ROMA TRE
UNIVERSITÀ DEGLI STUDI

iNGEGNERIA
CIVILE, INFORMATICA E DELLE TECNOLOGIE AERONAUTICHE

Information Visualization

Infovis on the Web

The D3.js library

G. Da Lozzo, V. Di Donato, M. Patrignani, C. Squarcella

Copyright notice

Graphic Web libraries

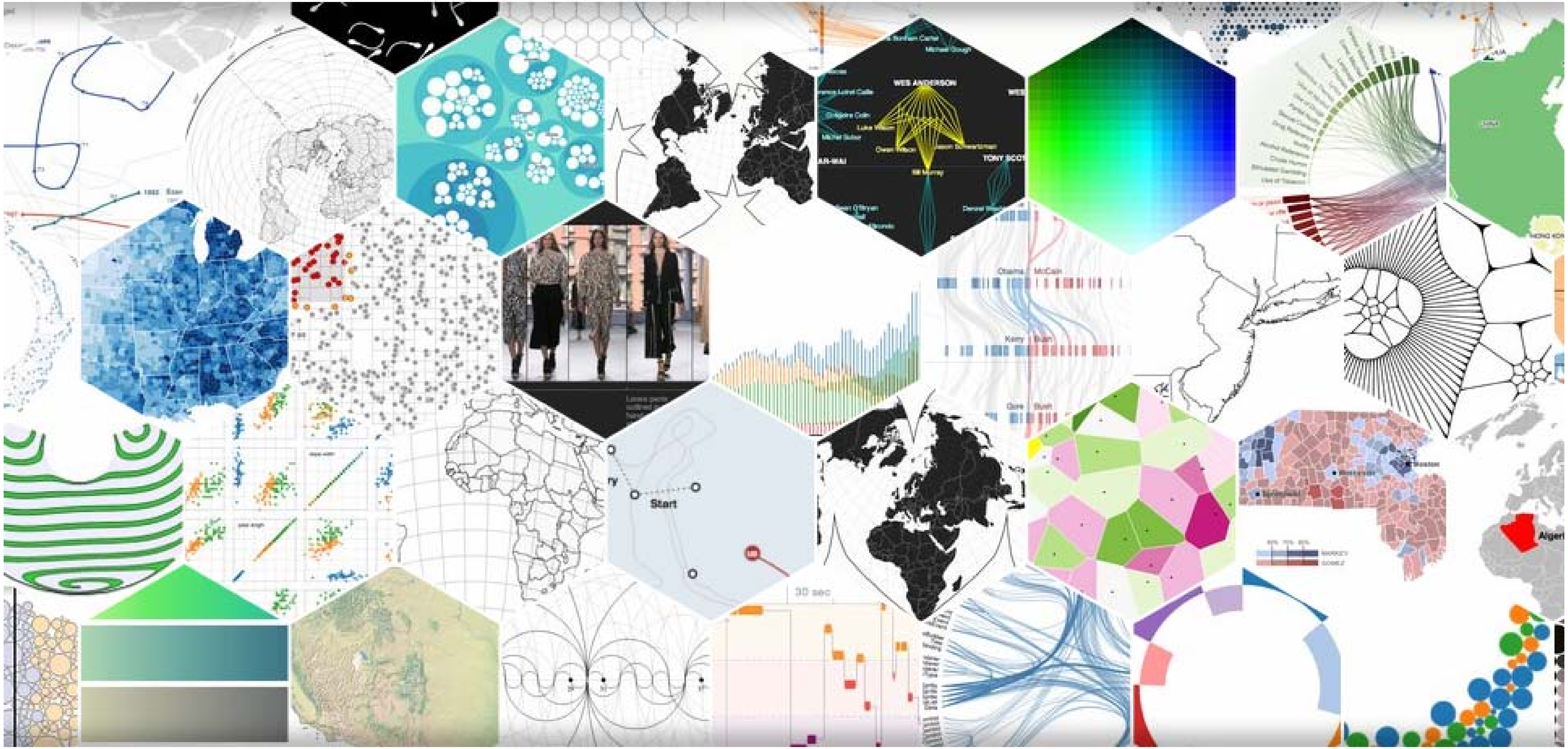
- Serveral libraries are available for managing 2D and 3D graphics on the Web
- Some of them use WebGL
  - a dialect of HTML5 canvas designed to intensively exploit the client hardware

2D		3D
SVG	Canvas	WebGL
D3.js	Paper.js	Three.js
Raphaël.js	Kinetic.js	Copperlicht
SVG.js	Fabric.js	O3D
BonsaiJS	Easel.js	…
…	…	…

D3.js

- D3.js (Data-Driven Documents) is a JavaScript library for producing dynamic, interactive data visualizations in web browsers
  - developed by Mike Bostock and others at Stanford
  - open source
  - uses SVG, HTML5, and CSS
  - big community and well documented ☺
    - https://github.com/mbostock/d3/wiki/API-Reference
    - https://github.com/mbostock/d3/wiki
    - https://groups.google.com/forum/#!forum/d3-js
    - http://stackoverflow.com/questions/tagged/d3.js
  - known to have a quite steep learning curve ☹

What is D3.js for?



[http://d3js.org/]

D3.js in brief

- D3.js can (among other things):
  - bind data to DOM elements
    - when the data change, the elements of the DOM and their properties change
    - this is why its name is “data driven documents”
  - handle interaction and animation
    - decide what happens when the user interacts with the graphics
    - decide the timings of the changes
  - generate SVG on-the-fly

D3.js history

- 2009
  - Protovis, the library that inspired D3.js, is first released
- 2011
  - version 2.0 of D3.js is released
    - and Protovis project is abandoned
- 2012
  - version 3.0 released
- 2016
  - version 4.0 released
- 2018
  - version 5.0 released
- 2020
  - version 6.0 released

Summary

- Installation and configuration
- Support for arrays
- D3 Collections
  - Objects conversion
  - Maps, Sets, Nests
- Scales
- Data Join, data transformations
- Example application: Enter, Exit, Update
- Layouts

Installing: accessing from the Web

- Directly link the latest release:

```
<script src="http://d3js.org/d3.v7.min.js"></script>
```

- The `d3` object (together with its properties and its methods) is now available to your scripts
- Pros
  - very easy
  - the latest D3.js release of version 7 is downloaded
    - currently 7.8.4 (Apr 2023)
- Cons
  - it does not work when offline
  - accessing the official d3js.com website may be slower

Installing: accessing locally (1/2)

- Download the file **`d3.v7.min.js`** from **`http://d3js.org`**
  - the word “min” in the file name means that this file is “minified”
    - it is equivalent to the full version **`d3.v7.js`**
    - it is lighter (273 KB versus 573 KB)
    - it is not human readable
- Load the file from your html page
- Pros
  - works even when offline
- Cons
  - you are not sure to get the latest release

Installing: accessing locally (2/2)

- An example
  - suppose that file **`d3.v7.min.js`** is stored in the same directory where this html file is

```
<!DOCTYPE html>
<html>
<head>
<title>D3 Test</title>
<script type="text/javascript" src="d3.v7.min.js"></script>
</head>
<body>
<script type="text/javascript">
                // Here we can use the d3 object!!
</script>
</body>
</html>
```


Supported browsers

- From version 5 on, D3.js is compatible with all modern browsers
  - Chrome, Edge, Firefox, and Safari
    - we need a browser that supports SVG and CSS3 Transitions
- Since version 4 D3.js also supports IE 9+
- Parts of D3.js may work in older browsers
  - “everything except IE 8 and older versions”

Opening an HTML file using D3.js

- Note that your browser may enforce strict security permissions for reading files from the local file system
  - after directly opening an .html file from your file system you will not be allowed to load a .js file from the same directory
- Workaround for Google Chrome
  - *google-chrome --allow-file-access-from-files mio_file.html*
- Workaround for Mozilla Firefox
  - type in the address bar “about:config”
  - click on the button “I accept the risks!” that warns you against dangerous configuration changes
  - search for “privacy.file_unique_origin” and change the value to “false”

Opening an HTML file using D3.js

- The best workaround is to configure a local Web server
  - launching a local Web server in python
    - *python -m SimpleHTTPServer 8888*
    - *python3 -m http.server 8888*
  - launching a local Web server with php
    - *php -s 127.0.0.1:8888*
  - launching a local Web server with Node.js
    - first install: npm install -g http-server
    - then run: http-server &
      - the default port is 8080
  - access “http://localhost:8888” from your browser

Supported platforms

- D3.js also runs on Node.js
  - use "npm install d3" to install it
- Node.js itself lacks a DOM
  - likely, there exist multiple DOM implementations for NODE (e.g., JSDOM)
  - you'll need to explicitly pass in a DOM element to your d3 object

```
var d3 = require("d3"),
    jsdom = require("jsdom");

var document = jsdom.jsdom(),
    svg = d3.select(document.body).append("svg");
```


D3.js and arrays

- Since arrays are the canonical data representation in D3.js, the library provides a set of utilities for arrays

common use

d3.**min**(*array*[, *accessor*])
d3.**max**(*array*[, *accessor*])
d3.**extent**(*array*[, *accessor*])
d3.**sum**(*array*[, *accessor*])
…

```
d3.extent([4,6,7,8,3])  // [3, 8]
```

from probability theory and statistics

d3.**mean**(*array*[, *accessor*])
d3.**median**(*array*[, *accessor*])
d3.**deviation**(*array*[, *accessor*])
d3.**variance**(*array*[, *accessor*])
…

```
d3.variance([4,6,7,8,3])  // 4.3
```


Why accessor functions?

d3.**min**(*array*[, *accessor*]) ← ?

- Accessor functions are needed when
  - the input array does not directly contain numbers, but objects with several properties
  - you need to access the value of some property of the objects
- Accessor functions are callback functions
  - they take an object as parameter
  - they produce a value

Accessor function example

- Suppose you have the following array

```
var people = [
  { name: "Valentino", age: 25, height:1.7},
  { name: "Giordano",  age: 32, height:1.8}]
```

- You want to find the youngest person

```
d3.min(people, function(element){
 return element["age"]
})       // 25
```

- You want to find the tallest person

```
d3.max(people, function(element){
 return element["height"]
})       // 1.8
```


Without accessor functions?

- It would be equivalent to call array.map(callbackFunction) before computing the minimum/maximum value
  - array.map() is a built-in JavaScript method that returns an array obtained by calling the callbackFunction on each element of the array

```
myArray = people.map(function(el){return el['age’]})        // [25, 32]
d3.min(myArray)     // 25
```


Ordering arrays

- The default order used by the JavaScript array.sort() mutator is lexicographic (that is, alphabetical) and not natural
  - this can lead to unexpected behaviors when sorting an array of numbers

```
[2,1005,10000].sort()
//  the array becomes [10000, 1005, 2]
```

- D3.js array functions, instead, compare elements using their natural order

```
d3.min([“20”,“3”]);            // outputs “20”
d3.min([20,3]);                // outputs 3
d3.min([21,“3”,“200”]);        // outputs “200”
```


D3.js comparator functions

- The method `d3.ascending(a, b)` returns -1, 1, or 0, depending whether *a* is smaller, greater, or equal than *b*
- Analogously you have `d3.descending(a, b)`
- Example of use

```
d3.ascending(5,2)    // outputs 1
```

- Example of use in conjunction with JavaScript built-in array.sort() mutator

```
[5,2,3,6,7].sort(d3.ascending)   // [2, 3, 5, 6, 7]
[5,2,3,6,7].sort(d3.descending)  // [7, 6, 5, 3, 2]
```


Array transformations

- D3.js offers some additional helpers for transforming arrays and for generating new arrays
  - d3.**merge**(*arrays*)
    - concatenates the input arrays
    - example:
```
d3.merge([[1],[2,3]]);      // [1,2,3]
```
  - d3.**pairs**(*array*)
    - returns an array of adjacent pairs
    - example:
```
d3.pairs([1,2,3,4]);        // [[1,2],[2,3],[3,4]]
```


Array transformations

- Other D3.js helpers for arrays
  - d3.**cross**(*arrays*)
    - returns all combinations
    - example:

```
d3.cross([1,2],["x", "y"]);      // [[1,"x"],[1,"y"],[2,"x"],[2,"y"]]
```

  - d3.**zip**(*arrays*)
    - returns the colums of a matrix provided by rows
    - example:

```
d3.zip([1,2],[3,4],[5,6]);      // [[1,3,5],[2,4,6]]
```

  - further array operators can be found at https://github.com/d3/d3-array

D3.js collections

- Objects
  - methods that convert associative arrays (objects) to standard arrays
- Maps and sets
  - similar to ES6 Maps and Sets, but with string keys and a few other differences
- Nests
  - allow the programmer to group data into arbitrary hierarchies

JavaScript objects

- JavaScript objects are associative arrays, i.e., collections of (key, value) pairs
- The “for...in” statement iterates over the properties of an object
  - properties are processed in arbitrary order

```
for (let variable in object) {
...
}
```

variable → a different **property name** is assigned to *variable* on each iteration

object → the object whose properties are iterated

Converting objects to arrays

- D3.js provides several operators for converting objects to standard arrays
  - the order of the output array is undefined

d3.**keys**(*object*)
- returns an array of the property names

d3.**values**(*object*)
- returns an array of the property values

d3.**entries**(*object*)
- returns an array of the property keys and values (each entry is an object with a key and value attribute)

```
d3.entries({foo: 42, bar: true});
>[{key: "foo", value: 42}, {key: "bar", value: true}]
```


Objects as hash tables?

- One could think of using bare JavaScript objects as hash tables
  - they are associative arrays, don’t they?
- However, this can lead to unexpected behaviors when built-in property names are used as keys
  - each object inherits from the prototype Object some properties and methods, which are already among its keys
  - inserting keys that are already defined by Object you overwrite them
- Further, objects keys are JavaScript names
  - they cannot start with a number

Objects as hash tables?

- Example of (erroneous) use of an object as a hash table

```
patrigna@tittolo:~$ node
> var obj = {};
undefined
> obj.hasOwnProperty('hasOwnProperty');
false
> obj.hasOwnProperty = "ciao";
'ciao'
> obj
{ hasOwnProperty: 'ciao' }
> obj.hasOwnProperty('hasOwnProperty');
TypeError: obj.hasOwnProperty is not a function
>
```

- Never use objects as hash tables!

D3.js maps

- To avoid problems D3.js defines its own hash tables, that are called maps

d3.**map**([*object*][, *key*])
map.**has**(key)
map.**get**(key)
map.**set**(key, value)
map.**remove**(key)
map.**keys**()
map.**values**()
map.**entries**()
map.**each**(function)
map.**empty**()
map.**size**()

```
var m = d3.map()

m.set("map-key", "map-value")
m.get("map-key") //map-value
```

```
m.set("hasOwnProperty", "ciao")
m.get("hasOwnProperty") // 'ciao'
```


D3.js sets

- D3.js implements sets
  - the elements of set are exclusively strings!

d3.**set**([*array*]) set.**has**(value) set.**add**(value) set.**remove**(value)	set.**values**() set.**each**(function) set.**empty**() set.**size**()

- D3.js sets are actually multisets
  - the method set.values() can be used to find the unique values of the set

```
d3.set(["foo","bar","foo","bar"]).values();
// produces the array [‘foo’, ‘bar’]
```


d3.set() function

- The values of the input array are always coerced to strings
  - D3.js converts objects and functions to strings using the toString() function
  - numbers already have the built-in JavaScript toString() function

```
d3.set(["foo","bar",32.4]).values()   //  [ "foo", "bar", "32.4" ]
```

  - objects don't have the toString() function
    - you will have an error unless you provide it

```
d3.set(["foo",{toString: () => "bar"}]).values();
                                          //  [ "foo", "bar" ]
```


D3.js nests (1/3)

- Nesting allows elements in an array to be grouped into a hierarchical tree structure
  - like the GROUP BY operator in SQL, except you can have multiple levels of grouping
  - the levels in the tree are specified by key functions

d3.**nest**()	• creates a new nest operator
.**key**(*key function*)	• the key() function will be invoked for each element in the array and must return a string identifier for each group
.**entries**(*array*)	• applies the nest operator to the specified array, returning an array of key-values entries

  - key() and entries() are methods of the object returned by d3.nest()

D3.js nests (2/3)

- Example data: barley yields, from various sites in Minnesota during 1931-2

```
var yields = [
  {yield: 39.93, variety: "Manchuria", year: 1931, site: "Crookston"},
  {yield: 32.00, variety: "Peatland", year: 1931, site: "Duluth"},
  {yield: 22.57, variety: "Manchuria", year: 1932, site: "Morris"},
  {yield: 25.87, variety: "Glabron", year: 1932, site: "Waseca"},
  {yield: 22.23, variety: "Svansota", year: 1932, site: "Morris"}
]
```

- 1-Level Nesting:

```
var entries = d3.nest()
         .key(function(d) { return d.year; })
         .entries(yields);
         // [{key: "1931", values: Array(2)}, {key: "1932", values: Array(3)}]
entries[0].values[0]
         // {yield: 39.93, variety: "Manchuria", year: 1931, site: "Crookston"}
```


D3.js nests (3/3)

- Example data: barley yields, from various sites in Minnesota during 1931-2

```
var yields = [
  {yield: 39.93, variety: "Manchuria", year: 1931, site: "Crookston"},
  {yield: 32.00, variety: "Peatland", year: 1931, site: "Duluth"},
  {yield: 22.57, variety: "Manchuria", year: 1932, site: "Morris"},
  {yield: 25.87, variety: "Glabron", year: 1932, site: "Waseca"},
  {yield: 22.23, variety: "Svansota", year: 1932, site: "Morris"}
]
```

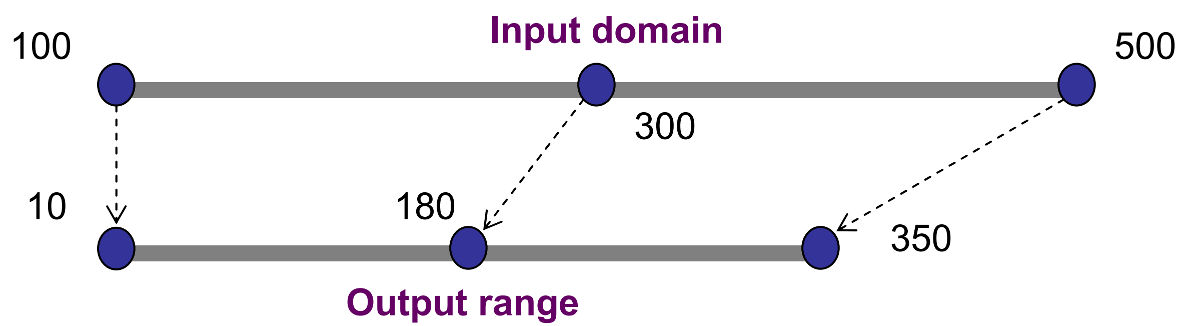
- 2-Level Nesting:

```
var entries = d3.nest()
         .key(function(d) { return d.year; })
         .key(function(d) { return d.site; })
         .entries(yields);
         // [{key: "1931", values: Array(2)}, {key: "1932", values: Array(2)}]
entries[1].values
         // [{key: "Morris", values: Array(2)},{key:"Waseca",values: Array(1)}]
```


D3.js scales (1/3)

- Scales are functions that map an input domain to an output range

```
var scale = d3.scaleLinear();
scale.domain([100, 500]); // Set the input domain
scale.range([10, 350])    // Set the output range
```




```
scale(100); // Returns 10
scale(300); // Returns 180
scale(500); // Returns 350
```


D3.js scales (2/3)

- Scales simplify the code needed to map a dimension of the data to a visual representation
- Scales are functions

```
var scale = d3.scaleLinear();
scale.domain([100, 500]); // Set the input domain
scale.range([10, 350])    // Set the output range
```

  - `typeof scale` returns “function”
  - domain() and range() are methods of functions

D3.js scales (3/3)

- Input domain
  - typically a dimension of the data that you want to visualize
    - e.g., height of students
- Output range
  - typically a dimension of the desired output visualization
    - e.g., the height of some bars

```
var heightToBar= d3.scaleLinear();
heightToBar.domain([0, 1.9]);
heightToBar.range([0, 100])
```


Most common types of scale

- Continuous scales
- Ordinal scales
- Time scales

Continuous scales

- Have a continuous domain (e.g., a set of real numbers or dates) and a continuous range
  - d3.scaleLinear()
    - by far the most common type of scale
  - d3.scaleSqrt()
  - d3.scalePow()
  - d3.scaleLog()
  - …

Ordinal scales

- Have a discrete domain (e.g., a set of names or categories) and discrete range
  - d3.scaleOrdinal()

```
var scale = d3.scaleOrdinal();
scale.domain(["A", "B", "C", "D", "E", "F"]);
scale.range([0, 1, 2, 3, 4, 5]);
scale("C")  // 2
```


Time scales

- Is an extension of d3.scaleLinear that uses JavaScript Date objects as the domain representation
  - domain values are coerced to dates

```
var scale = d3.scaleTime();
scale.domain([new Date("10/1/2016"), new Date("10/30/2016")]);
scale.range([0,100]);
```

- scale.ticks() returns representative dates from the scale's input
  - scale.ticks(n)
    - creates an array with n equally-distributed ticks
  - scale.ticks(d3.timeMinute, 15)
    - creates ticks at 15-minute intervals

Selectors

- Selections of elements is similar to jQuery
  - CSS selectors
    - see [learn.co]
  - simple selectors
    - identify elements by one facet

```
#foo            // <any id="foo">
foo             // <foo>
.foo            // <any class="foo">
[foo=bar]       // <any foo="bar">
foo bar         // <foo><bar></foo>
```

  - compound selectors
    - identify elements by two or more facets

```
foo.bar         // <foo class="bar">
foo#bar         // <foo id="bar">
```


Selection method

- D3 provides two top-level methods for selecting elements:
  - d3.select(`selector_str`)
  - d3.selectAll(`selector_str`)
- Selections are wrappers for DOM elements
  - provide utility functions to edit DOM elements (style, attr, remove, append, etc)

```
var titles = d3.selectAll("h1");
titles.style("color", "red");
```

Select and append

- Semantic of the append method
  - applied to a single element, adds the specified child

```
// select the <body> element
var body = d3.select("body");
// add an <h1> element with text "Last h1"
body.append("h1").text("Last h1");
```

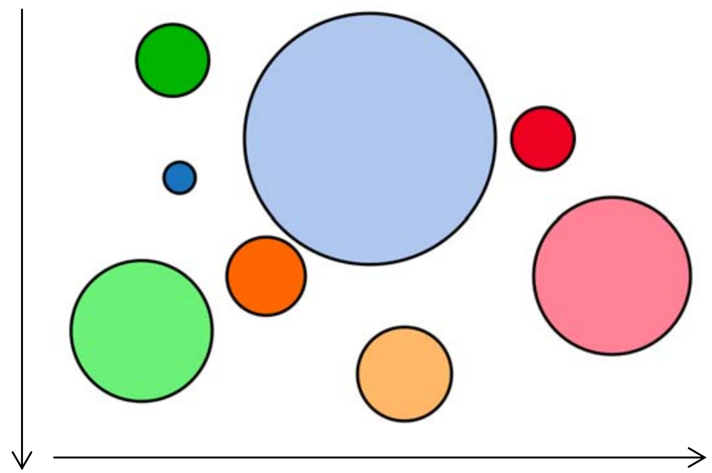
  - applied to an array of elements, adds one child to each

```
//Append two elements h1
body.append("h1").text("second h1");
body.append("h1").text("third h1");
//Select h1 elements
var hs = d3.selectAll("h1");
//Append h2 to each h1 element (always the same text)
hs.append("h2").text("heading 2").style("color", "green");
```


Data join

- Suppose we want to make a basic bubble chart using D3.js
  - we need to create an SVG circle element for each of your data point

```
var  data = [{"x": 50,  "y": 100, "z": 10},  {"x": 100, "y": 150, "z": 25}, …]
```



(x,y) is the **center**

z is the **radius**

Data join example code

- This code creates the bubble chart

```
<html>

// load d3.js

<body>
<svg width="500" height="400"></svg>
<script>
var  dataSet = [
      {"x": 50,  "y": 100, "z": 10},
      {"x": 100, "y": 150, "z": 25}, …]

var  svg = d3.select("svg");
…
```

```
…
svg.selectAll("circle")
                    .data(dataSet)              // this part will
                    .enter()                    // be clear soon
                    .append("circle")
                    .attr("cx", function(d) { return d.x})
                    .attr("cy", function(d) { return d.y})
                    .attr("r",   function(d) { return d.z})
                    .attr("fill", ..)
                    .attr("stroke-width", ..)
                    .attr("stroke", ..);
</script>
</body>
</html>
```

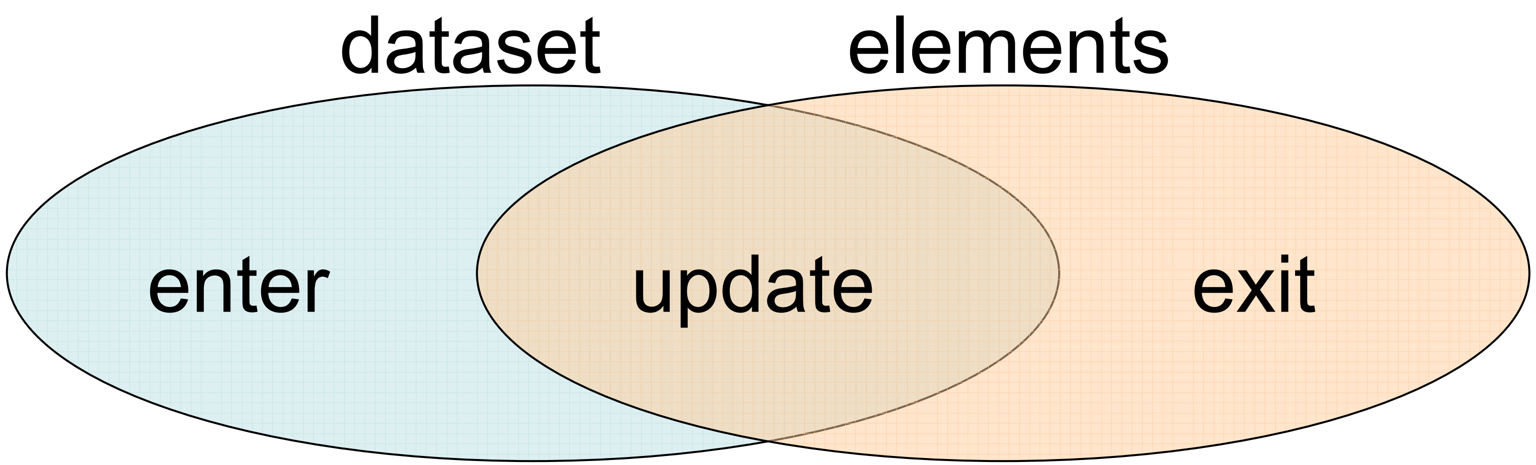
- Observe that we select elements that we know don’t exist in order to create new ones

The data driven paradigm

```
…
svg.selectAll("circle")
        .data(dataSet)
        .enter()
…
```

- Instead of instructing D3.js to create circles, we tell D3.js that the elements of the selection "circle" should always correspond (i.e., are “joined”) to the data contained in the array dataSet
  - see [Bostock 12]

Three kinds of elements



- Elements already present in the DOM that are already joined to some data
  - they are selected by the update() function
- Elements that are not yet in the DOM and that are joined to new data
  - they are selected by the enter() function
- Elements of the DOM that do not have a joined datum anymore
  - they are selected by the exit() function

The data() default output

```
...
var p = d3.select("body")
          .selectAll("p")
          .data([4, 8, 15, 16, 23, 42])
          .text(function(d) { return d; });
...
```

- Elements to be updated are the default selection
  - the actual result of the data() operator
  - this allows you to automatically select only the elements for which there exists corresponding data

A first common pattern

```
var p = d3.select("body")                    // Update…
          .selectAll("p")
          .data([4, 8, 15, 16, 23, 42])
          .text(function(d) { return d; });

 p.enter().append("p")              // Enter…
   .text(function(d) { return d; });

 p.exit().remove();                          // Exit…
```

- Break elements into three sets
  - elements to be updated
  - elements to be inserted
  - elements to be removed

A second common pattern

```
var p = d3.select("body")                    // Enter…
        .selectAll("p")
        .data([4, 8, 15, 16, 23, 42])
        .enter().append("p")
        .text(function(d) { return d; });

 p.doSomething();                            // Enter + Update

 p.exit().remove();                          // Exit…
```

- The sequence of enter() and append() functions add the new data to the update clause
- The doSomething() function applies to the default update data (now enter + update)

The data() function

```
selection.data([values[, key]])
```

- The data() function joins the specified array of data with the current selection
- values
  - is an array of number or objects
- key()
  - is an optional function that controls how data is joined to elements
  - if it is not specified, then the i-th datum in values is assigned to the i-th element in the current selection
  - otherwise, it returns a string which is used to join a datum with its corresponding element
    - the one that has the same key

Loading data from CSV

```
d3.csv("stocks.csv“)
        .then(function(data) {
                // executed right after loading
        })
        .catch(function(error) {
                // executed if errors occurr
        });
```

- Loads data from a file in CSV (Comma-Separated Values) format
- Inside the .then callback the “data” parameter corresponds to the array of objects loaded from the CSV file

Loading data from JSON

```
d3.json("stocks.json“)
        .then(function(data) {
                // executed right after loading
        })
        .catch(function(error) {
                // executed if errors occurr
        });
```

- Loads data from a file in JSON (JavaScript Object Notation) format
- Inside the .then callback the “data” parameter corresponds to the object loaded from the JSON file

Transitions

- A transition is a special type of selection where the operators apply smoothly over time rather than instantaneously

```
d3.select("body").transition().duration(5000).attr("bgcolor","yellow");
```

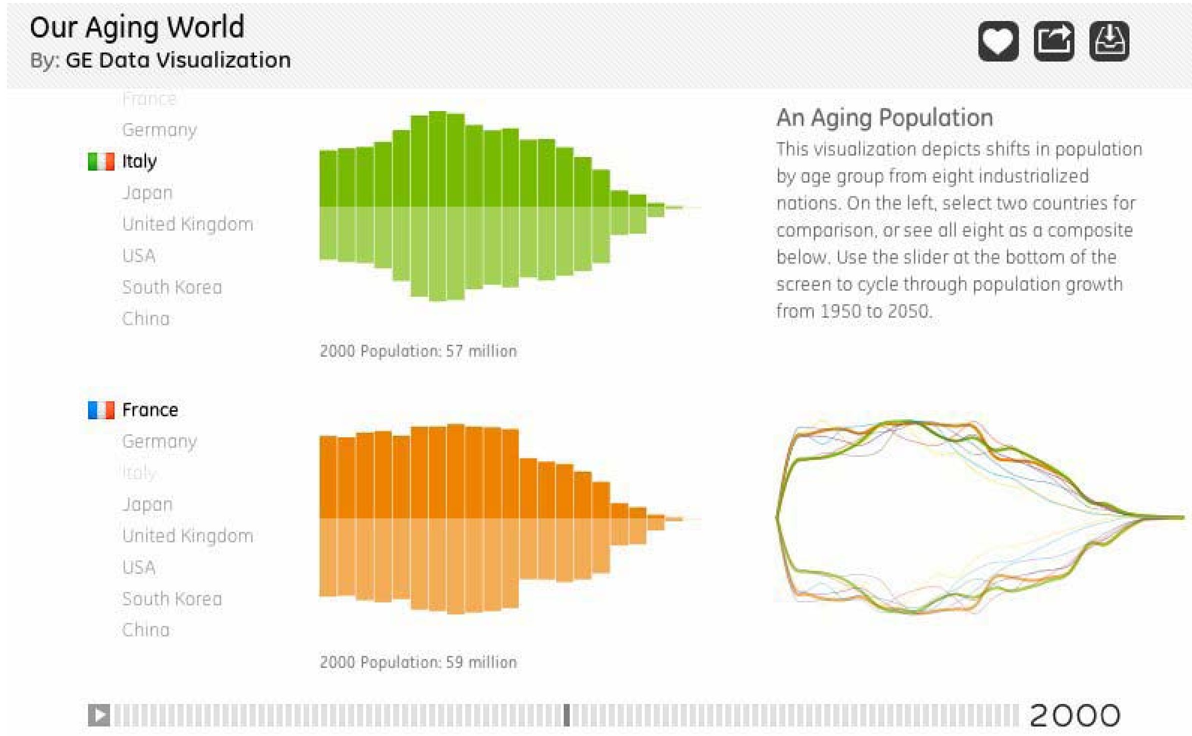
  - selection.transition(): derive a transition from an existing selection
  - transition.duration([duration]): specifies duration in milliseconds
  - transition.attr(attribute, value): transforms the specified attribute into the specified value

Interpolations in transitions

- Transitions interpolate values over time
- D3.js determines an appropriate interpolator by inferring a type for each pair of starting and ending values
  - colors
  - geometric transforms
  - strings with embedded numbers (e.g., "96px")
    - string interpolators have several applications:
      - interpolating font sizes, stroke-width, etc
      - interpolating path data (e.g., "M 0, 0 L 20, 30")

Example: Italy aging

- "Our aging world" depicts shifts in population
  - by age group
  - for different countries
  - for both genders



https://fathom.info/aging/

- We will visualize shifts in population
  - by age group
  - for a single country
  - aggregating genders

http://www.dia.uniroma3.it/~infovis/demos/italy_aging_d3.zip

Our data format

```
[
  {
    "year": "1950",
    "ageGroups": [
     { "ageGroup": "0-4", "population": 4369 },
     { "ageGroup": "5-9", "population": 3839 },
     { "ageGroup": "10-14", "population": 4170 },
     {...}
    ]
  },
  {
    "year": "1960",
    "ageGroups": [{…}, {…}]
   }
]
```

The code

- Use data to create multiple elements

```
[
  {
    "year": "1950",
    "ageGroups": [
     { "ageGroup": "0-4", "population": 4369 },
     { "ageGroup": "5-9", "population": 3839 },
     { "ageGroup": "10-14", "population": 4170 },
     {...}
    ]
  },
  {
    "year": "1960",
    "ageGroups": [{…}, {…}]
  }
]
```

```
// dataset is initialized with
// our data (for instance from 1950)

var values = dataset[0]["ageGroups"];
var bar = d3.selectAll(".bar").data(values, function(d){
    return d.ageGroup;});

    bar.enter()
    .append("rect")
    .attr("class", "bar")
    .attr("x", function(d) { return x(d.ageGroup);})
    .attr("y", function(d) { return y(d.population);})
    ...
```

Think with joins!

```
function updateDrawing(data){// data = dataset[i] for some i
  var values = data["ageGroups"];
  // Data join
  var bar = svg.selectAll(".bar").data(values, function(d){
      return d.ageGroup});

  // Exit clause: Remove elements
  bar.exit().remove();

  // Enter clause: Add elements
  bar.enter().append("rect").attr("class", "bar")
  .attr("width", x.bandwidth())
  …

  // Enter + Update clause: Update y and height
  bar.transition().duration(updateTime)
  .attr("y", function(d) { return y(d.population); })
  .attr("x", function(d) { return x(d.ageGroup); })
  .attr("width", x.bandwidth())
  .attr("height", function(d) { return height - y(d.population); });
... }
```


More examples

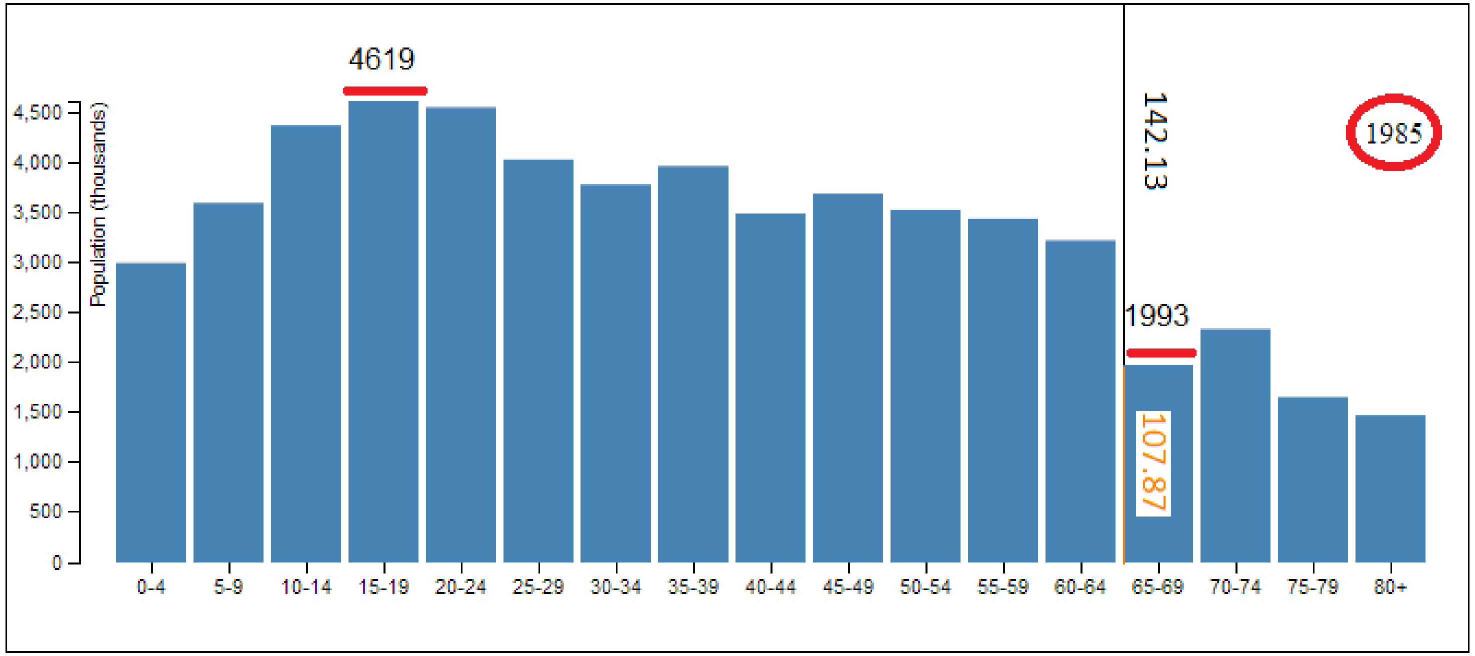
- Suppose you have the following current datasets

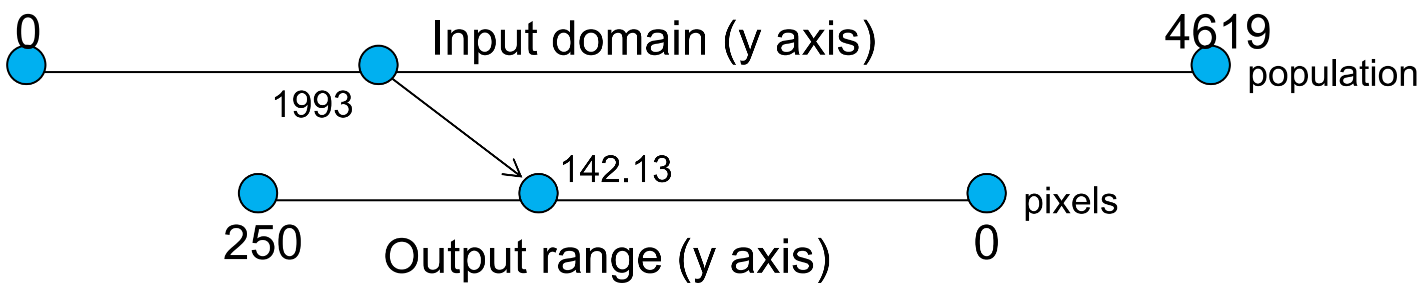
```
var input = { "year": "1955",
  "ageGroups":
  [
    { "ageGroup": "0-4",
     "population": 4034 },
    { "ageGroup": "5-9",
      "population": 4286}
  ]
}
```

```
var input = { "year": "1960",
  "ageGroups":
  [
    { "ageGroup": "0-4",
      "population": 2503 },
    { "ageGroup": "5-9",
      "population": 1300 },
    { "ageGroup": "30-34",
      "population": 2700}
  ]
}
```

- try to update the drawing using updateDrawing(input) in the console

Use of the y linear scale





Height = 250 – y(1993) = 250 – 142.13 = 107.87 pixels

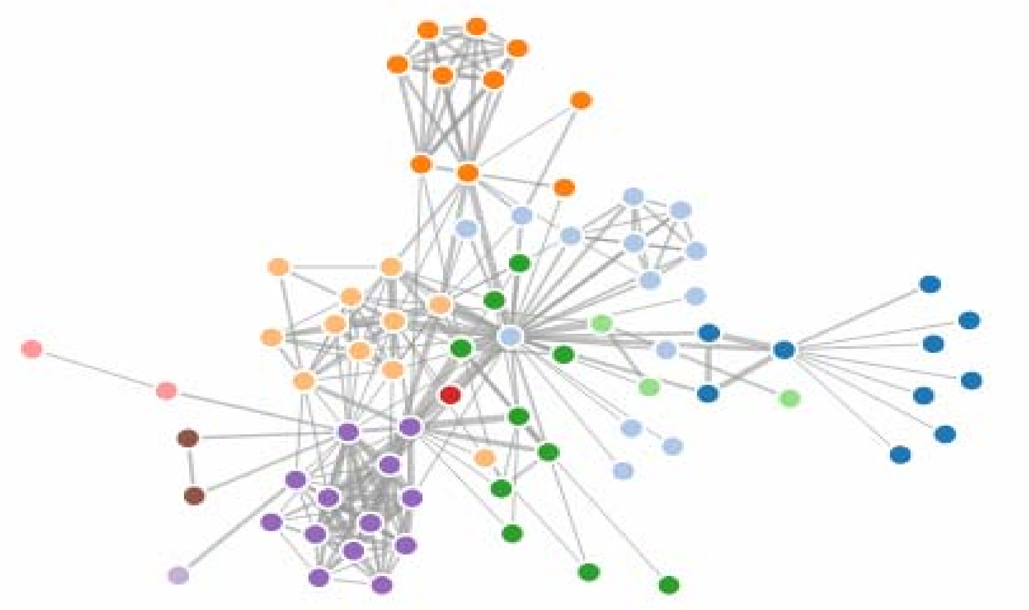
Domains and ranges

- Update periodically the domain of our x and y scales using the current values
  - update accordingly axes and bars’ heights and widths

```
var y = d3.scaleLinear().range([height, 0]);
...
function updateYScaleDomain(data){
  var values = data["ageGroups"];
  y.domain([0, d3.max( values, function(d) {
    return d.population;})
  ]);
}
```


Layouts: Force Directed

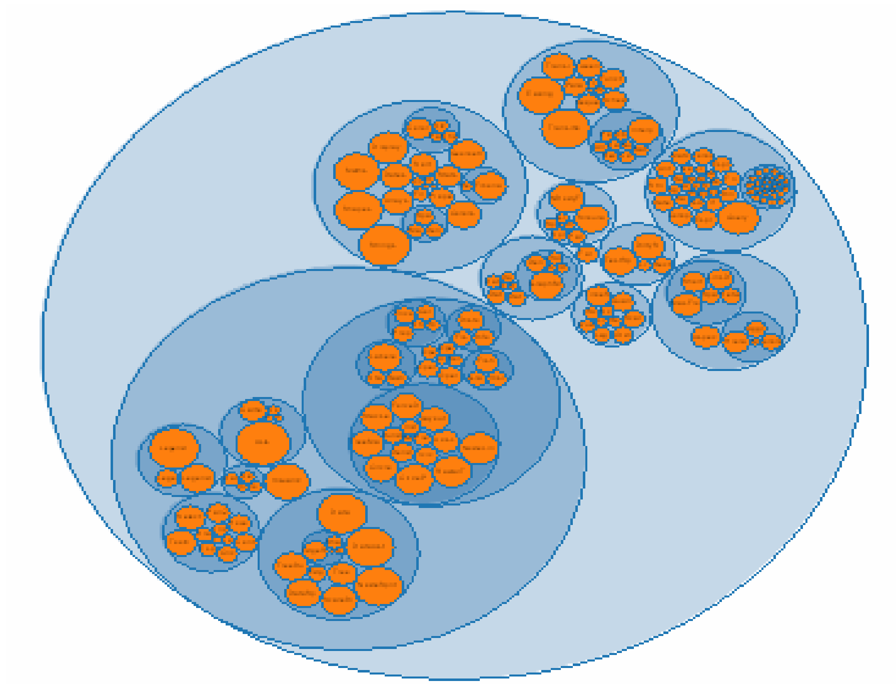
- This simple force-directed graph shows character co-occurence in *Les Misérables*



- observablehq.com/@d3/force-directed-graph

Layouts: circle packing

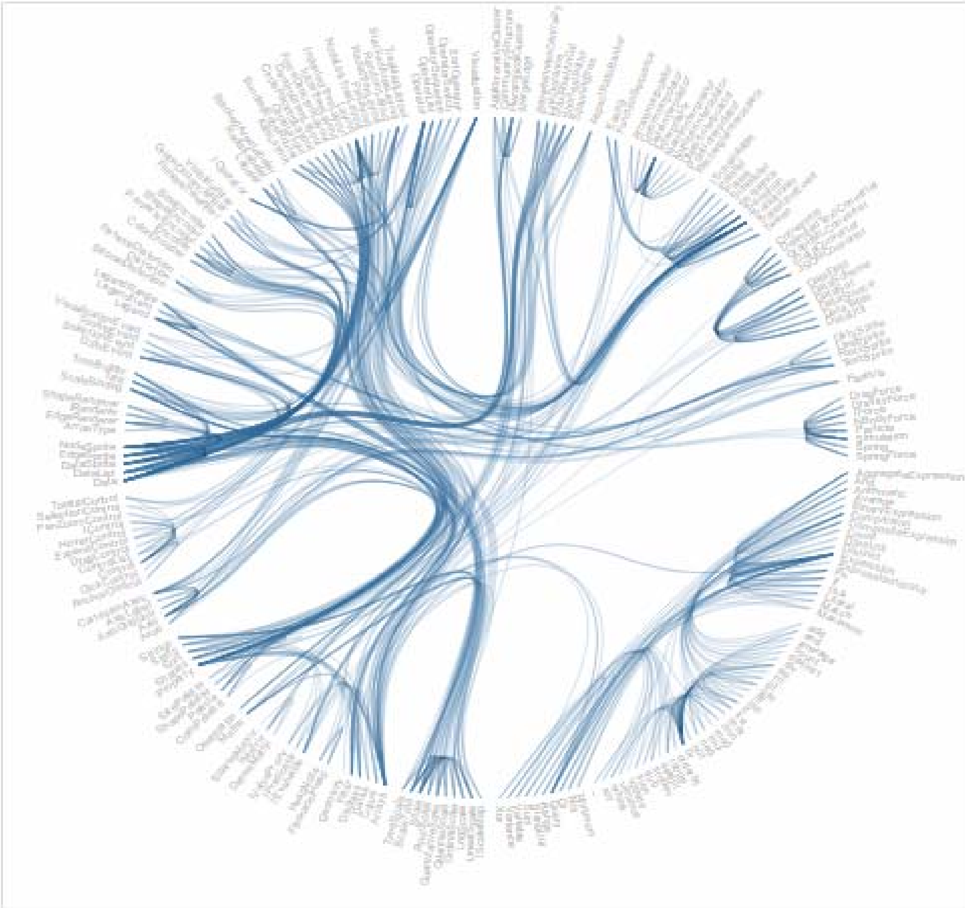
- Enclosure diagrams use containment to represent the hierarchy.



- observablehq.com/@d3/circle-packing

Layouts: Bundle

- Danny Holten's heirarchical edge bundling



- Dependencies between classes in a software class hierarchy
- Dependencies are bundled according to the parent packages

- observablehq.com/@d3/hierarchical-edge-bundling

Layouts: Radial Trees

- The tree layout implements the Reingold-Tilford algorithm

  - observablehq.com/@d3/radial-tidy-tree

References

- https://github.com/mbostock/d3/wiki/API-Reference
- https://github.com/mbostock/d3/wiki
- https://groups.google.com/forum/#!forum/d3-js
- http://stackoverflow.com/questions/tagged/d3.js
- [learn.co] https://learn.co/lessons/css-selectors
- [Bostock 12] Mike Bostock, “D3 Workshop”, 2012 https://bost.ocks.org/mike/d3/workshop/#0
- [protovis] Mike Bostock, “Protovis: a graphical approach to visualization”, https://mbostock.github.io/protovis/
- [W3C] “W3C Recommendation”, http://www.w3.org/TR/SVG/intro.html
- [D3 API] https://github.com/mbostock/d3/wiki/API-Reference

