# 2-4, Lesson 1 - Dynamic Attribute Selection

This module will cover adding linked interactions to your D3 coordinated visualization. The first lesson will implement methods for allowing users to change the expressed attribute. It will also introduce you to D3 Transitions, demonstrating how they can be used to add attractive feedback to the user in response to a change in attributes or other interactions. The second lesson will implement a *retrieve* operator linked to both map and visualization. The third and final lesson will share web hosting resources that you can use to deploy your coordinated visualization.

When you have finished this module, you should be able to:

- Implement a dropdown menu affordance and feedback that smoothly changes the expressed attribute in both the map and the data visualization
- Implement a *retrieve* operator that links enumeration units on the map to elements in the data visualization.

---

## 1.1 Selection Affordance

So far, we have only been working with one expressed attribute to create the map and data visualization for the D3 lab assignment. However, your dataset should come with at least *five* attributes. The first step in allowing the user to change the expressed attribute is to provide a visual affordance. Recall that a **visual affordance** is a signal given by the interface to the user about how to interact with the interface. You might also call an affordance a handle, button, or widget.

In previous Module, you created affordances for the *sequence* operator in the form of map interface controls—a slider and/or skip buttons. For the D3 lab assignment, you should create an affordance that makes use of an appropriate visual metaphor. For instance, a slider would be an inappropriate affordance for changing attributes that are not in time series or otherwise sequential. This tutorial makes use of a simple HTML `<select>` element, which provides a dropdown menu for attribute selection. You should think about whether this type of affordance makes sense for your coordinated visualization or if another type would be more appropriate or visually appealing to the user. If you do use a `<select>` element, make sure to position it on the page so it fits with principles of usability and good design.

To start out, we will need to add a `<select>` (http://www.w3schools.com/tags/tag_select.asp) dropdown menu to the DOM. Note that the `<select>` element is merely a container for the menu; we will also need to add each value we want in the menu as an `<option>` (http://www.w3schools.com/tags/tag_option.asp) child element. By now, you probably have some idea of how to accomplish both of these tasks using D3 selections. Example 1.1 shows one possible set of selection blocks:

**Example 1.1**: Adding a dropdown menu in *main.js*

JavaScript

```
//function to create a dropdown menu for attribute selection
function createDropdown(){
    //add select element
    var dropdown = d3.select("body")
        .append("select")
        .attr("class", "dropdown");

    //add initial option
    var titleOption = dropdown.append("option")
        .attr("class", "titleOption")
        .attr("disabled", "true")
        .text("Select Attribute");

    //add attribute name options
    var attrOptions = dropdown.selectAll("attrOptions")
        .data(attrArray)
        .enter()
        .append("option")
        .attr("value", function(d){ return d })
        .text(function(d){ return d });
};
```

In Example 1.1, we create a new function, `createDropdown()`, which is called from the end of the `callback()` function (not shown). The first block appends the `<select>` element to the `<body>` (lines 4-6). The `titleOption` block creates an `<option>` element with no `value` attribute and instructional text to serve as an affordance alerting users that they should interact with the dropdown menu (lines 9-12). Disabling the title option ensures that the user cannot mistakenly select it (line 11). Finally, the `attrOptions` block uses the `.selectAll().data().enter()` sequence with the `attrArray` pseudo-global variable that holds an array of our attribute names, creating one `<option>` element for each attribute (lines 15-18). Each option element is assigned a `value` attribute that holds the name of the attribute, and its text content (what the user sees) is also assigned the name of the attribute (lines 19-20).

Once we have created the dropdown menu, we need to do a little styling so that it does not simply appear below the previous element on the page (Example 1.2):

**Example 1.2**: Styling the dropdown in *style.css*

CSS

```
.dropdown {
    position: absolute;
    top: 30px;
    left: 40px;
    z-index: 10;
    font-family: sans-serif;
    font-size: 1em;
    font-weight: bold;
    padding: 2px;
    border: 1px solid #999;
    box-shadow: 2px 2px 4px #999;
}
```

```
option {
    font-weight: normal;
}
```

In Example 1.2, we position the dropdown `<select>` element absolutely so that it is not affected by other elements on the page (line 2). We then use `top` and `left` styles to position it down and to the right of the top-left corner of the page (lines 3-4). Adding a `z-index` of 10 ensures that it floats to the top of all other elements on the page (line 5). We then add some `font` styles and `padding` around the text (lines 6-9). Finally, a `border` and `box-shadow` make the `<select>` element visually float above the map, making it more obvious to the user (lines 10-11). The `option` style simply reduces the text of the `<option>` elements in the dropdown menu to normal weight so they are not emboldened by the `font-weight` of the `<select>` element (lines 14-16).

We can now see our dropdown menu with each of our attribute options atop the map (Figure 1.1):

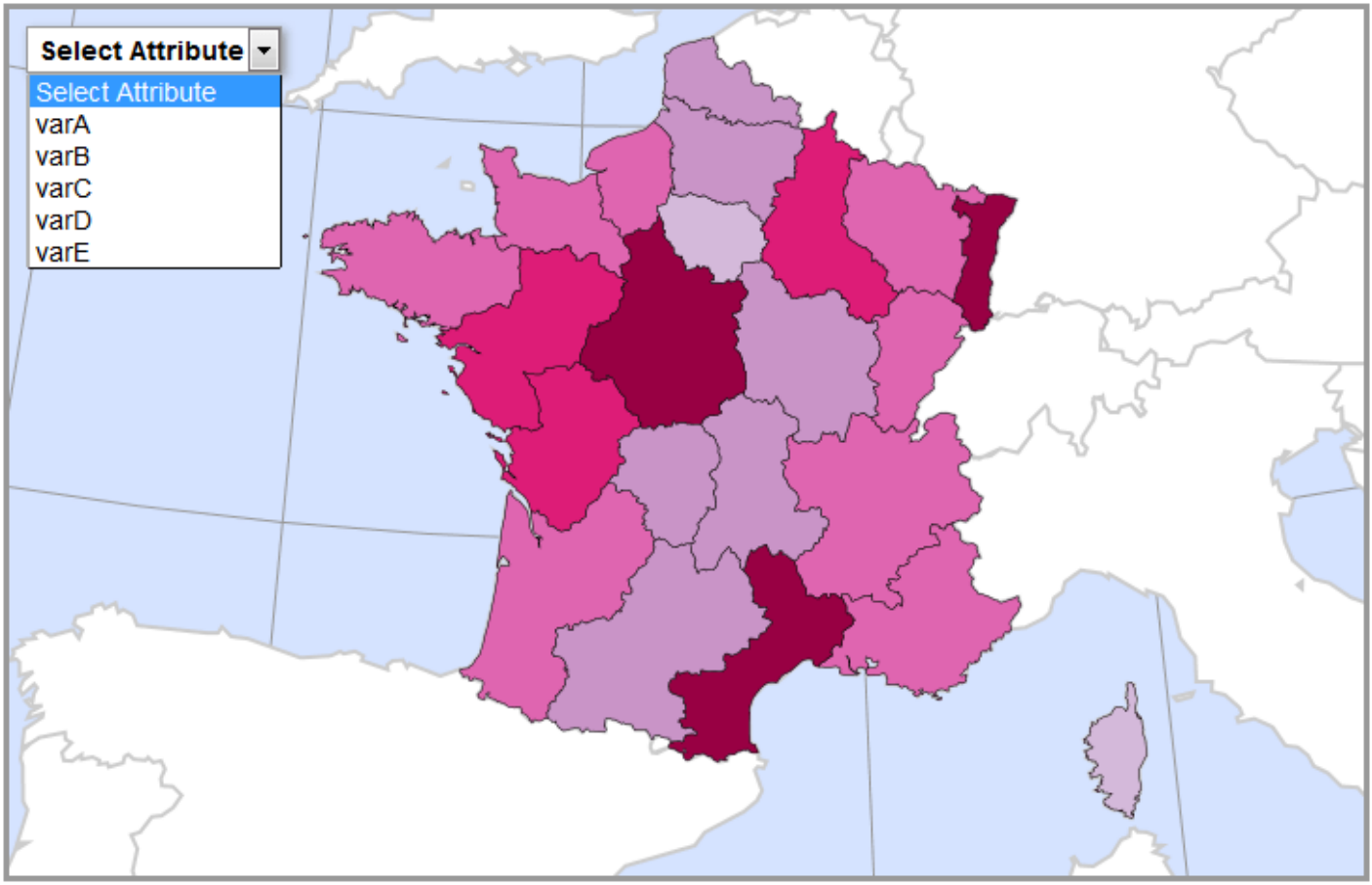


*Figure 1.1: Attribute selection dropdown menu*

## 1.2 Selection Feedback

Once our menu is in place, we need to enable it by adding an event listener to the script to listen for a user selection, and a listener handler function to respond by changing the expressed attribute (thus

giving **visual feedback** to the user). Let's pseudo-code the feedback tasks (Example 1.3):

**Example 1.3**: Pseudo-code for attribute change listener

```
// ON USER SELECTION:
// 1. Change the expressed attribute
// 2. Recreate the color scale with new class breaks
// 3. Recolor each enumeration unit on the map
// 4. Re-sort each bar on the bar chart
// 5. Resize each bar on the bar chart
// 6. Recolor each bar on the bar chart
```

The first three tasks are relatively simple to take care of within a listener handler function (Example 1.4):

**Example 1.4**: Adding a change listener and handler function in *main.js*

JavaScript

```
//Example 1.1 line 1...function to create a dropdown menu for attribute selection
function createDropdown(csvData){
    //add select element
    var dropdown = d3.select("body")
        .append("select")
        .attr("class", "dropdown")
        .on("change", function(){
            changeAttribute(this.value, csvData)
        });

    //OPTIONS BLOCKS FROM EXAMPLE 1.1 LINES 8-19
};

//dropdown change listener handler
function changeAttribute(attribute, csvData){
    //change the expressed attribute
    expressed = attribute;

    //recreate the color scale
    var colorScale = makeColorScale(csvData);

    //recolor enumeration units
    var regions = d3.selectAll(".regions")
        .style("fill", function(d){
            return choropleth(d.properties, colorScale)
        });
};
```

In Example 1.4, we add a `.on()` [(https://github.com/d3/d3-selection#handling-events)](https://github.com/d3/d3-selection#handling-events) operator to the end of the `dropdown` block to listen for a `"change"` interaction on the `<select>` element (line 7). In this context, `.on()` is a D3 method, but it works similarly to Leaflet's `.on()` method. We pass it an anonymous function, within which we call our new listener handler, `changeAttribute()` (lines 7-9). The parameters of `changeAttribute()` are the `value` of the `<select>` element (referenced by `this`), which holds the attribute selected by the user, as well as our `csvData`. The `csvData` will be used to recreate the

color scale. Note that we also need to add it as a parameter to the `createDropdown()` function (line 2) and its function call within the `callback()` (not shown).

Within `changeAttribute()`, we take care of the first task by simply assigning the user-selected attribute to the `expressed` pseudo-global variable (line 17). For the second task, we repeat the call to `makeColorScale()`, passing the scale generator our `csvData` and assigning the returned scale to a new `colorScale` variable (line 20). For the third task, we create a selection of all enumeration units (line 23). Since these already have our GeoJSON data attached to them as a hidden property in the DOM, we can easily re-use their GeoJSON `properties` with the `choropleth()` function to reset each enumeration unit's `fill` attribute (lines 24-26).

The map should now recolor itself when a new attribute is selected from the dropdown menu (Figure 1.2):

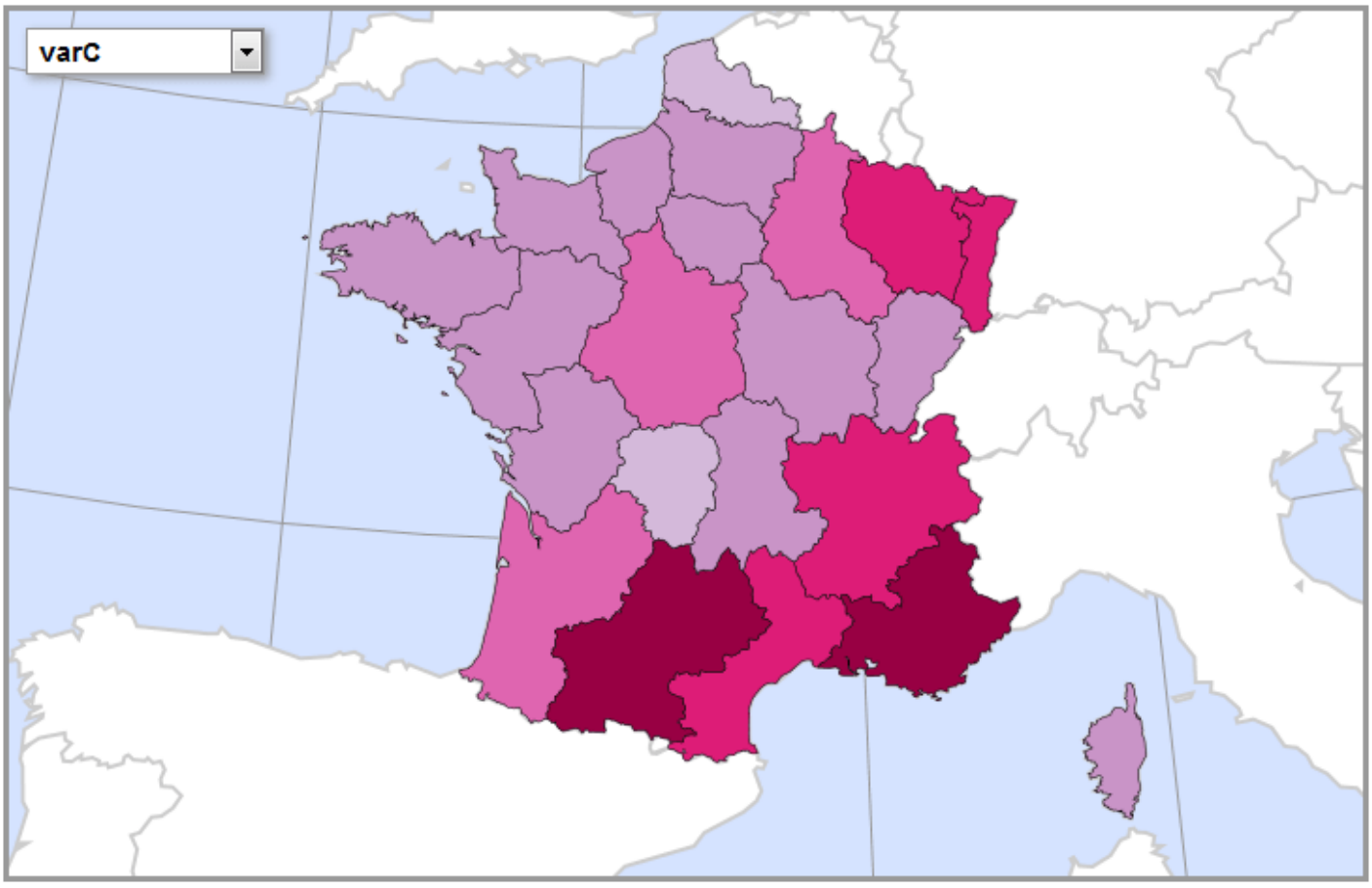


*Figure 1.2: Dynamic attribute selection changes the choropleth*

Restyling the dynamic visualization is a bit trickier, but we can use the same principle of recycling a multi-element selection that we did for recoloring the enumeration units on the map (pseudo-code task 3). The block that we build should contain a selection of all visualization elements (bars in the bar chart) and each operator that affects an aspect of the element we want to change when a new attribute is selected. For re-sorting the bars (task 4), we need `.sort()` and the `x` attribute; for resizing the bars (task 5), we need the `height` and `y` attributes; and for recoloring the bars (task 6), we need the `fill` style (Example 1.5):

**Example 1.5**: Manipulating the chart bars on attribute change in *main.js*

JavaScript

```
//Example 1.4 line 14...dropdown change listener handler
function changeAttribute(attribute, csvData){
    //change the expressed attribute
    expressed = attribute;

    //recreate the color scale
    var colorScale = makeColorScale(csvData);

    //recolor enumeration units
    var regions = d3.selectAll(".regions")
        .style("fill", function(d){
            return choropleth(d.properties, colorScale)
        });

    //re-sort, resize, and recolor bars
    var bars = d3.selectAll(".bar")
        //re-sort bars
        .sort(function(a, b){
            return b[expressed] - a[expressed];
        })
        .attr("x", function(d, i){
            return i * (chartInnerWidth / csvData.length) + leftPadding;
        })
        //resize bars
        .attr("height", function(d, i){
            return 463 - yScale(parseFloat(d[expressed]));
        })
        .attr("y", function(d, i){
            return yScale(parseFloat(d[expressed])) + topBottomPadding;
        })
        //recolor bars
        .style("fill", function(d){
            return choropleth(d, colorScale);
        });
};
```

In Example 1.5, we use `.sort()` to sort the data values for the new attribute from greatest to least (lines 18-20), then reset the `x` attribute of each bar to position the bars in the new order of the data (lines 21-23). To resize the bars, we reset the `height` attribute using our `yScale` with the new expressed attribute values (lines 25-27), then position the bars vertically by resetting the `y` attribute (lines 28-30). Finally, we recolor the bars by resetting the `fill` just as we did for the choropleth enumeration units in Example 1.4.

Note that all of these operators are simply copy-pasted from the `setChart()` function we created in previous module. The problem with doing this is that most of the anonymous functions within the operators access variables that are local to `setChart()`, including the dimension variables and our `yScale`. To make these variables accessible to the `changeAttribute()` function, we need to move them to the top of our wrapper function, declaring them as pseudo-global variables (Example 1.6):

**Example 1.6**: Moving chart variables to make them pseudo-global in *main.js*:

JavaScript

```
//Top of main.js...wrap everything in a self-executing anonymous function to move to local scope
(function(){

//pseudo-global variables
var attrArray = ["varA", "varB", "varC", "varD", "varE"]; //list of attributes
var expressed = attrArray[0]; //initial attribute

//chart frame dimensions
var chartWidth = window.innerWidth * 0.425,
    chartHeight = 473,
    leftPadding = 25,
    rightPadding = 2,
    topBottomPadding = 5,
    chartInnerWidth = chartWidth - leftPadding - rightPadding,
    chartInnerHeight = chartHeight - topBottomPadding * 2,
    translate = "translate(" + leftPadding + "," + topBottomPadding + ")";

//create a scale to size bars proportionally to frame and for axis
var yScale = d3.scale.linear()
    .range([463, 0])
    .domain([0, 110]);

//begin script when window loads
window.onload = setMap();

//...the rest of the script
```

Once we have done this, these variables are available for use by *any* function in the script. Since we copy-pasted the operators of the `bars` block that we needed to update the bars from `setChart()`, we now have a number of repetitive lines in our *main.js* script. We can clean up the script by moving these lines into their own function, called from both `setChart()` and `changeAttribute()` (Example 1.7):

**Example 1.7**: Consolidating repetitive chart script in *main.js*:

JavaScript

```
    //in setChart()...set bars for each province
    var bars = chart.selectAll(".bar")
        .data(csvData)
        .enter()
        .append("rect")
        .sort(function(a, b){
            return b[expressed]-a[expressed]
        })
        .attr("class", function(d){
            return "bar " + d.adm1_code;
        })
        .attr("width", chartInnerWidth / csvData.length - 1);

    //CHARTTITLE, YAXIS, AXIS, AND CHARTFRAME BLOCKS

    //set bar positions, heights, and colors
    updateChart(bars, csvData.length, colorScale);
```

```
}; //end of setChart()

    //...

    //in changeAttribute()...Example 1.5 line 15...re-sort bars
    var bars = d3.selectAll(".bar")
        //re-sort bars
        .sort(function(a, b){
            return b[expressed] - a[expressed];
        });

    updateChart(bars, csvData.length, colorScale);
}; //end of changeAttribute()

//function to position, size, and color bars in chart
function updateChart(bars, n, colorScale){
    //position bars
    bars.attr("x", function(d, i){
            return i * (chartInnerWidth / n) + leftPadding;
        })
        //size/resize bars
        .attr("height", function(d, i){
            return 463 - yScale(parseFloat(d[expressed]));
        })
        .attr("y", function(d, i){
            return yScale(parseFloat(d[expressed])) + topBottomPadding;
        })
        //color/recolor bars
        .style("fill", function(d){
            return choropleth(d, colorScale);
        });
};
```

In Example 1.7, the positioning, sizing, and coloring of the bars has been moved into a new `updateChart()` function, which is called from within both `setChart()` and `changeAttribute()` (lines 17 and 29). This function receives the `bars` selection, the length of the `csvData` which corresponds to the number of bars, and the `colorScale`. Note that although it is still repeated in `updateChart()` and `changeAttribute()`, we did not move the `.sort()` operator into `updateChart()` because it necessarily comes before the `class` and `width` attribute assignments in `setChart()`, which should not be repeated when the attribute is changed (lines 6-12 and 25-27).

The final step to updating the chart is to change the chart title. For this, we can simply move the `.text()` operator from the `chartTitle` block in `setChart()` into `updateChart()` (Example 1.8):

**Example 1.8**: Turning the chart title into visual feedback in *main.js*

JavaScript

```
    //at the bottom of updateChart()...add text to chart title
    var chartTitle = d3.select(".chartTitle")
        .text("Number of Variable " + expressed[3] + " in each region");
```

We now have a fully interactive choropleth map and data visualization, with the affordance of a dropdown menu and the feedback of updated enumeration units and bars (Figure 1.3):

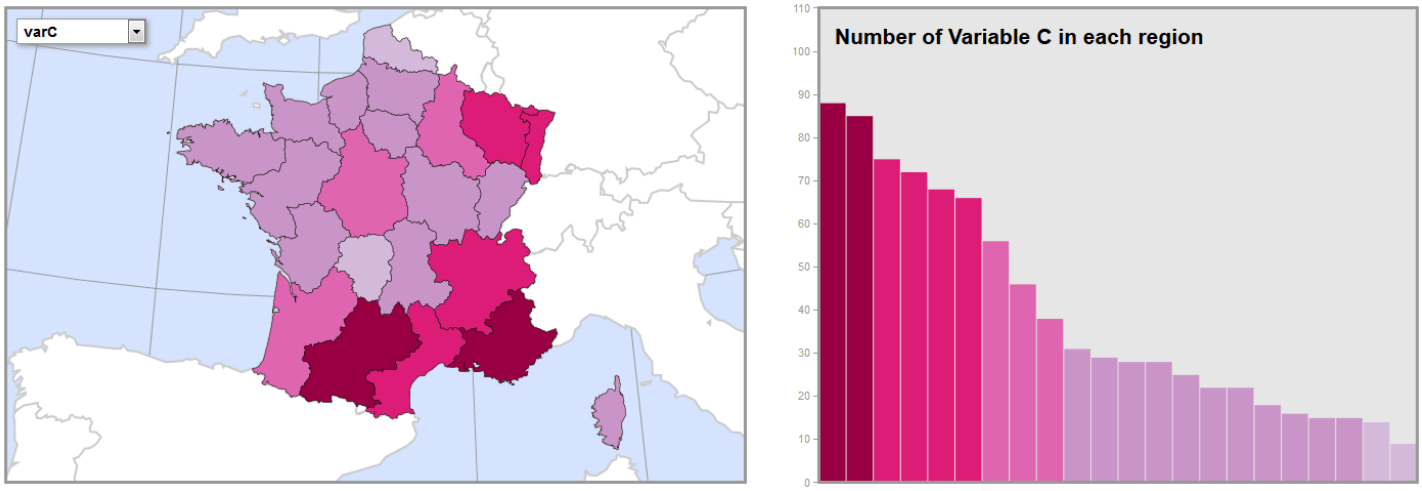


*Figure 1.3: Interactive choropleth map and chart*

> **Practice!** Implement a visual affordance allowing the user to change the expressed attribute, and feedback updating your choropleth map and data visualization in response to user input.

---

## 1.3 Transitions

Although our map and visualization now change their state in response to user input, that change isn't always noticeable, particularly when there is not a large amount of change in the data between attributes. To make the change more noticable, we can add additional feedback to the user in the form of a D3 **transition** (https://github.com/d3/d3-transition#d3-transitions). Transitions take advantage of the **animation capabilities** (http://www.w3.org/TR/SVG/animate.html) built into the SVG specification to animate between visual states. Animation guides the user's eye from one visual state to another, allowing time for the change to register cognitively. It also improves the aesthetic appeal of the graphics by making them appear to react and flow smoothly in response to user input.

We will cover only basic transitions here. You may wish to explore more deeply into transition options such as different types of **easing** (https://github.com/d3/d3-ease#d3-ease) and **interpolation** (https://github.com/d3/d3-interpolate#d3-interpolate). Each of these options has a default behavior that D3 implements automatically on any transition if they are not set manually.

The simplest and most common way to create a D3 transition is to call the operator `.transition()` in a selection block with no parameters. Every `.attr()` and `.style()` operator invoked on the selection after this will be implemented through the transition; that is, the current values for those element attributes and styles will be gradually replaced with the new values according to the default easing function or a different easing function that is specified by the `.ease()` operator. In-between values will be created by an interpolator to form the animation.

Let's start by implementing a transition on the choropleth map (Example 1.9):

**Example 1.9**: Implementing a choropleth transition in *main.js*

JavaScript

```
//Example 1.5 line 9...recolor enumeration units
var regions = d3.selectAll(".regions")
    .transition()
    .duration(1000)
    .style("fill", function(d){
        return choropleth(d.properties, colorScale)
    });
```

In Example 1.9, we modify the `regions` block in the `changeAttribute()` function, adding a `.transition()` operator and a `.duration()` operator above the `.style()` operator (lines 3-4). The `.duration()` (https://github.com/d3/d3-transition#transition_duration) operator specifies a duration in milliseconds; hence the transition will last 1000 milliseconds or 1 second. The effect is to smoothly animate between colors when the color of each enumeration units is changed in response to user input.

The bars of our bar chart can also be animated within `changeAttribute()` (Example 1.10):

**Example 1.10**: Animating the bars with `changeAttribute()`

JavaScript

```
//Example 1.7 line 22...re-sort, resize, and recolor bars
 var bars = d3.selectAll(".bar")
    //re-sort bars
    .sort(function(a, b){
        return b[expressed] - a[expressed];
    })
    .transition() //add animation
    .delay(function(d, i){
        return i * 20
    })
    .duration(500);

updateChart(bars, csvData.length, colorScale);
```

In Example 1.10, we add a `.transition()` after the data has been re-sorted according to the new expressed attribute (line 7). We then add a `.delay` (https://github.com/d3/d3-transition#transition_delay) operator with an anonymous function that delays the start of animations 20 additional milliseconds for each bar in the sequence (lines 8-10). This gives the appearance that the bars consciously rearrange themselves. The `.duration()` operator gives each bar half a second to complete its transition (line 11). When the `bars` selection is passed to `updateChart()`, the transition is passed with it, so that each of the changing attributes and the `fill` style are animated when the attribute changes (Figure 1.4).

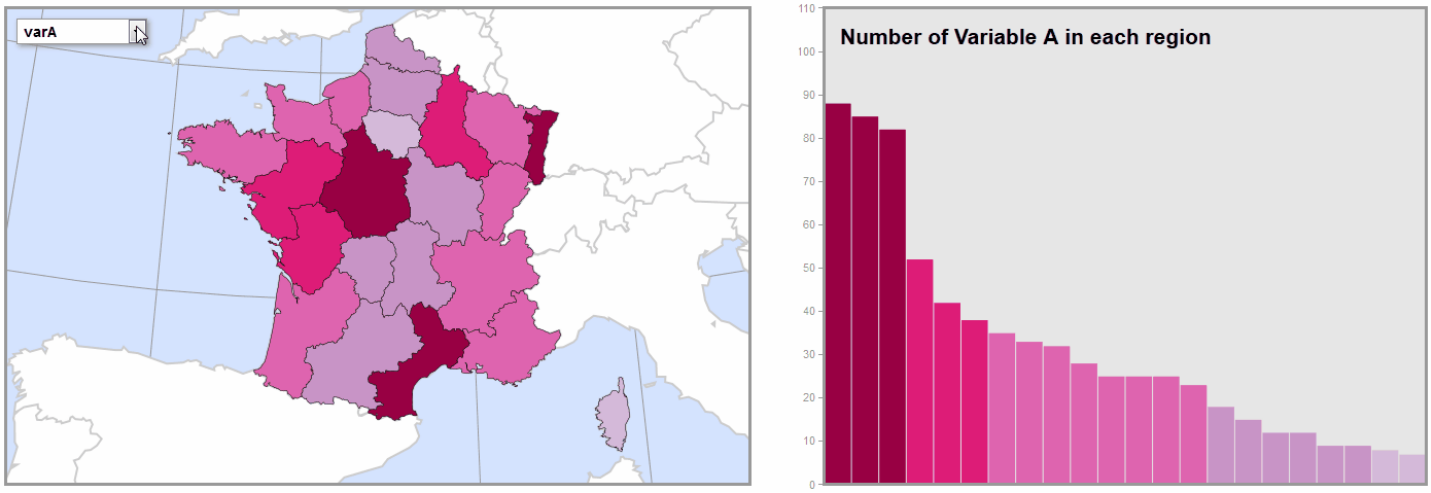


*Figure 1.4: Animated transitions between attributes on choropleth map and chart*

> **Practice!** Implement transitions in response to attribute change on your choropleth map and on your data visualization if appropriate.

---

> **Self-Check:**
>
> **1. In a dynamic bar chart, which of the following `<rect>` attributes do *not* need to be modified when the expressed attribute changes?**
>
> **a.** x
> **b.** width
> **c.** y
> **d.** height
> **e.** all of the above need to be modified
>
> **2. True/False: A slider would be an appropriate affordance for updating the map and visualization between *non-sequential* attributes because it provides the correct visual metaphor.**