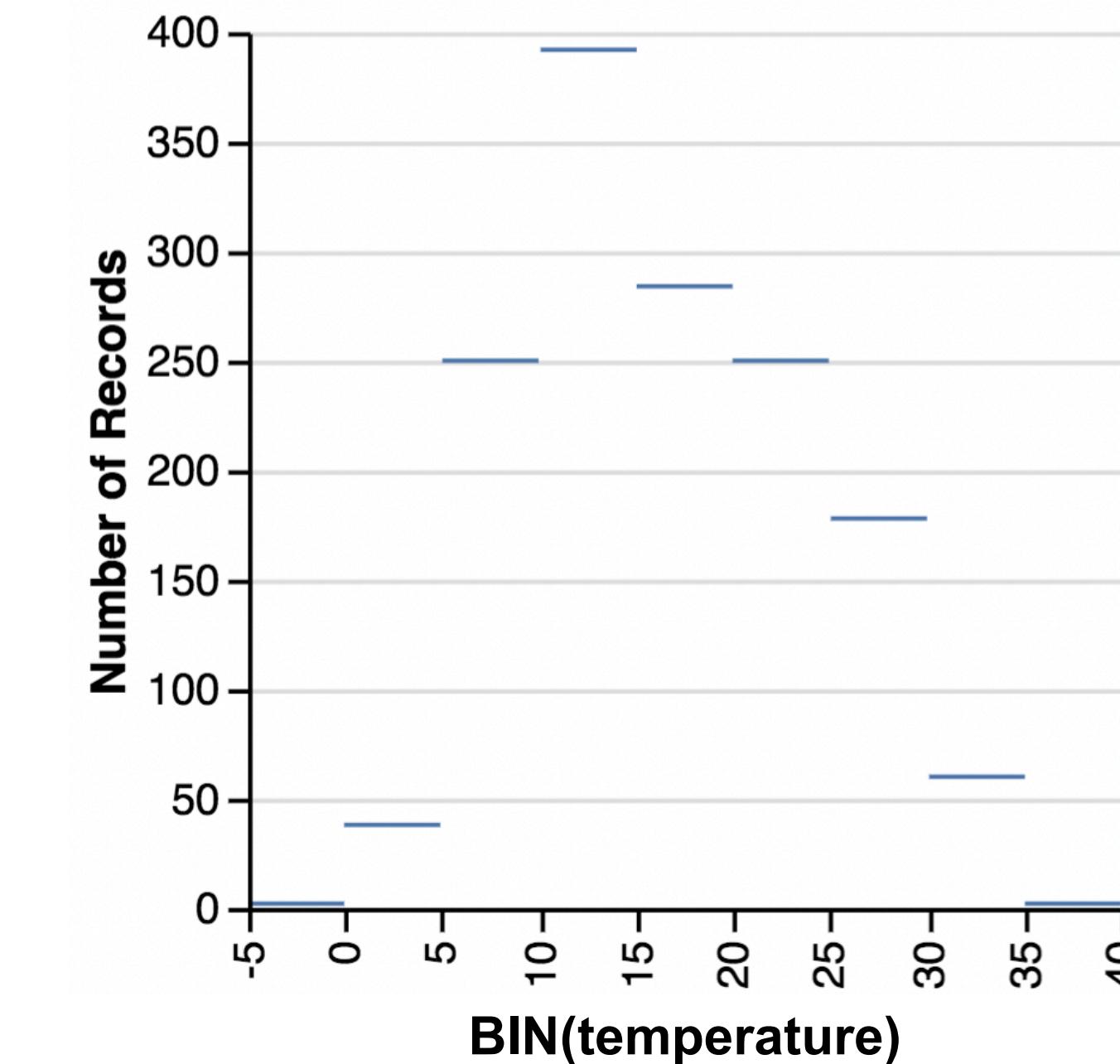


Vega-lite

Histogram = (Bar with x=binned field, y=count)

```
{  
  data: {url: "weather-seattle.json"},  
  mark: "tick",  
  encoding: {  
    x: {  
      bin: true,  
      field: "temperature",  
      type: "quantitative"  
    },  
    y: {  
      aggregate: "count",  
      type: "quantitative"  
    }  
  }  
}
```

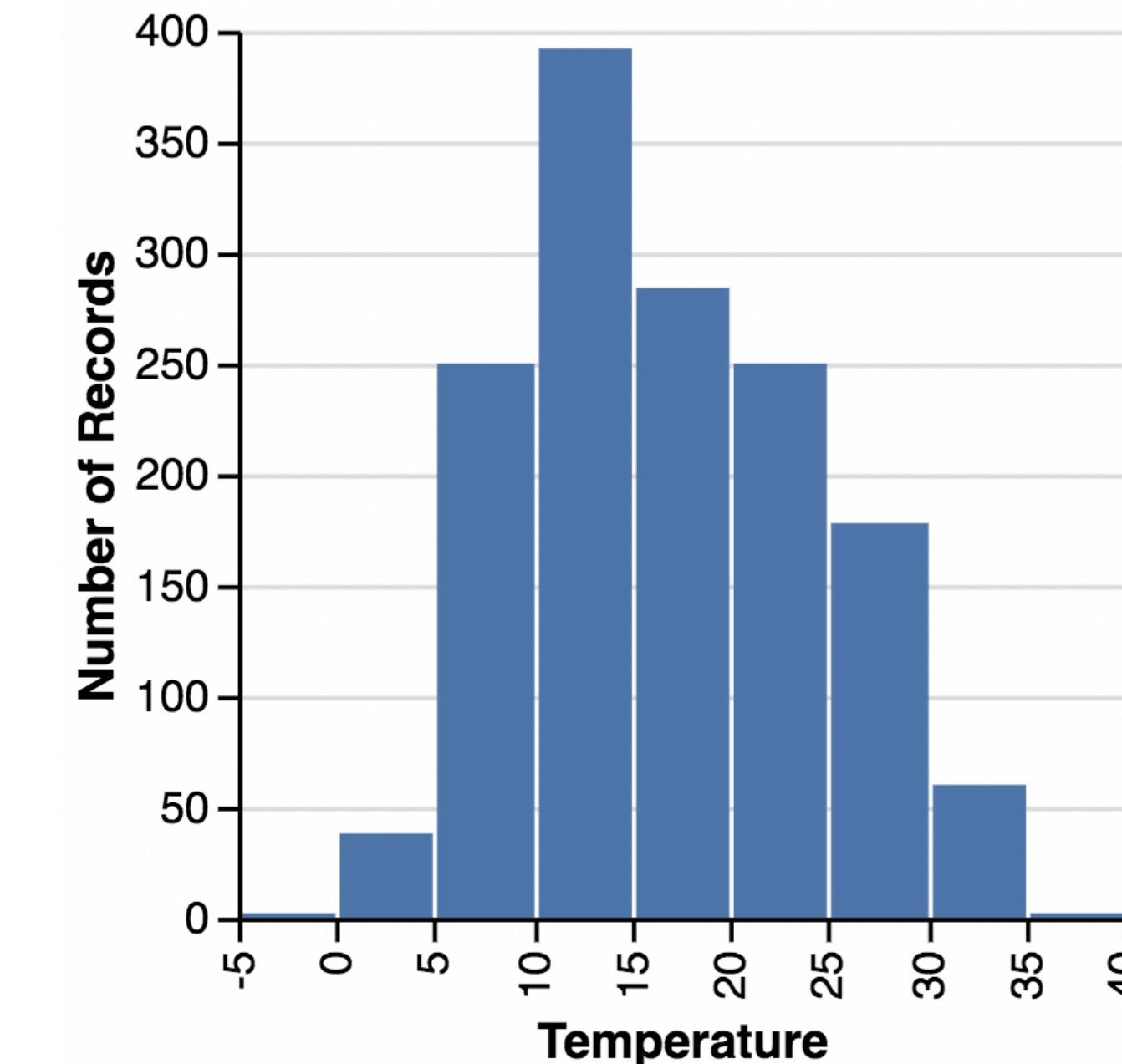
y should aggregate
the bins by counting
how many values
are in them



Vega-lite

Histogram = (Bar with x=binned field, y=count)

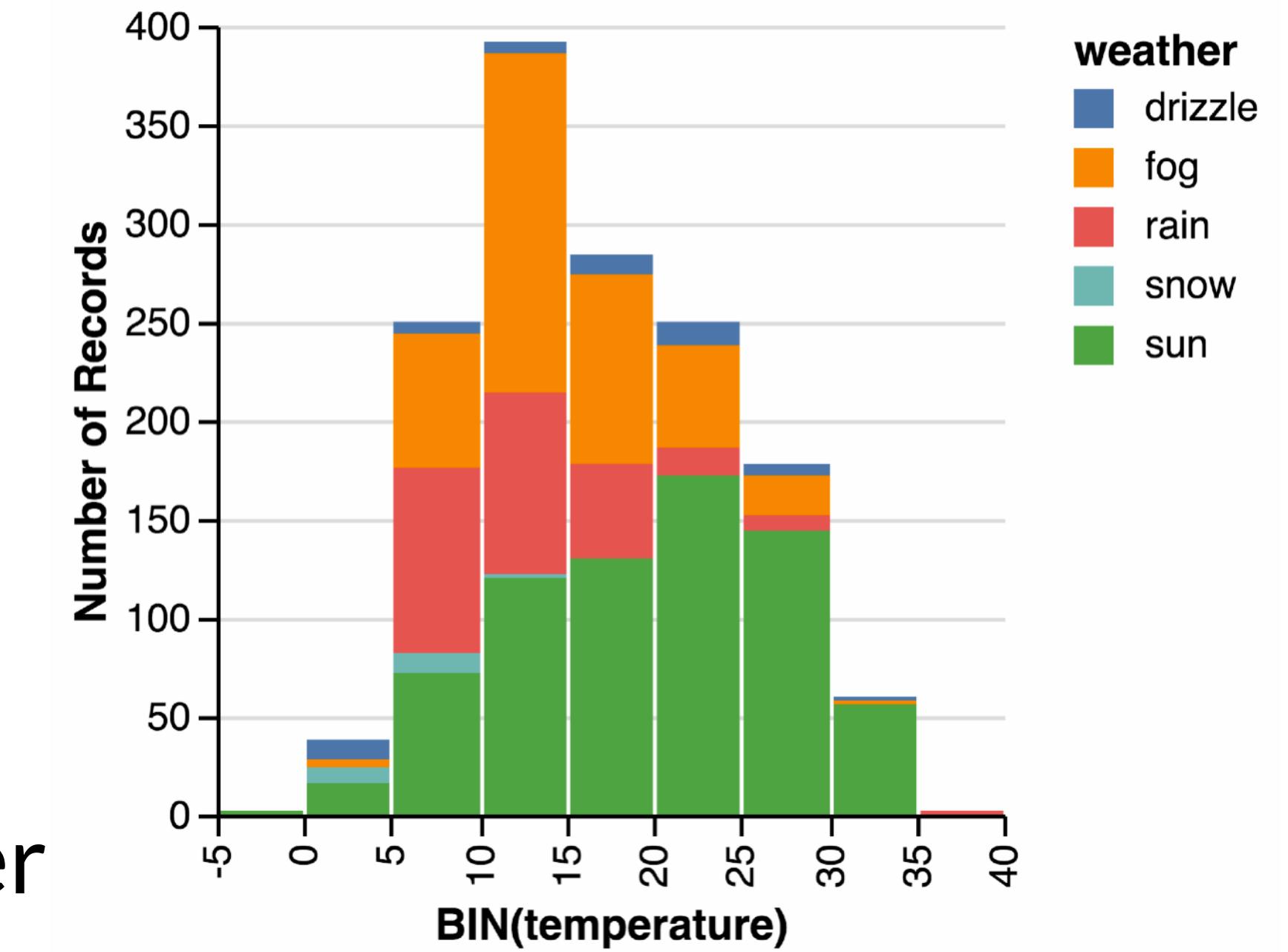
```
{  
  data: {url: "weather-seattle.json"},  
  mark: "bar", ←Change the mark to a bar  
  encoding: {  
    x: {  
      bin: true,  
      field: "temperature",  
      type: "quantitative"  
    },  
    y: {  
      aggregate: "count",  
      type: "quantitative"  
    }  
  }  
}
```



Vega-lite

Histogram + Color

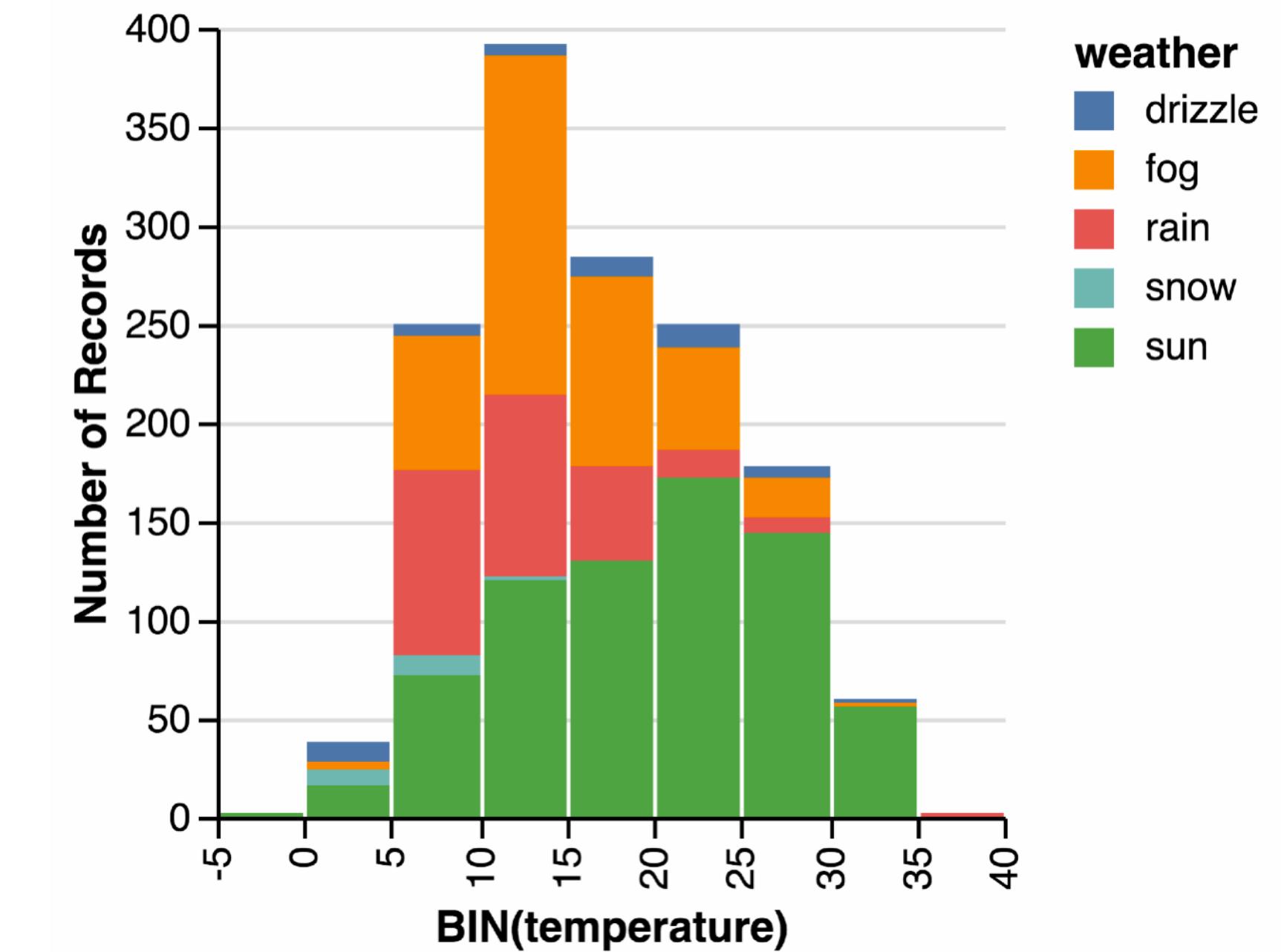
```
{  
  data: {url: "weather-seattle.json"},  
  mark: "bar",  
  encoding: {  
    x: {  
      bin: true,  
      field: "temperature",  
      type: "quantitative"  
    },  
    y: {  
      aggregate: "count",  
      type: "quantitative"  
    },  
    color: { ←Set the color to follow the weather  
      field: "weather",  
      type: "nominal"  
    }  
  }  
}
```



Vega-lite

“Sensible defaults”

- The field chose reasonable defaults for presenting the data
 - We didn't specify what colors to use
 - Or how wide bins should be
 - Or how to label the axes
 - Or that the bars should be stacked
 - ...

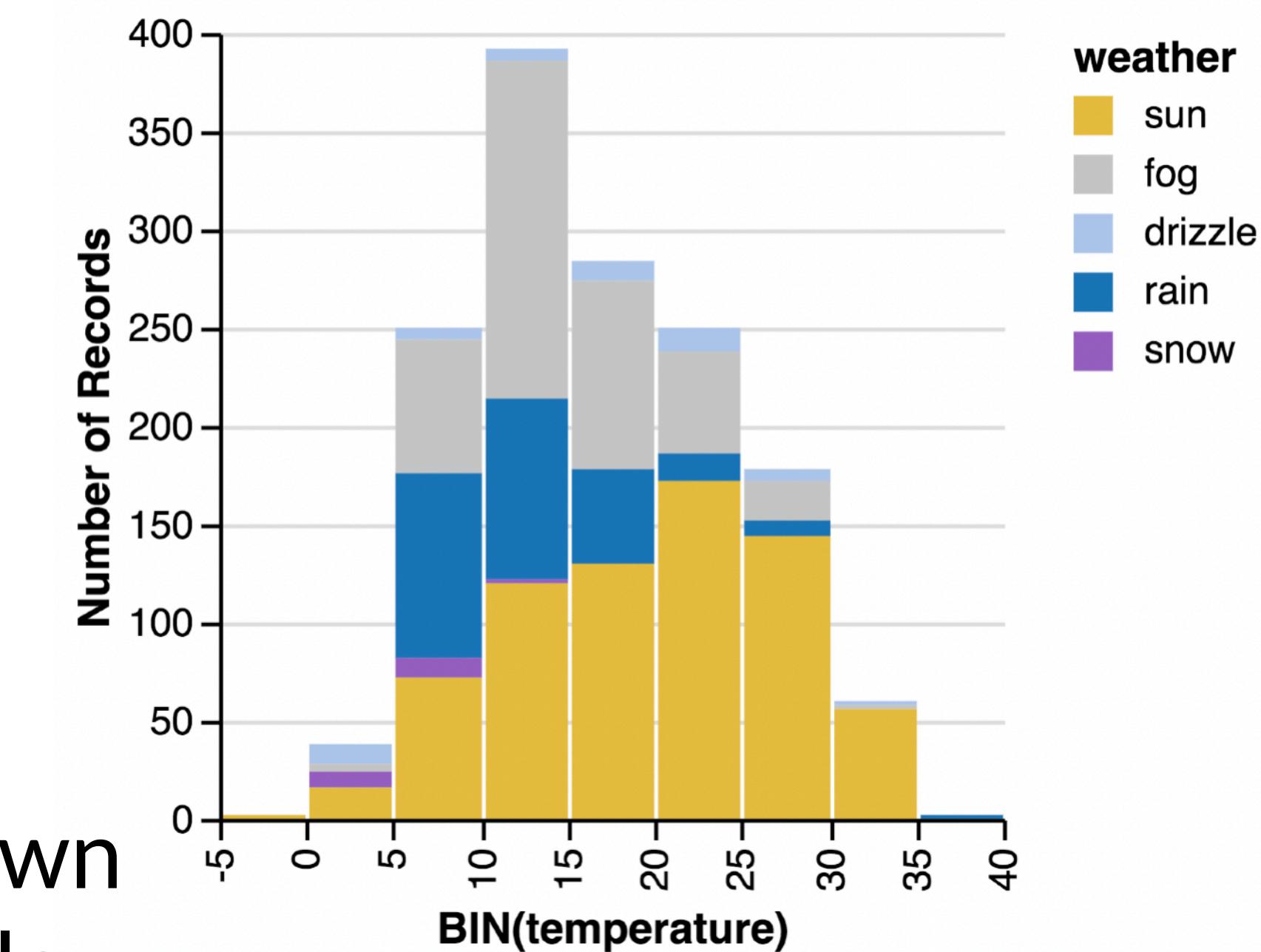


Vega-lite

Overriding the sensible defaults

```
{  
  data: {url: "weather-seattle.json"},  
  mark: "bar",  
  encoding: {  
    x: {  
      bin: true,  
      field: "temperature",  
      type: "quantitative"  
    },  
    y: {  
      aggregate: "count",  
      type: "quantitative"  
    },  
    color: {  
      field: "weather",  
      type: "nominal"  
    },  
    scale: {  
      domain: ["sun", "fog", "drizzle", "rain", "snow"],  
      range: ["#e7ba52", "#c7c7c7", "#aec7e8",  
              "#1f77b4", "#9467bd"]  
    }  
  }  
}
```

Set our own
color scale



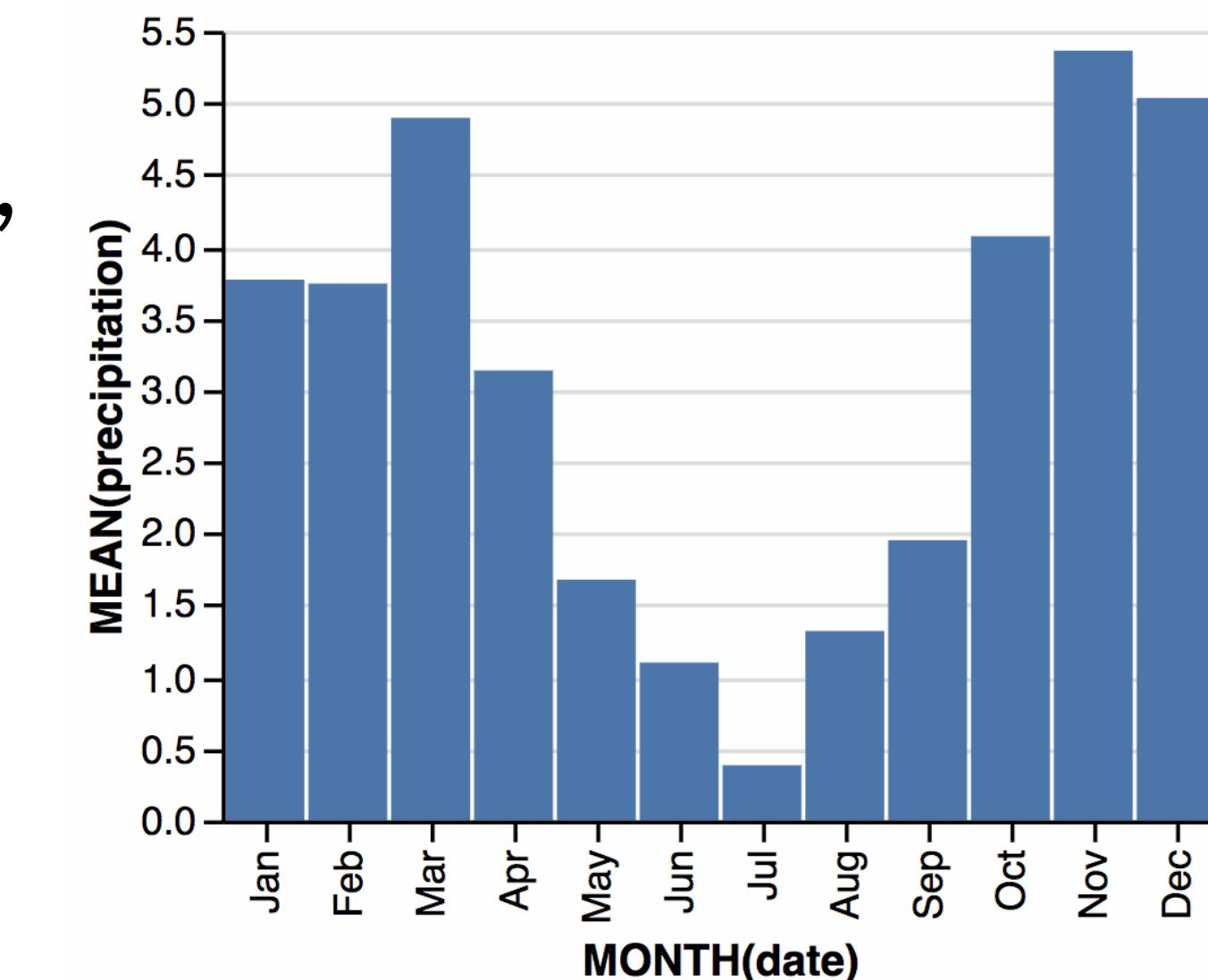
Vega-lite

Monthly precipitation

```
{  
  data: {url: "weather-seattle.json"},  
  mark: "bar",  
  encoding: {  
    x: {  
      timeUnit: "month", field: "date",  
      type: "quantitative"  
    },  
    y: {  
      aggregate: "mean", field: "precipitation",  
      type: "quantitative"  
    }  
  }  
}
```

Field is a date string ("2018-10-17"),
but display and bin it as a month

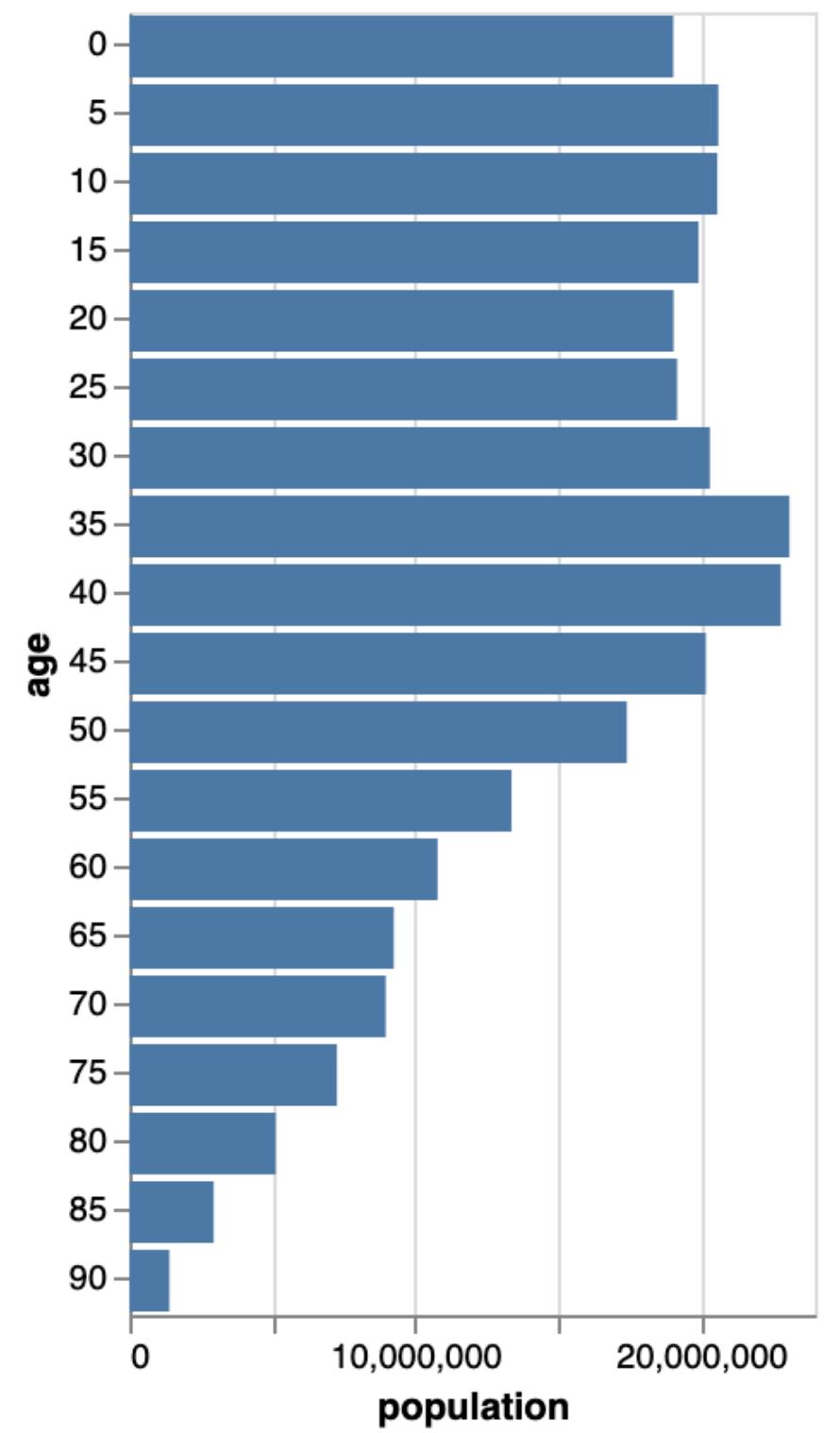
Aggregate by the mean



Vega-lite demo



U.S. age distribution in 2000



Question

How would this visualization
be specified in Vega-Lite?
(Assume field names are correct)

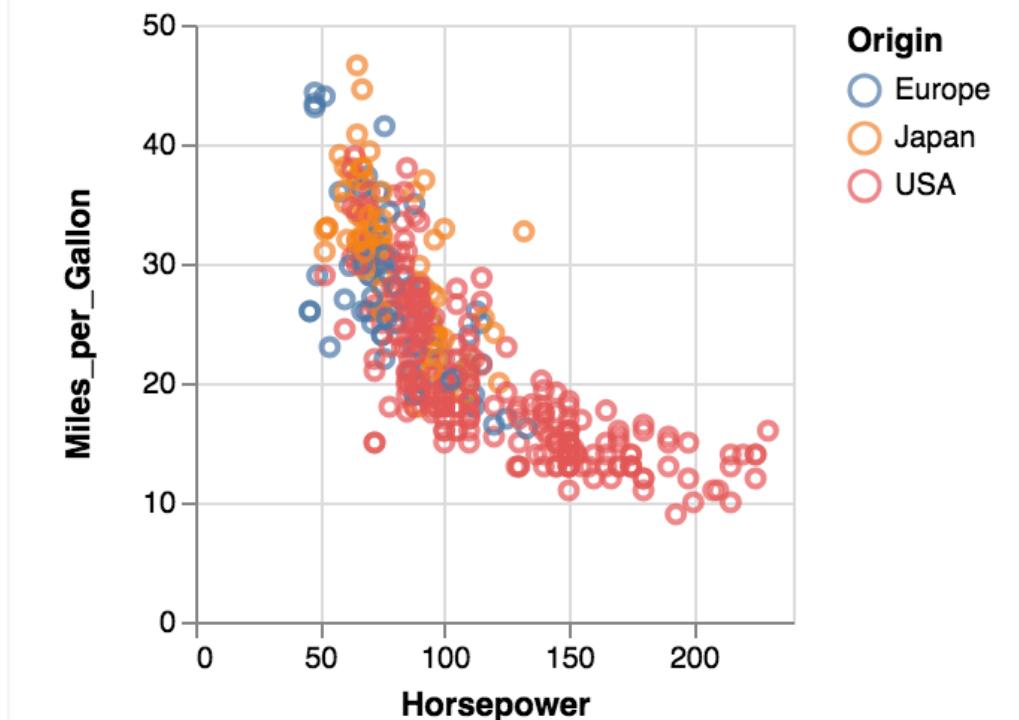
A mark: "point",
encoding: {
 x: {field: "MPG", type: "ordinal"},
 y: {field: "Horsepower", type: "ordinal"},
 color: {field: "Origin", type: "nominal"}
}

B mark: "point",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "nominal"}
}

D mark: "point",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "ordinal"}
}

C mark: "ticks",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "nominal"}
}

E mark: "ticks",
encoding: {
 x: {field: "MPG", type: "ordinal"},
 y: {field: "Horsepower", type: "ordinal"},
 color: {field: "Origin", type: "nominal"}
}



Question

How would this visualization
be specified in Vega-Lite?
(Assume field names are correct)

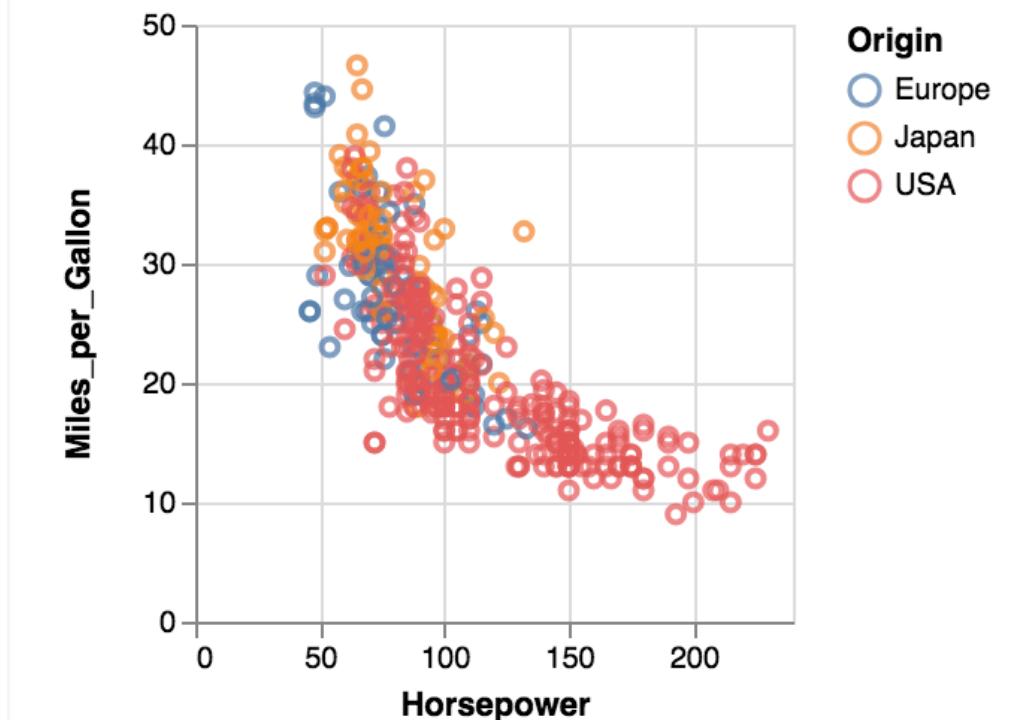
A mark: "point",
encoding: {
 x: {field: "MPG", type: "ordinal"},
 y: {field: "Horsepower", type: "ordinal"},
 color: {field: "Origin", type: "nominal"}
}

B mark: "point",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "nominal"}
}

D mark: "point",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "ordinal"}
}

C mark: "ticks",
encoding: {
 x: {field: "Horsepower", type: "quantitative"},
 y: {field: "MPG", type: "quantitative"},
 color: {field: "Origin", type: "nominal"}
}

E mark: "ticks",
encoding: {
 x: {field: "MPG", type: "ordinal"},
 y: {field: "Horsepower", type: "ordinal"},
 color: {field: "Origin", type: "nominal"}
}



Sensible defaults: Vega-lite's secret

- Threshold is lower
 - More concise definitions, less to understand up front
- Ceiling remains the same
 - The sensible defaults can be overridden
- A downside: visualizations made with Vega-Lite look similar
 - The path of least resistance Vega-Lite provides influences what visualizations people make and what they look like

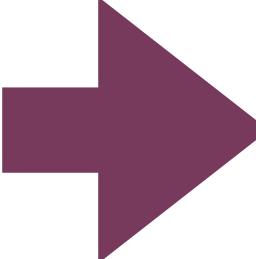
Downside: path of least resistance

- The path of least resistance: tools influence what is created
- Sensible defaults make Vega-Lite visualizations look similar to one another
 - These defaults *can* be overwritten, but are they in practice?
- Similar concern to the widespread adoption of grid frameworks

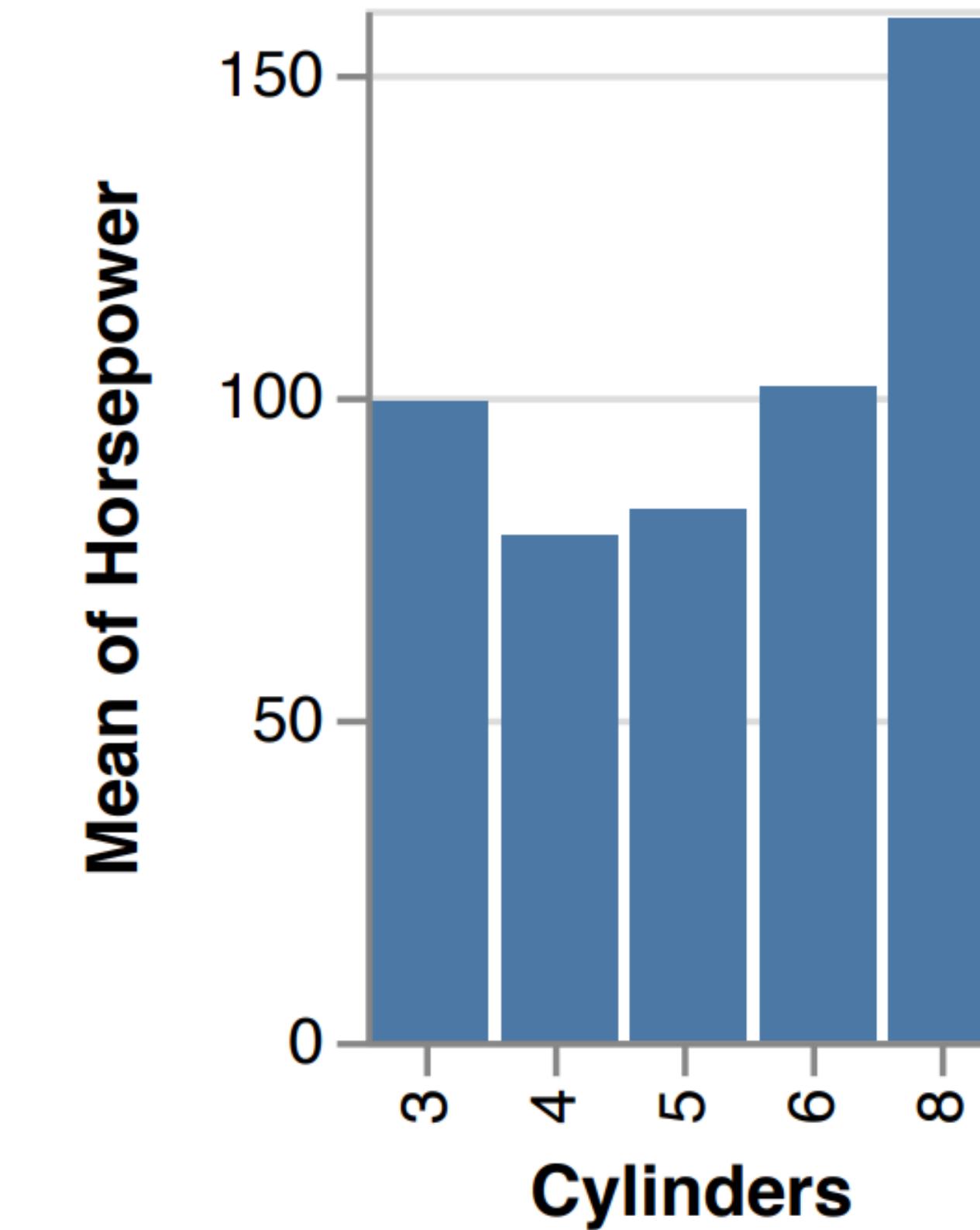
Draco

- Given a dataset and what fields to plot, use constraints to select the best mark type and data type
 - Transpiles to a Vega-Lite specification

```
data(cars).  
encoding(e0).  
field(e0, cylinders).  
encoding(e1).  
field(e1, horsepower).  
aggregate(e1, mean)
```



```
data: {url: "cars.csv"},  
mark: "bar",  
encoding: {  
  x: {  
    field: "Cylinders",  
    type: "ordinal"  
  },  
  y: {  
    field: "Horsepower",  
    type: "quantitative",  
    aggregate: "mean"  
  }  
}
```



Today's goals

By the end of today, you should be able to...

- Write code which follows object-oriented principles in TypeScript
- Explain the advantages and disadvantages of using TypeScript
- Describe the concepts of threshold and ceiling in software tools and what tool designers should be striving to create
- Explain the relative threshold and ceilings of visualization tools like Protopis, D3, and Vega-Lite
- Describe common visualization primitives like marks, axes, and scales
- Implement simple visualizations with Vega-Lite

IN4MATX 133: User Interface Software

Lecture 7:
TypeScript &
Data Visualization Tools

Professor Daniel A. Epstein
TA Lucas de Melo Silva
TA Jong Ho Lee

A few more TypeScript details

Type declaration files

- Because types get stripped out when transpiling, a “declaration file” (.d.ts) can be created
 - Important when someone else will use your code as a library
 - Declaration file helps their code check types in your library
 - `tsc --declaration test.ts`
- You can install typings declarations for many libraries from npm
 - `npm install --save @types/your-library-here`

Type declaration files

```
//test.ts
function area(shape: string, width: number, height: number):string {
    var area:number = width * height;
    return "I'm a " + shape + " of area " + area + " cm^2.";
}

document.body.innerHTML = area('square', 15, 15);

// transpiled test.js
function area(shape, width, height) {
    var area = width * height;
    return "I'm a " + shape + " of area " + area + " cm^2.";
}
document.body.innerHTML = area('square', 15, 15);

// generated test.d.ts
declare function area(shape: string, width: number, height: number): string;
```

Interfaces

- Just like in Java, describes the “inputs” and “outputs” of an object

```
interface Shape {  
    name: string;  
    width: number;  
    height: number;  
    color?: string; // ? Specifies an "optional" value  
}  
  
function area(shape : Shape):string {  
    var area = shape.width * shape.height;  
    return "I'm " + shape.name + " with area " + area + " cm squared";  
}  
  
console.log(area({name: "rectangle", width: 30, height: 15}));  
console.log(area({name: "square", width: 30, height: 30, color: "red"}));
```

Inheritance

- Like in Java, classes and interfaces can be extended

```
class Shape3D extends Shape {  
    volume: number;  
  
    constructor (name: string, width: number, height: number, length: number ) {  
        super( name, width, height ); //calls base class constructor  
        this.volume = length * this.area;  
    };  
  
    shoutout() { //overrides the base class  
        return "I'm " + this.name + " with a volume of " + this.volume + " cm cube.";  
    }  
  
    superShout() { //calls base class shoutout method  
        return super.shoutout();  
    }  
}  
  
var cube = new Shape3D("cube", 30, 30, 30);  
console.log( cube.shoutout() );  
console.log( cube.superShout() );
```

Generics

- Also work the same as Java

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

```
let output = identity<string>("myString"); // type of  
output will be 'string'  
let output = identity("myString"); // type of output  
will be 'string'
```

<https://www.typescriptlang.org/docs/handbook/generics.html>