Front End Visualization Notes:

Sean al-Baroudi

July 26, 2019

D3.js[1]

- Steps to Any Visualization:
  1. **Acquire:** Just get the data.
  2. **Parse:** Extract the raw data; give it meaning and organize it into sets and catagories.
  3. **Filter:** Apply more constraints on your parsing, and extract the data of interest. You can have multiple filters on top of a parsed data set (one for each Data Visualization).
  4. **Mine:** Apply statistical, data mining or transform methods.
  5. **Represent:** Choose your visual model. Make sure the data has been transformed to be fed into this representation.
  6. **Refine:** Apply style, aestetics and make the visual rep stand out and be appealing to the reader
  7. **Interact:** Allow users to interact with the data visualization; make it dynamic.
- **High Level Method Calls:**
  1. *.select():* Selects a Dom object from the page.
     Selections in D3 are Arrays
     Every sub-chained call returned is just another selection, with the exception of .data (in which selections can be virtual).
  2. *.attr():* Adds an attribute to a selected DOM object; takes key value pairs.
  3. *.style():* Adds CSS style attributres to a selected Object.
  4. *.text()* Sets the text for closed tags, such as P and DIV.
  5. *.selectall()* Self Explanitory; will select what it can for bindings, even if data is greater than element size.
  6. *.data()* Joins an array of data with the current selection; this means an association (sequential mapping) is done between the element sets. If one set is smaller than the other, .data() just maps the first K elements.
     .data() is the only method that returns virtual selections; hence .enter() .exit() etc. must be called on the return value of this operator.
  7.
- Call Rational and Rules:
  1. **Every .Operator() call returns a selector object.**. Many calls (such as .style()) will just return what they were called on.
  2. Evaluated Eagarly, L to R.
- In D3.js, you can chain syntax or assign sub-chains to variables, and just call on the variable names.
-
- **New/Unique Concepts:**
  1. **D3 Data Join:[2]** This allows us to call .data() mapping methods on selected set of data, even if the two sets have a different number of elements. If a difference occurs, **Virtual Selections** are returned instead of **Literal Selections**. There are three types:
     - **Enter:** Data points that are unmapped (data > selected elements), they are put in this set, primed to be created at a later date
     - **Update:** Points that were mapped are placed here.
     - **Exit:** Points are put here if (data < selected elements).
  2. With the data join, we can dynamically create visualizations, and not worry about our data set updating or dynamically changing; we can code in such a way that we generate data points on the fly.
  3. **Anonymous functions:** Can be declared in a complex call, inserted inline. Syntax is:

     $$function\ (X...)\ [d, i, this]...return$$

     There are three predefined variables we can use:d refers to a bound data array element, i refers to the index of the selected DOM element and *this* refers to our selector set our call is being made on.
-
-
-

Graphics in D3.js

- Coordinate Space: "Computer Monitor": take your Cartesian Plane and flip it 180 degrees around the X axis. Attach the origin to the Top Left of your screen. This is how D3.js orients itself.
-
-

Scales in D3.js:

- Data Transforms in D3 are functional: domain $\longrightarrow$ range.
- In D3, and *Interval* is just a *Domain*.
- **Linear Scaling:** Sets up a simple slope equation between x and y:

  $$f(x) = \frac{1}{q}(x) = y$$

  where:

  $$q = \left(\frac{\Delta d}{\Delta r}\right) = \left(\frac{dmax - dmin}{rmax - rmin}\right)$$

  An Example of some code is below:

  ```
  var aScale = d3.scaleLinear().domain([1,10]).range([1,100]);
  ```

- **Ordinal Scaling:** Makes a 1 to 1 mapping between domain and range; which can be any data type on either side $\{1 \rightarrow A, 2 \rightarrow B, ...\}$.
  For Example:

  ```
  var someData = ['a','b','c','d','e','f','g']
  var oScale = d3.scaleOrdinal().domain(someData).range([1,3])
  ```

- If $|dom| \leq |range|$, you use modulo arithmetic to map the domain onto the range.
- Clamping: Functions in D3 will extrapolate by default (if you give outside of domain values). You can restrict to just the mapped domain by using the .clamp(true) function on your defined function. **Giving an outside value will just return the max or min bounds of the range.**
- 'Nice-ing': With real data, .nice() rounds domain and range endpoints to clean and simple values. Must be called everytime your function is updated with new data.
- Inverse: The .inversion() function maps range $\rightarrow$ domain, as one might expect.
- Scale Bands: Are used for barcharts and histograms. Data is mapped into box regions and displayed. More specifically, this determines the geometry and spacing of the bars. For a given domain and range, scaleBand determines the bandwith() of each band automatically. An example of usage:

  ```
  var bandScale = d3.scaleBand().domain(['Mon', 'Tue', 'Wed', 'Thu'
  ```

-
-
-
-
-
-
-
-
-

External Data References recognized by D3:

- XMLHttpRequest
- raw textfile

- JavaScript Object Notation (JSON) Blobs
  Json objects are stored in an array, and attributes are accessed using key:value pairs such as:

$$jsonArray[2].color$$

- HTML docs and fragments
- XML docs and fragments
- CSV Files
- TSV Fiels
- **Note:** When these are loaded, complex data objects are imported into object/nested array like structures.

Additions:

- **JavaScript Memory Model Fundamentals:**
  - Objects are Associative Arrays.
  - Functional Passing: Primitives: These are passed by value.
  - Functional Passing: Objects: These are passed by reference. The recieving function can modify and add new properties; it cannot completely reassign the object reference.
  -
  -
- Selections in D3 are made with the select() and selectAll() statement.
- Such statements return selections, *which are a modified subclasses of the Array data structure.*
- this subclassing overrides using the prototype property. Other global JS methods (like filter(), sort()) are also overridden to implement D3 specific behaviours.
- All selections are structured like the following:

$$(Array \rightarrow Group(Field) \rightarrow Results)$$

- Details of **select() Method:**
  1. One Group → One Matched Element (at most).
  2. Parent Node is propogated in each chained call.
  3. Grouping is always preserved (precisely one only).
- Details of **selectAll() Method:**
  1. **First Call:** One Group → [Array of results]. Parent tag set to [Document] Object.
  2. Next Call: One group for every result in array - search is focused with each previous result as a search root.
  3. Parent tags updated to previous group elements.
  4. **Data does not propogate between chained commands, must be rebound everytime.**
  5. **SelectAll does not preserve groupings.**
  6.
  7.
  8.
- Data Binding Implementation: Each DOM element has a __ data __ field in its object representation. Datum that is attached to a DOM element is linked up here.
- Ways to Bind Data:
  1. Datum(): Binds data to a selection; no joining is done. All this does is set the entire data array to every elements __ data __ field, ignoring data structure.
  2. Data(): Will compute a join between DOM elements and Data elements. If the group structure and data structure match, subarrays of data will be matched to each element. If not, Enter, Exit and Group fields will be generated for the selection.
  3. Inheritence using chains of selects().
- When data cannot be joined or bound, Null Elements are returned to indicate this.
- Key Functions: When you are given structured data, and you are trying to match it with grouped structured elements, key functions can be used to match the data accordingly.
- By default, data and elements will be auto-indexed (in the order they appear in the NodeLists) and matched up. If the elements need to be rearranged, use a key function.
- IN order to do key accessing, data must already be in the DOM elements. This seems to work better on transitions and updatings, but not setups ***.
- When multiple groupings are involved, **keys must be unique at the group level.**
-
-
- DATA does the join, which defines group, enter and update fields. ENTER(), EXIT() are selectors that allow you to select these array fields after a join has been done; you can clean up or add new elements (with virtual selectors).
-
-
- Note: You can bind data to elements (like circles, lines, etc), however if you don't use access functions to set their attributes, they will not show up on screen.
-
- To make a dynamic chart you need to use enter() and exit(). If you just want to generate a static chart, use datum() or data()
-
-
-

General Update Pattern:

1. $\{SelectionElements\}.data(theData, keySelector)$
   This returns: A reference to an object with __enter__, __exit__ and __group__ array fields.
   Data joins are attempted, and depending on missing data/elements, datum and elements are binned in respective groups. Successful joins are put in the group array.
2. **selection.update():**
   Returns the group array. Here, elements and data are bound. Set properties here.
3. **selection.enter()**: the Enter array is returned, and this contains datum with no bounded elements. Append and generate elements here, for every element in the enter array.
4. selection.exit(): These are elements with no bound data. Typically, we remove() such elements on an update.
5. **General Idea:** New data flows in and is updated/bound to elements. Old elements lose their data, and are quickly removed. Anything without data is removed in the next phase.

Scales:

- A scale is a function that maps Abstract Data → Visual Data
- Abstract Data Types: Ordinal, Numerical
- Visual Data: Bars, lines, charts, etc.
- Note: Scales naturally interpolate for continuous data!
- Scales have a domain and range, these must both be specified for the mapping to occur.
- scaleLinear() Generates a [0,1] → [0,1] mapping all on its own.
- Typical Usage:

$$varscale = scale().domain().range().otherproperties()...$$

-
-
-
-
-
-

Axes:

- Axes are human visualizations of d3-scales.
- In Terms of SVGs, an axis is a complex group ($< g >$) of SVG lines and text, that are prearranged by the d3-axis methods.
- Because of the HTML5 complexity of making an axis object, it is best just to use this library.
- General Usage: Use axisTop/Bott/etc to place a default axis. Transform it to the right location in the main SVG box. And then apply ticks, scales and styles for the formatting phase.
-
-
-
-
-
-
-

Basics of SVG Elements:

- SVG Data Structure: Tags/Commands → DOM Model → Generated Image. Manipulating DOM Model is what allows all this to be dynamic.

- Our D3 visualizations are enclosed in SVG elements.
- Note: Conflation: SVG is not a Canvas; a canvas is a rasterized, static image that is apart of the HTML5 standard; these can't be used to make dynamic visualizaitons.
- To do meaningful things in D3, we need the group tag $< g >$.
- This acts as a container to hold other SVG objects (text, circles, etc).
- Group elements have no x,y attributes. You can apply SVG transformations to groups.
- Groups pass on attributes and styles to their children. For styles (at least), children can of course override them.
- Useful SVG elements: circles, lines, Paths.
- Note: All of our complex line graphs will be SVG PATH elements.
-

Residual Questions:

How does modulo mapping work for the .invert() function?

References

[1] `https://www.dashingd3js.com`

[2] `https://bost.ocks.org/mike/join/`