Prerequisites

Last time you learned about how to use SVGs within HTML documents to produce visual outputs, how to select and manipulate elements via JavaScript and D3, and how to load and prepare data. You will need this knowledge to participate in this tutorial.

Feel free to look up what we did in the last tutorial at any time:

http://vda.univie.ac.at/Teaching/Vis/16s/LectureNotes/D3_Tutorial_1.pdf

Scales

Last time we worked with static data to create diagrams within which the position and size of visual elements directly corresponded to the data. If we are working with dynamic data, we do not always know the range of values we will receive as input. Hence we cannot encode the position and size of our values in absolute pixel values as we might receive data that exceeds the size of our SVG.

D3 provides scales to solve this problem. They can be used to dynamically scale the visual representation of our data so it always fits into a certain area of our webpage – independent of the value range of our data.

You can think of scales as a function that scales the input domain to an output range.

If we know the **input domain** and **output range** of our scaling, we can create a scaling function like this:

```
var myScale = d3.scale.linear()
        .domain([0, 100])
        .range([0, 10]);
myScale(50); //Maps 50 -> 5 and returns 5
```

You can find more about scales here: https://github.com/mbostock/d3/wiki/Scales

To find out the min and max values of already loaded data, we can use the *min(array)*, *max(array)*, and *extent(array)* functions. min() and max() return the minimum and maximum value of the given array respectively, and extent() returns an array containing the min and max.

We can call these functions like this:

```
var data = [1, 2, 3, 4, 5];
var min = d3.min(data);
```

If your data consists of more complex objects, you can pass any of these methods an anonymous accessor function as second parameter.

```
var complexData = [
        {name: "Alice", height: "174"},
        {name: "Bob", height:"165"}
];
```

```
var min = d3.min(compexData,
        function(d){
                return d.height; //We only evaluate the height.
        });
```

And more about min(), max(), and extent() here: https://github.com/mbostock/d3/wiki/Arrays#d3_min

Other Scales

Besides the linear transformation we achieved with the *scale.linear()* call, D3 also offers further methods to define scales. E.g.:

scale.pow().exponent(myExponent)	Provides a power scale with the given exponent.
scale.log()	Provides a logarithmic scale with base 10.
scale.oridnal()	Provides an ordinal scale for which you can provide arrays in the .domain() and .range() method.

There are some additional methods helping you to deal with ordinal scales. One of them you might need is the *ordinal.rangeBands(interval)* method you can use instead *ordinal.range(interval)*. This method splits the range you pass in into as many sub-intervals of equal length as you have values in your input domain and maps to those interval.

More about ordinal scales here: https://github.com/mbostock/d3/wiki/Ordinal-Scales

They also allow for color scales: https://github.com/mbostock/d3/wiki/Ordinal-Scales#category10

And scales specifically for time data: https://github.com/mbostock/d3/wiki/Time-Scales

Exercise

- Download the data http://vda.univie.ac.at/Teaching/Vis/16s/data/fruit.csv
- Load the data.
- Convert the attributes to numbers as appropriate.
- Create an SVG element and store it in a variable.
- Create a scale from the *input domain* of **price** to the *output range* defined by the **height** of the SVG. Use the *min()*, *max()*, or *extent()* methods as appropriate.
- Create a simple barchart with the bar length corresponding to the scaled **price**, each bar labelled by the **fruit** name and the bar color encoding the pit-type.
- Make sure that everything is visible. You can refine your plot by changing the scaling.

Groups & Transformations

In the last assignment you probably plotted the bars and the labels on the same level in either the x or the y dimension. Those elements belong together spatially and if we manipulate their position (imagine sorting the bars by size), we would like to do that in one transformation for the bar and label at once.

SVGs provide the group element via the tag <g></g> to allow for group-wise manipulation of elements. Transformations can be applied to the group element (as to basic shapes) and affect all its children.

```
var group = mySVG.append("g"); //Adds the group
group.append("circle") //Appends a circle to the group
        .attr("r",10);
group.append("rect") //Appends a rectangle to the group
        .attr("width",5)
        .attr("height",5);
group.attr("transform","translate(70, 50)"); //Translates the group, hence
both the rectangle and circle
```

You can find a list of transformations and examples here:
https://developer.mozilla.org/en/docs/Web/SVG/Attribute/transform

Axes

Axes are an essential tool for creators of data visualizations. This is why D3 provides us with the axis component.

```
var axis = d3.svg.axis();
```

For our axis to map the correct value of the input domain to the correct output range, we can pass the same scale we used for our marks, e.g. bars.

```
axis.scale(myScale);
```

D3 provides some additional methods to manipulate your axes, eg.:

> *axis.orient(orientation)*: Takes a string specifying on which side of the line the tick-marks should appear ("top", "bottom", "left", "right").
>
> *axis.ticks(arguments…)*: Takes arguments that are used for the generation of ticks. In the simplest case, the argument would be the number of ticks to use.
>
> *axis.tickValues(values)*: Takes an array of values to be used as tick values.

Axes can be attached either to SVGs or to groups via the *call(axis)* method.

```
d3.select("g").call(axis);
```

To place the axis where you want it to be, you can use transformations as before.

More about axes in general here: https://github.com/mbostock/d3/wiki/SVG-Axes

Exercise

- Reuse your previous code. In case you don't have it anymore or didn't finish, get the example code here: http://vda.univie.ac.at/Teaching/Vis/16s/data/tutorialCode/tutorial2_1.html
- Add an axis to your plot that correctly measures the bar height in terms of the input domain.
- Make sure the axis is displayed on the left or bottom (depending on your bar rotation) and the tics are completely visible. **Hint**: Make sure the axis *orientation* is correct and maybe add some padding if the ticks are not completely visible.

Interactivity

Until now our visualizations do not allow any user interaction. Since interaction is an integral building block of most recent visualizations we will now take a glance at simple ways to integrate this in our page.

HTML GUI elements

Probably the simplest form of adding interactivity to your visualization is providing HTML GUI elements that call JavaScript functions, which, in turn, manipulate the visual elements.

For example, we can create a button that calls the function onButtonClick().

```
<div id="buttonDiv">
        <button type="button" onclick="onButtonClick()">
                Click me!
        </button>
</div>
<script>
function onButtonClick () {
        //Change the vis
}
</script>
```

Within the method provided we could for example remove all our visual elements and redraw them according to new data.

Enter – Update – Exit

Deleting and recreating visual elements on any change is expensive. That is one reason why D3 provides the Enter – Update – Exit pattern. As you have already heard in the last lecture, D3 provides two methods, *enter()*, *exit()* that control the interaction between data and HTML elements it is bound to.

Let's recap:

mySelection.data(mydata).**enter()**	handles **data items for which no elements exist** yet in the selection. Often we want to append missing elements here.
mySelection.data(mydata).**exit()**	handles **elements for which no data item exists** in our selection anymore. Often we want to remove the superfluous elements here.
mySelection.data(mydata)	only binding data and elements returns the 'update' selection and handles **existing elements for which the data changed**. Often here most things are done, e.g. setting the element's properties.

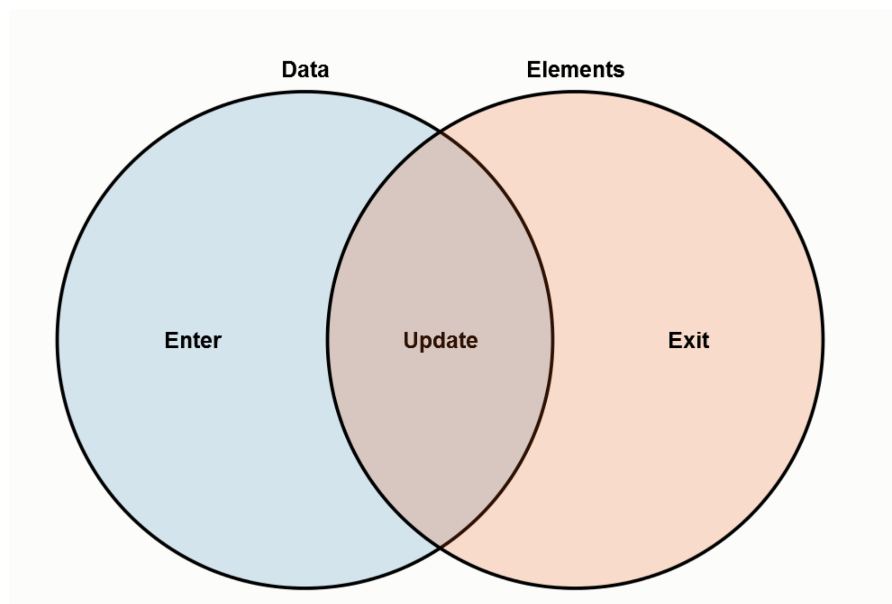

Figure 1 https://bost.ocks.org/mike/join/

Imagine we want to draw circles based on data that is regularly updated (think of a scatterplot of time-dependent data). We could use the enter – update – exit scheme like this:

```
function updateDrawing(newData){
        var circles = mySvg.selectAll("circle")
                .data(newData); //Join circels and data
        circles.enter().append("circle "); //Append missing circles
        circles.attr("x", function(d){ return d.x; })
                .attr("y", function(d){ return d.y; })
        circles.exit().remove(); // Remove superfluous circles.
}
```

Joining Data and HTML Elements

Usually when we join data and HTML elements with the *.data()* function, they are linked by index, meaning that the nth element is joined with the nth data-item.

Often we don't want elements to be joined by their index, but *we want the same element to be joined to the same data item*, no matter what happened to the data. This has efficiency advantages and also helps create beautiful animations.

To achieve that we can give the *.data()* function an additional anonymous function, called the *key function*, as argument, specifying by what criterion the HTML elements and data item should be joined.

Letting the key function return the data item itself, or one of its attributes that uniquely identifies it, allows us to make the joins exactly as stated above.

```
svg.selectAll("circle")
        .data(data, function(d) { return d; });
```

More on joining data and elements here: https://bost.ocks.org/mike/constancy/

Exercise

Hint 1: There is the *.remove()* method to remove the elements in the selection.

Hint 2: You can store selections *with* data bound to it in a variable. E.g.:

```
var rects = svg.selectAll("rect")
                .data([1, 2, 3, 4, 5]);
```

- Reuse your previous code (you might have to rearrange its structure a bit in the course of this exercise).
- Load an additional dataset: http://vda.univie.ac.at/Teaching/Vis/16s/data/fruit_updated.csv
- Create a JavaSript function (e.g. named OnMyButtonClick()) and add a button that calls it.
- On calling the function change the barchart to depict the new data. Meaning that bars, axes, and bar labels should reflect the change of the dataset. Use enter() and exit() to achieve that.
- **Bonus**: Let the user switch between the datasets by repeatedly clicking the button.
    - Show an additional label/text/title indicating what dataset is active.

Acknowledgements

Many thanks to Michael Opperman for all the help and material he provided.