Reusable D3 components

Author: Michael Oppermann and Tamara Munzner. Last change date: 11 Oct 2019.

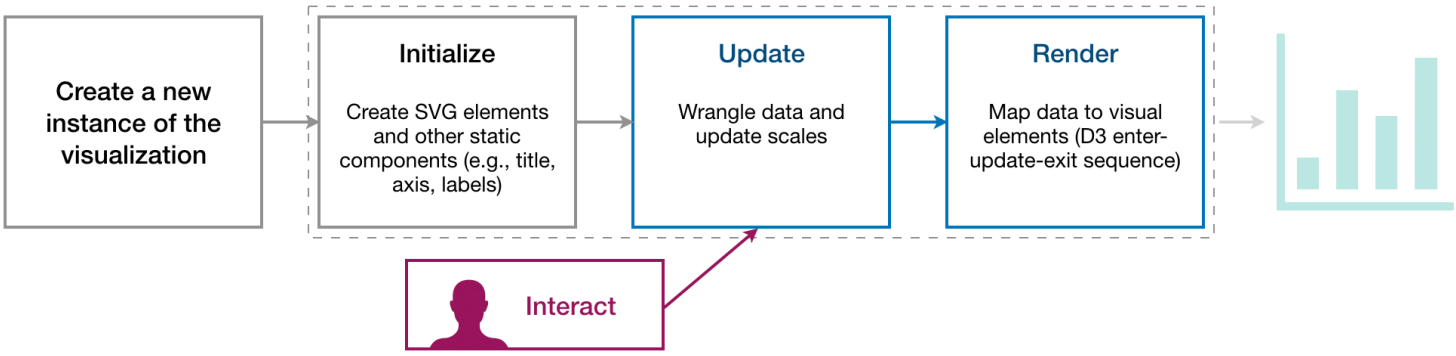
Thinking about the structure of your project early on can save you a lot of time and will make your implementation more robust, extensible and reusable.

Divide and conquer

You should always try to split a complex problem into smaller, easier-to-tackle sub-problems. Each sub-problem can then be solved independently and afterwards integrated into the final system.

D3 visualizations should be organized and structured using individual classes for different chart types: do not have one monolithic file containing all of your code. Also, think about **consistent structure for the functions within each class**, to make these components as flexible and reusable as possible.

We recommend creating a JavaScript class for each visualization chart type that is used following this pipeline:



Function Structure of an Individual Class

ES6 introduced classes to better support object-oriented programming that you should use. Note that when reading code for older JavaScript versions, a similar functionality was achieved by using Object.prototype functions.

Similar to other programming languages ES6 classes have a constructor and properties can be defined by using the keyword `this`:

```
// ES6 Class
class BarChart {

  constructor(_config) {
    this.config = {
      parentElement: _config.parentElement,
      height: _config.height || 300,
      margin: { top: 10, bottom: 30, right: 10, left: 30 }
    }

    // Call a class function
    this.initVis();
  }

  initVis() {
    ...
  }

  ...
}
```

Instantiating a new instance of a class is done by using the keyword `new`. The constructor should be used to define properties:

```
// Create an instance
let barchart = new BarChart({
    "parentElement": "bar-chart-container",
    "height": 400
});
```

Object attributes can be updated afterwards:

```
barchart.data = data;
```

The variables should be stored in the chart object. We recommend avoiding simply using the `this` keyword within complex class code involving SVG elements because the scope of `this` will change and it will cause undesirable side-effects. Instead, we recommend creating another variable (for example `vis` or `_this`) at the start of each function to store the *this*-accessor.

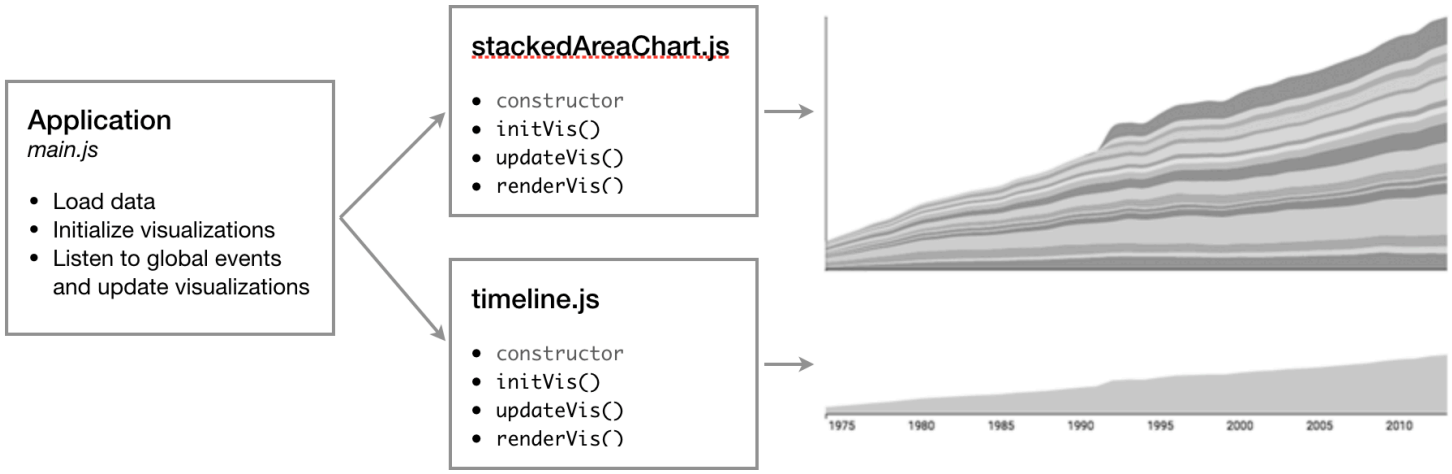
```
initVis() {
    let vis = this;

    vis.svg = d3.select(vis.config.parentElement).append("svg");
    vis.chart = vis.svg.append('g')
        .attr('transform', `translate(${vis.config.margin.left},${vis.config.margin.top
    ...
}
```

The functions `initVis()`, `updateVis()`, and `renderVis()` should be used, following the implementation pipeline above. You might also need additional functions. The goal is to execute only the code that is needed to update the chart instead of removing and redrawing the entire chart after a user interaction; this code structure makes that goal straightforward to achieve.

```
initVis() {
    let vis = this;
    ...
}
updateVis() {
    let vis = this;
    ...
}
renderVis() {
    let vis = this;
    ...
}
```

Breakdown of Project Into Classes

In this example, there are two chart types, stacked area chart and timeline, and so there are two class files, `stackedAreaChart.js` and `timeline.js`



The divide-and-conquer concept (i.e., splitting up a complex problem into various sub-tasks) also applies to the overall file structure of your project.

We recommend that you create object instances for the chart type classes in the file `main.js`, which should be the entry point for your application, so that your code stays clean and understandable. For example, if you want to use the same data for multiple charts, you would load the data only once in *main.js*, and then re-use it in each class instance.

This methodology will become very helpful for developing larger systems and more sophisticated interaction mechanisms.