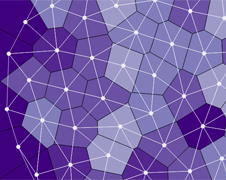
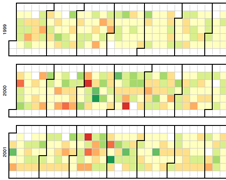
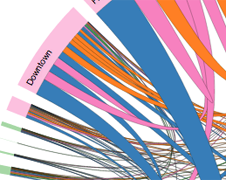
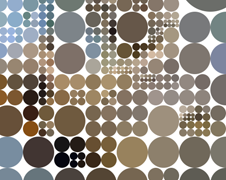
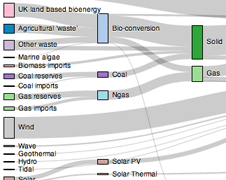
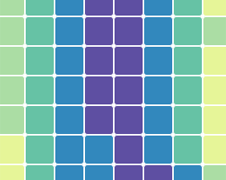
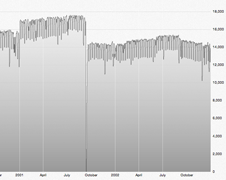
‍ [**Overview**](http://d3js.org/) [Examples](https://github.com/mbostock/d3/wiki/Gallery) [Documentation](https://github.com/mbostock/d3/wiki) [Source](https://github.com/mbostock/d3)

Data-Driven Documents

[](http://hn.metamx.com/)[](http://www.christophermanning.org/projects/voronoi-diagram-with-force-directed-nodes-and-delaunay-links/)[](http://mbostock.github.com/d3/ex/calendar.html)[](http://bost.ocks.org/mike/uberdata/)[](http://www.koalastothemax.com/)[](http://bost.ocks.org/mike/sankey/)[](http://trends.truliablog.com/2011/09/house-hunter-by-day-not-so-much-after-midnight/)[](http://mbostock.github.com/d3/talk/20111018/area-gradient.html)

See [more examples](https://github.com/mbostock/d3/wiki/Gallery).

**D3.js** is a JavaScript library for manipulating documents based on data. **D3** helps you bring data to life using HTML, SVG and CSS. D3’s **emphasis** on web standards gives you the full **capabilities** of modern browsers without tying yourself to a **proprietary** framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

Download the latest version here:

* [d3.v3.zip](http://d3js.org/d3.v3.zip)

Or, to link directly to the latest release, copy this **snippet**:

<script src="http://d3js.org/d3.v3.min.js"></script>

The [full source and tests](https://github.com/mbostock/d3) are also available [for download](https://github.com/mbostock/d3/zipball/master) on GitHub.

**[#](http://d3js.org/" \l "introduction)Introduction**

Read [more tutorials](https://github.com/mbostock/d3/wiki/Tutorials).

**D3** allows you to bind **arbitrary** data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to **generate** an HTML table from an array of numbers. Or, use the same data to create an interactive SVG **bar chart** with smooth **transitions** and interaction.

D3 is not a **monolithic** framework that seeks to provide every **conceivable** feature. Instead, D3 solves the **crux** of the problem: efficient manipulation of documents based on data. This avoids proprietary representation and **affords extraordinary flexibility**, exposing the full capabilities of web standards such as CSS3, HTML5 and SVG. With minimal **overhead**, D3 is extremely fast, supporting large datasets and dynamic behaviors for interaction and **animation**. D3’s functional style allows code reuse through a diverse collection of [components](https://github.com/mbostock/d3/wiki/API-Reference) and [plugins](https://github.com/d3/d3-plugins).

**[#](http://d3js.org/" \l "selections)Selections**

Read [more about selections](https://github.com/mbostock/d3/wiki/Selections).

Modifying documents using the [W3C DOM API](http://www.w3.org/DOM/DOMTR) is **tedious**: the method names are **verbose**, and the **imperative** approach requires **manual** **iteration** and **bookkeeping** of temporary state. For example, to change the text color of paragraph elements:

var paragraphs = document.getElementsByTagName("p");

for (var i = 0; i < paragraphs.length; i++) {

var paragraph = paragraphs.item(i);

paragraph.style.setProperty("color", "white", null);

}

D3 **employs** a declarative approach, operating on arbitrary sets of nodes called *selections*. For example, you can rewrite the above loop as:

d3.selectAll("p").style("color", "white");

Yet, you can still manipulate individual nodes as needed:

d3.select("body").style("background-color", "black");

Selectors are defined by the [W3C Selectors API](http://www.w3.org/TR/selectors-api/) and supported natively by modern browsers. **Backwards-compatibility** for older browsers can be provided by [Sizzle](http://sizzlejs.com/). The above examples select nodes by tag name ("p" and "body", respectively). Elements may be selected using a variety of **predicates**, including containment, attribute values, class and ID.

D3 provides numerous methods for **mutating** nodes: setting **attributes** or styles; registering event listeners; adding, removing or sorting nodes; and changing HTML or text content. These **suffice** for the vast majority of needs. Direct access to the underlying DOM is also possible, as each D3 selection is simply an array of nodes.

**[#](http://d3js.org/" \l "properties)Dynamic Properties**

Readers familiar with other DOM frameworks such as [jQuery](http://jquery.com/) or [Prototype](http://www.prototypejs.org/) should immediately recognize similarities with D3. Yet styles, attributes, and other properties can be specified as *functions of data* in D3, not just simple constants. Despite their apparent simplicity, these functions can be surprisingly powerful; the d3.geo.path function, for example, projects [geographic coordinates](http://geojson.org/) into SVG [path data](http://www.w3.org/TR/SVG/paths.html#PathData). D3 provides many built-in reusable functions and function factories, such as [graphical **primitives**](https://github.com/mbostock/d3/wiki/SVG-Shapes) for area, line and pie charts.

For example, to randomly color paragraphs:

d3.selectAll("p").style("color", function() {

return "hsl(" + Math.random() \* 360 + ",100%,50%)";

});

To alternate shades of gray for even and odd nodes:

d3.selectAll("p").style("color", function(d, i) {

return i % 2 ? "#fff" : "#eee";

});

Computed properties often refer to bound data. Data is specified as an array of values, and each value is passed as the first argument (d) to selection functions. With the default join-by-index, the first element in the data array is passed to the first node in the selection, the second element to the second node, and so on. For example, if you bind an array of numbers to paragraph elements, you can use these numbers to compute dynamic font sizes:

d3.selectAll("p")

.data([4, 8, 15, 16, 23, 42])

.style("font-size", function(d) { return d + "px"; });

Once the data has been bound to the document, you can omit the data operator; D3 will **retrieve** the previously-bound data. This allows you to recompute properties without rebinding.

**[#](http://d3js.org/" \l "enter-exit)Enter and Exit**

Read [more about data joins](http://bost.ocks.org/mike/join/).

Using D3’s *enter* and *exit* selections, you can create new nodes for incoming data and remove outgoing nodes that are no longer needed.

When data is bound to a selection, e ach element in the data array is paired with the corresponding node in the selection. If there are fewer nodes than data, the extra data elements form the enter selection, which you can **instantiate** by appending to the enter selection. For example:

d3.select("body").selectAll("p")

.data([4, 8, 15, 16, 23, 42])

.enter().append("p")

.text(function(d) { return "I’m number " + d + "!"; });

Updating nodes are the default selection—the result of the data operator. Thus, if you forget about the enter and exit selections, you will automatically select only the elements for which there exists corresponding data. A common pattern is to break the initial selection into three parts: the updating nodes to modify, the entering nodes to add, and the exiting nodes to remove.

// Update…

var p = d3.select("body").selectAll("p")

.data([4, 8, 15, 16, 23, 42])

.text(String);

// Enter…

p.enter().append("p")

.text(String);

// Exit…

p.exit().remove();

By handling these three cases separately, you **specify** precisely **which operations run on which nodes**. This improves performance and offers greater control over transitions. For example, with a bar chart you might initialize entering bars using the old scale, and then transition entering bars to the new scale along with the updating and exiting bars.

D3 lets you transform documents based on data; this includes both creating and destroying elements. D3 allows you to change an existing document in response to user interaction, animation over time, or even **asynchronous** notification from a **third-party**. A **hybrid** approach is even possible, where the document is initially **rendered** on the server, and updated on the client via D3.

**[#](http://d3js.org/" \l "transformation)Transformation, not Representation**

D3 is not a new graphical representation. Unlike [Processing](http://processing.org/), [Raphaël](http://raphaeljs.com/), or [Protovis](http://vis.stanford.edu/protovis/), the vocabulary of marks comes directly from web standards: HTML, SVG and CSS. For example, you can create SVG elements using D3 and style them with external stylesheets. You can use composite filter effects, dashed strokes and clipping. If browser vendors introduce new features tomorrow, you’ll be able to use them immediately—no toolkit update required. And, if you decide in the future to use a toolkit other than D3, you can take your knowledge of standards with you!

Best of all, D3 is easy to debug using the browser’s built-in element inspector: the nodes that you manipulate with D3 are exactly those that the browser understands natively.

**[#](http://d3js.org/" \l "transitions)Transitions**

D3’s focus on transformation extends naturally to animated transitions. Transitions gradually **interpolate** styles and attributes over time. **Tweening** can be controlled via **easing** functions such as “**elastic**”, “cubic-in-out” and “**linear**”. D3’s interpolators support both primitives, such as numbers and numbers **embedded** within strings (font sizes, path data, *etc.*), and **compound** values. You can even extend D3’s **interpolator** registry to support complex properties and data structures.

For example, to fade the background of the page to black:

d3.select("body").transition()

.style("background-color", "black");

Or, to resize circles in a symbol map with a **staggered** delay:

d3.selectAll("circle").transition()

.duration(750)

.delay(function(d, i) { return i \* 10; })

.attr("r", function(d) { return Math.sqrt(d \* scale); });

By modifying only the attributes that actually change, D3 reduces overhead and allows greater graphical complexity at high frame rates. D3 also allows **sequencing** of complex transitions via events. And, you can still use CSS3 transitions; D3 does not replace the browser’s toolbox, but exposes it in a way that is easier to use.

Want to learn more? Read [these **tutorials**](https://github.com/mbostock/d3/wiki/Tutorials).

Released under [BSD license](http://opensource.org/licenses/BSD-3-Clause).Copyright 2012 [Michael Bostock](http://bost.ocks.org/mike/)

[](https://github.com/mbostock/d3)

[Overview](http://d3js.org/) [Examples](https://github.com/mbostock/d3/wiki/Gallery) [**Documentation**](https://github.com/mbostock/d3/wiki) [Source](https://github.com/mbostock/d3)

d3.js

Three Little Circles

Once upon a time, there were three little circles. This tutorial shows you how to manipulate them using selections.

**Selecting Elements**

The [selectAll](https://github.com/mbostock/d3/wiki/Selections#selectAll) operator takes a [selector](http://www.w3.org/TR/CSS2/selector.html) string, such as “circle”, and returns a selection:

var circle = svg.selectAll("circle");

Run

Once we have a selection, we can apply various operators to the selected elements. For example, we might change the fill color using [style](https://github.com/mbostock/d3/wiki/Selections#style), and the radius and the *y*-position using [attr](https://github.com/mbostock/d3/wiki/Selections#attr):

circle.style("fill", "steelblue");

circle.attr("cy", 90);

circle.attr("r", 30);

Run

We can also compute the attribute values dynamically, using functions rather than constants. For example, maybe we want to set the *x*-coordinate to a random value:

circle.attr("cx", function() {

return Math.random() \* w;

});

Run

If you run this example multiple times, you’ll see that the attribute is recomputed as a number random number each time. Unlike Protovis, D3 doesn’t **stash** these functions internally; they are run once, immediately, and then your code continues. So you can run them again or redefine them however you like.

**Binding Data**

This is beginning to look a lot like [jQuery](http://jquery.com/). More commonly, though, we want to use *data* to drive the appearance of our circles. To do that, we need some data. For the sake of example, let’s imagine that each of these circles represents a number: 32, 57 and 112. The [data](https://github.com/mbostock/d3/wiki/Selections#data) operator binds these numbers to the circles:

circle.data([32, 57, 112]);

Run3257112

All data in D3 is specified as an array of values. Conveniently, this **mirrors** the concept of a selection, which is just an array of elements. Notice then how the first number (the first ***datum***, 32) is bound to the first circle (the first *element*, on top), the second number is bound to the second circle, and so on.

Once data is bound, that data is accessible as an argument to our attribute and style functions. This means we visually **encode** data, or in other words, create a visualization! For example, here we set the*x*-position and radius using the data:

circle.attr("cx", function(d) {

return d;

});

circle.attr("r", function(d) {

return Math.sqrt(d);

});

Run3257112

There’s a second argument to each function you can use: it specifies the *index* of the element within its selection. This is a [zero-based](http://en.wikipedia.org/wiki/Zero-based_numbering) index, and it’s useful for computing offsets or as a simple way of identifying individual elements. The argument is optional; if you don’t specify it when declaring your function, it will be ignored. For example:

circle.attr("cx", function(d, i) {

return i \* 100 + 30;

});

Run012

Here we use the index *i* to position the elements sequentially only the *x*-dimension. Each element is separated by 100 pixels, with an offset of 30 pixels from the left side. In SVG, the origin is in the top-left corner.

**Creating Elements**

But what if we had *four* numbers to display, rather than three? We wouldn’t have enough circles to display all the numbers. When joining data to elements, D3 stores the leftover data in the [enter](https://github.com/mbostock/d3/wiki/Selections#enter)selection. (The terms “enter” and “exit” are adopted from stage **terminology**.) Here, the fourth number 293 remains in the enter selection, because we only have three circle elements:

var circle = svg.selectAll("circle")

.data([32, 57, 112, 293]);

Run3257112293

Using the enter selection, we can create new circles for any missing data. Each new circle is already bound to the data, so we can use data to compute attributes and styles:

var enter = circle.enter().append("circle");

enter.attr("cy", 90);

enter.attr("cx", 160);

enter.attr("r", function(d) {

return Math.sqrt(d);

});

Run3257112293

Taking this to the next logical step, then, what if we have *no* existing elements? Meaning, what if the document is empty? Say we start with an empty page, and we want to create new circles that correspond to our data? Then we’re joining data to an empty selection, and all of the data ends up in *enter*:

var enter = circle.enter().append("circle");

enter.attr("cy", 90);

enter.attr("cx", function(d) {

return d;

});

enter.attr("r", function(d) {

return Math.sqrt(d);

});

Run3257112293

This pattern is so common, you’ll often see the [selectAll](https://github.com/mbostock/d3/wiki/Selections#selectAll) + [data](https://github.com/mbostock/d3/wiki/Selections#data) + [enter](https://github.com/mbostock/d3/wiki/Selections#enter) + [append](https://github.com/mbostock/d3/wiki/Selections#append) operators called sequentially, one immediately after the other. Despite it being common, keep in mind that this is just one special case of a data join; we’ve already seen another common case (selecting elements for update) and we’ll see other interesting cases to consider in a bit.

Another technique you can use to make your code more **concise** is [method chaining](http://en.wikipedia.org/wiki/Method_chaining). Each operator in D3 returns the current selection, so you can apply multiple operators sequentially. For example, the above code can be rewritten:

svg.selectAll("circle")

.data([32, 57, 112, 293])

.enter().append("circle")

.attr("cy", 90)

.attr("cx", String)

.attr("r", Math.sqrt);

Run3257112293

As you can see, the code is made even smaller using built-in JavaScript functions, rather than defining anonymous ones. The built-in [String](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/String) method, for example, is a shorthand way of using JavaScript’s default string **conversion** to compute the attribute value from the associated data. Similarly, we can plug in [Math.sqrt](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Math/sqrt) to set the radius attribute as the square root of the associated data. This technique of plugging in reusable functions to compute attribute values is used extensively in D3, particularly in conjunction with [scales](https://github.com/mbostock/d3/wiki/Quantitative-Scales) and [shapes](https://github.com/mbostock/d3/wiki/SVG-Shapes).

**Destroying Elements**

Sometimes you have the opposite problem from creation: you have too *many* existing elements, and you want to remove them. You can select nodes and [remove](https://github.com/mbostock/d3/wiki/Selections#remove) them, but more commonly, you’ll use the [exit](https://github.com/mbostock/d3/wiki/Selections#exit)selection to let D3 determine which elements are exiting the stage. The exit selection is the opposite of the enter selection: it contains all elements for which there is no corresponding data.

var circle = svg.selectAll("circle")

.data([32, 57]);

Run3257

All that’s left to do, then, is to remove the exiting elements:

circle.exit().remove();

Run3257112

The enter, update and exit selections are computed by the [data](https://github.com/mbostock/d3/wiki/Selections#data) operator, and don’t change when you append or remove elements—at least until you call [selectAll](https://github.com/mbostock/d3/wiki/Selections#selectAll) again. So, if you keep variables around that point to selections (such as circle, above), you’ll probably want to reselect after adding or removing elements.

**All Together Now**

Putting everything together, consider the three possible outcomes that result from joining data to elements:

1. enter - incoming actors, entering the stage.
2. update - persistent actors, staying on stage.
3. exit - outgoing actors, exiting the stage.

When we use the default join-by-index, either the enter or exit selection will be empty (or both): if there are more data than elements, the extra data are in the enter selection; if there are fewer data than elements, the extra elements are in the exit selection. However, by specifying a key function to the data operator, we can control exactly how data is bound to elements. And in this case, we have both enter and exit.

var circle = svg.selectAll("circle")

.data([32, 57, 293], String);

circle.enter().append("circle")

.attr("cy", 90)

.attr("cx", String)

.attr("r", Math.sqrt);

circle.exit().remove();

Run32572933257112

Want to learn more about selections and transitions? Read [A Bar Chart, Part 2](http://mbostock.github.com/d3/tutorial/bar-2.html) for a practical example of using enter and exit to display realtime data.

Copyright © 2012 [Mike Bostock](http://bost.ocks.org/mike/)

[](https://github.com/mbostock/d3)

February 5, 2012[Mike Bostock](http://bost.ocks.org/mike/)

Thinking with Joins

Say you’re making a basic scatterplot using [D3](http://d3js.org/), and you need to create some [SVG circle](http://www.w3.org/TR/SVG/shapes.html#CircleElement) elements to visualize your data. You may be surprised to discover that D3 has no primitive for creating multiple DOM elements. [**WAT?**](https://www.destroyallsoftware.com/talks/wat)

Sure, there’s the [append](https://github.com/mbostock/d3/wiki/Selections#wiki-append) method, which you can use to create a single element:

Here svg refers to a single-element selection containing an<svg> element created previously (or selected from the current page, say).

svg.append("circle")-

.attr("cx", d.x)

.attr("cy", d.y)

.attr("r", 2.5);

But that’s just a single circle, and you want *many* circles: one for each data point. Before you **bust out** a for loop and brute-force it, consider this **mystifying** sequence from one of D3’s examples:

Here data is an array of JSON objects with x and y properties, such as: [{"x": 1.0, "y":1.1}, {"x": 2.0, "y":2.5}, …].

svg.selectAll("circle")

.data(data)

.enter().append("circle")

.attr("cx", function(d) { return d.x; })

.attr("cy", function(d) { return d.y; })

.attr("r", 2.5);

This code does exactly what you need—it creates a circle element for each data point, using the xand y data properties for positioning. But what’s with the selectAll("circle")? Why do you have to select elements that don’t exist in order to create new ones? WAT.

Here’s the deal: instead of telling D3 *how to do something*, **tell D3 *what you want***. In this case, you*want* the circle elements to correspond to data: you want one circle per datum. Instead of instructing D3 to create circles, then, tell D3 that the selection "circle" should correspond to data—and describe how to get there. This concept is called the *data-join*:

This Venn diagram illustrates the data-join. Data bound to existing elements produce the *update*(inner) selection. Unbound data produce the *enter* selection (left), and unbound elements produce the *exit* selection (right).DataEnterUpdateElementsExit

Thinking with joins reveals the mystery behind the sequence:

1. The selectAll("circle") returns the empty selection, since the SVG container element (svg) is empty. No magic here.
2. The empty selection is joined to data: data(data). The [data](https://github.com/mbostock/d3/wiki/Selections#wiki-data) method binds data to elements, producing three virtual selections: *enter*, *update* and *exit*. The *enter* selection contains **placeholders** for any missing elements. The *update* selection contains existing elements, bound to data. Any remaining elements end up in the *exit* selection for removal.
3. Since the selection was empty, all data ends up as placeholder nodes in enter().

This is the same append as in the first example, but applied to multiple placeholders; selection methods **implicitly** iterate over selected elements.

1. The missing elements are added to the SVG container by append("circle").

So that’s it. You wanted the selection "circle" to correspond to data, and you described how to create the missing elements.

But why all the trouble? Why not have a primitive to create multiple elements? The beauty of the data-join is that it **generalizes**. The above code only handles the *enter* selection. That’s sufficient for static visualizations, but you can extend it to support [dynamic](http://bost.ocks.org/mike/miserables/) [visualizations](http://mbostock.github.com/d3/ex/population.html) with only minor modifications for *update* and *exit*. And that means you can visualize [realtime data](http://bost.ocks.org/mike/path/), allow [interactive exploration](http://mbostock.github.com/d3/ex/splom.html), and [transition smoothly](http://mbostock.github.com/d3/talk/20111116/transitions.html) between datasets!

Here’s an example of handling all three states:

var circle = svg.selectAll("circle")

.data(data);

circle.enter().append("circle")

.attr("r", 2.5);

circle

.attr("cx", function(d) { return d.x; })

.attr("cy", function(d) { return d.y; });

circle.exit().remove();

To control how data is **assign­ed** to elements, you can pro­vide a[key function](http://bost.ocks.org/mike/constancy/).

If we run this code repeatedly, it recomputes the data-join each time. If the new dataset is smaller than the old one, the **surplus** elements end up in the *exit* selection and get removed. If the new dataset is larger, the surplus data ends up in the *enter* selection and new nodes are added. If the new dataset is exactly the same size, then all the elements are simply updated with new positions, and no elements are added or removed.

Thinking with joins also means your code is more *declarative*: you handle these three states with no **branching** (if) and no **iteration** (for), simply by describing how elements should correspond to data. If a given *enter*, *update* or *exit* selection happens to be empty, the corresponding **chunk** of code is a **no-op** with minimal overhead.

Joins also let you target operations to specific states, if needed. For example, you can set constant attributes (such as the circle’s radius, defined by the "r" attribute) on enter rather than update. By reselecting elements and minimizing DOM changes, you vastly improve **rendering** performance! Similarly, you can target animated transitions to specific states. For example, for entering circles to expand-in:

circle.enter().append("circle")

.attr("r", 0)

.transition()

.attr("r", 2.5);

Likewise, to shrink-out:

circle.exit().transition()

.attr("r", 0)

.remove();

Now you’re thinking with joins!

Comments or questions? [Discuss on HN.](http://news.ycombinator.com/item?id=3581614)

**Addendum**

I’ve written a series of examples on the [general update pattern](http://bl.ocks.org/3808218) as a **followup** to this post.

February 5, 2012[Mike Bostock](http://bost.ocks.org/mike/)

[mbostock](http://bl.ocks.org/mbostock)’s block [#3808218](https://gist.github.com/3808218)

# General Update Pattern, I

December 14, 2012

[Open in a new window.](http://bl.ocks.org/d/3808218/)

This example demonstrates the **general update pattern** in [D3](http://d3js.org/), where a data-join is followed by operations on the enter, update and exit selections. Entering elements are shown in green, while updating elements are shown in black. Exiting elements are removed immediately, so they're invisible.

This example does not use a key function for the data-join, so entering elements are always added to the end: when the new data has more letters than the old data, new elements are entered to display the new letters. Likewise, exiting letters are always removed from the end when the new data has fewer letters than the old data.

Next: [Key Functions](http://bl.ocks.org/3808221)

## index.html[#](http://bl.ocks.org/3808218" \l "index.html)

<!DOCTYPE html>

<meta charset="utf-8">

<style>

text {

font: bold 48px monospace;

}

.enter {

fill: green;

}

.update {

fill: #333;

}

</style>

<body>

<script src="http://d3js.org/d3.v2.min.js?2.10.1"></script>

<script>

var alphabet = "abcdefghijklmnopqrstuvwxyz".split("");

var width = 960,

height = 500;

var svg = d3.select("body").append("svg")

.attr("width", width)

.attr("height", height)

.append("g")

.attr("transform", "translate(32," + (height / 2) + ")");

function update(data) {

// DATA JOIN

// Join new data with old elements, if any.

var text = svg.selectAll("text")

.data(data);

// UPDATE

// Update old elements as needed.

text.attr("class", "update");

// ENTER

// Create new elements as needed.

text.enter().append("text")

.attr("class", "enter")

.attr("x", function(d, i) { return i \* 32; })

.attr("dy", ".35em");

// ENTER + UPDATE

// Appending to the enter selection expands the update selection to include

// entering elements; so, operations on the update selection after appending to

// the enter selection will apply to both entering and updating nodes.

text.text(function(d) { return d; });

// EXIT

// Remove old elements as needed.

text.exit().remove();

}

// The initial display.

update(alphabet);

// Grab a random sample of letters from the alphabet, in alphabetical order.

setInterval(function() {

update(shuffle(alphabet)

.slice(0, Math.floor(Math.random() \* 26))

.sort());

}, 1500);

// Shuffles the input array.

function shuffle(array) {

var m = array.length, t, i;

while (m) {

i = Math.floor(Math.random() \* m--);

t = array[m], array[m] = array[i], array[i] = t;

}

return array;

}

</script>

December 14, 2012[mbostock](http://bl.ocks.org/mbostock)’s block [#3808218](https://gist.github.com/3808218)

By adding a key to the data-join, letters that are already displayed are put in the update selection. Now updates can occur anywhere in the array, depending on the **overlap** between the old letters and the new letters. The text content only needs updating on enter because the mapping from letter to element never changes; however, the *x*-position of the text element must now be recomputed on update as well as enter.

It'll be easier to see what's going on when we add animated transitions next!

Next: [Update Transitions](http://bl.ocks.org/3808234)

Previous: [General Update Pattern](http://bl.ocks.org/3808218)

**index.html**[**#**](http://bl.ocks.org/3808221#index.html)

<!DOCTYPE html>

<meta charset="utf-8">

<style>

text {

font: bold 48px monospace;

}

.enter {

fill: green;

}

.update {

fill: #333;

}

</style>

<body>

<script src="http://d3js.org/d3.v2.min.js?2.10.1"></script>

<script>

var alphabet = "abcdefghijklmnopqrstuvwxyz".split("");

var width = 960,

height = 500;

var svg = d3.select("body").append("svg")

.attr("width", width)

.attr("height", height)

.append("g")

.attr("transform", "translate(32," + (height / 2) + ")");

function update(data) {

// DATA JOIN

// Join new data with old elements, if any.

var text = svg.selectAll("text")

.data(data, function(d) { return d; });

// UPDATE

// Update old elements as needed.

text.attr("class", "update");

// ENTER

// Create new elements as needed.

text.enter().append("text")

.attr("class", "enter")

.attr("dy", ".35em")

.text(function(d) { return d; });

// ENTER + UPDATE

// Appending to the enter selection expands the update selection to include

// entering elements; so, operations on the update selection after appending to

// the enter selection will apply to both entering and updating nodes.

text.attr("x", function(d, i) { return i \* 32; })

// EXIT

// Remove old elements as needed.

text.exit().remove();

}

// The initial display.

update(alphabet);

// Grab a random sample of letters from the alphabet, in alphabetical order.

setInterval(function() {

update(shuffle(alphabet)

.slice(0, Math.floor(Math.random() \* 26))

.sort());

}, 1500);

// Shuffles the input array.

function shuffle(array) {

var m = array.length, t, i;

while (m) {

i = Math.floor(Math.random() \* m--);

t = array[m], array[m] = array[i], array[i] = t;

}

return array;

}

</script>

By adding transitions, we can more easily follow the elements as they are entered, updated and exited. Separate transitions are defined for each of the three states.

Note that no transition is applied to the merged enter + update selection; this is because it would supersede the transition already scheduled on entering and updating elements. It's possible to schedule concurrent elements by using [transition.transition](https://github.com/mbostock/d3/wiki/Transitions" \l "wiki-transition) or by setting transition.id, but it's simpler here to only transition the x-position on update; for entering elements, the x-position is assigned statically.

Want to read more? Try these tutorials:

* [Thinking with Joins](http://bost.ocks.org/mike/join/)
* [Nested Selections](http://bost.ocks.org/mike/nest/)
* [Object Constancy](http://bost.ocks.org/mike/constancy/)

See the [D3 wiki](https://github.com/mbostock/d3/wiki) for even more resources.

Previous: [Key Functions](http://bl.ocks.org/3808221)

## index.html[#](http://bl.ocks.org/3808234#index.html)

<!DOCTYPE html>

<meta charset="utf-8">

<style>

text {

font: bold 48px monospace;

}

.enter {

fill: green;

}

.update {

fill: #333;

}

.exit {

fill: brown;

}

</style>

<body>

<script src="http://d3js.org/d3.v2.min.js?2.10.1"></script>

<script>

var alphabet = "abcdefghijklmnopqrstuvwxyz".split("");

var width = 960,

height = 500;

var svg = d3.select("body").append("svg")

.attr("width", width)

.attr("height", height)

.append("g")

.attr("transform", "translate(32," + (height / 2) + ")");

function update(data) {

// DATA JOIN

// Join new data with old elements, if any.

var text = svg.selectAll("text")

.data(data, function(d) { return d; });

// UPDATE

// Update old elements as needed.

text.attr("class", "update")

.transition()

.duration(750)

.attr("x", function(d, i) { return i \* 32; });

// ENTER

// Create new elements as needed.

text.enter().append("text")

.attr("class", "enter")

.attr("dy", ".35em")

.attr("y", -60)

.attr("x", function(d, i) { return i \* 32; })

.style("fill-opacity", 1e-6)

.text(function(d) { return d; })

.transition()

.duration(750)

.attr("y", 0)

.style("fill-opacity", 1);

// EXIT

// Remove old elements as needed.

text.exit()

.attr("class", "exit")

.transition()

.duration(750)

.attr("y", 60)

.style("fill-opacity", 1e-6)

.remove();

}

// The initial display.

update(alphabet);

// Grab a random sample of letters from the alphabet, in alphabetical order.

setInterval(function() {

update(shuffle(alphabet)

.slice(0, Math.floor(Math.random() \* 26))

.sort());

}, 1500);

// Shuffles the input array.

function shuffle(array) {

var m = array.length, t, i;

while (m) {

i = Math.floor(Math.random() \* m--);

t = array[m], array[m] = array[i], array[i] = t;

}

return array;

}

</script>

# Nested Selections

D3’s [selections](https://github.com/mbostock/d3/wiki/Selections) can be **hierarchical**, much like the elements and data they join. Consider a table:

<table>

<thead>

<tr><td> A</td><td> B</td><td> C</td><td> D</td></tr>

</thead>

<tbody>

<tr><td> 0</td><td> 1</td><td> 2</td><td> 3</td></tr>

<tr><td> 4</td><td> 5</td><td> 6</td><td> 7</td></tr>

<tr><td> 8</td><td> 9</td><td> 10</td><td> 11</td></tr>

<tr><td> 12</td><td> 13</td><td> 14</td><td> 15</td></tr>

</tbody>

</table>

How would you select only the body cells? The selector "td" would match the td elements in the head as well as the body. To match only those elements *A* within some other elements *B*, use the[descendant combinator](http://www.w3.org/TR/selectors/#descendant-combinators), "B A". For example:

There are other combinators. For example, a comma results in a union: "th, td" matches bothth *and* td elements.

var td = d3.selectAll("tbody td");

Alternatively, select the tbody element first, then select the td elements within:

var td = d3.select("tbody").selectAll("td");

This produces the same result because [selectAll](https://github.com/mbostock/d3/wiki/Selections" \l "wiki-selectAll), for each element in the current selection, selects the matching **descendants**. This is convenient if you want to **derive** multiple selections from the same parent, such as splitting the even and odd rows of a table.

## [#](http://bost.ocks.org/mike/nest/" \l "index)Nesting and Index

If you select the td elements using [d3.selectAll](https://github.com/mbostock/d3/wiki/Selections#wiki-d3_selectAll), you get a flat selection, like so:

selectionparentNode: htmltd0td1td2td3td4td5td6td7td8td9td10td11td12td13td14td15

var td = d3.selectAll("tbody td");

**Flat** selections lack hierarchical structure: the table cells are **merged** into a single array, rather than grouped by parent row. This makes them more difficult to manipulate on a row- or column-basis. In contrast, D3’s nested selections **retain** the hierarchy. To group by row, first select the tr elements, then select the td elements:

The concept of nested selections is one of the main differences between D3 and other DOM libraries, such as jQuery and Prototype.selectionparentNode: trtd0td1td2td3parentNode: trtd4td5td6td7parentNode: trtd8td9td10td11parentNode: trtd12td13td14td15

var td = d3.selectAll("tbody tr").selectAll("td");

Now if you want to color the first column red, use the index i:

This **ternary** expression returns null if i is a truthy value (non-zero), and "red" if is not. By returning null for the other columns, the color is **inherited** from the current stylesheet.

td.style("color", function(d, i) { return i ? null : "red"; });

You can also access the row index (j) by adding a third argument to the function.

## [#](http://bost.ocks.org/mike/nest/" \l "data)Nesting and Data

Hierarchical elements are often driven by similarly hierarchical data; nested selections are therefore convenient for binding data, too. To continue the example, you might **represent** the table’s associated data as a matrix (an array of arrays):

var matrix = [

[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11],

[12, 13, 14, 15],

];

To join the numbers to the corresponding table cells, first join the outer array (matrix) to the rows, and then join the inner arrays (matrix[0], matrix[1], …) to the cells:

var td = d3.selectAll("tbody tr")

.data(matrix)

.selectAll("td")

.data(function(d, i) { return d; }); // d is matrix[i]

Notice that the [data](https://github.com/mbostock/d3/wiki/Selections#wiki-data) method can either take an *array* (such as matrix) or a *function* that returns an array. Arrays are often used with flat selections, since flat selections only have one group, while nested selections typically require a function. The initial row selection is flat, since it was created with d3.selectAll:

selectionparentNode: htmltr…tr…tr…tr…

var tr = d3.selectAll("tbody tr")

.data(matrix);

The cell selection, in contrast, is nested:

selectionparentNode: trtd0td1td2td3parentNode: trtd4td5td6td7parentNode: trtd8td9td10td11parentNode: trtd12td13td14td15

var td = tr.selectAll("td")

.data(function(d) { return d; }); // matrix[i]

The data operator defines the array of data for each group. Here the data function is invoked once per row and successively passed each parent datum. Since the parent data is an array of arrays, the data function simply returns the inner array for each row of cells.

## [#](http://bost.ocks.org/mike/nest/" \l "parent)Nesting and the Parent Node

See [Thinking with Joins](http://bost.ocks.org/mike/join/) for a quick overview of D3’s data-join concept.

Nesting selections has another subtle yet critical side-effect: it sets the *parent node* for each group. The parent node is a hidden property on selections that determines where to append entering elements. For example, if you attempt a data-join on a top-level selection, you get an error:

selectionparentNode: htmltr…tr…tr…tr…

d3.selectAll("table tr")

.data(matrix)

.enter().append("tr"); // error!

The error occurs because the default parent node is the document’s root (html) element, and you can’t add tr elements directly to the root. Instead, select a parent before performing the data join:

selectionparentNode: tabletr…tr…tr…tr…

d3.select("table").selectAll("tr")

.data(matrix)

.enter().append("tr"); // success

This approach extends to arbitrary levels of nested selection. Say you wanted to create the table from scratch, given the matrix of numbers. Start by selecting the body:

selectionparentNode: htmlbody…

var body = d3.select("body");

Next [append](https://github.com/mbostock/d3/wiki/Selections#wiki-append) a table element to the body:

selectionparentNode: htmltable…

var table = body.append("table");

Now append entering rows to the table via data-join. Since selectAll is called on the selected table element, it establishes a new parent node:

selectionparentNode: tabletr…tr…tr…tr…

var tr = table.selectAll("tr")

.data(matrix)

.enter().append("tr");

Finally, append entering cells to each row:

selectionparentNode: trtd0td1td2td3parentNode: trtd4td5td6td7parentNode: trtd8td9td10td11parentNode: trtd12td13td14td15

var td = tr.selectAll("td")

.data(function(d) { return d; })

.enter().append("td");

## [#](http://bost.ocks.org/mike/nest/" \l "conclusion)To Nest, or not to Nest

There is an important difference between [select](https://github.com/mbostock/d3/wiki/Selections#wiki-select) and [selectAll](https://github.com/mbostock/d3/wiki/Selections" \l "wiki-selectAll): select *preserves* the existing grouping, whereas selectAll *creates* a new grouping. Calling select thus preserves the data, index and even the parent node of the original selection! For example, the following selection is flat, with the parent node still the document root:

selectionparentNode: htmltd0td4td8td12

var td = d3.selectAll("tbody tr").select("td");

The only way to obtain a nested selection, then, is to call selectAll on an existing selection; this is why a data-join typically follows a selectAll rather than select.

This tutorial employed tables as an example of hierarchical structure. This is but one contrived example—nested selections are [surprisingly](http://bost.ocks.org/mike/miserables/) [common](http://bl.ocks.org/882152)! By making only a few code changes, nested selections can quickly turn any singular visualization into [small multiples](http://bl.ocks.org/1305111). Much like [thinking with joins](http://bost.ocks.org/mike/join/), nested selections require a different mental model for creating and manipulating elements. But once mastered, they provide a concise way of creating [data-driven documents](http://d3js.org/).

**Top States by Age Bracket, 2008**  
Age: 

This bar chart shows the top ten states for a given age bracket, sorted by population percentage. For example, Utah’s burgeoning youth population earns it the top spot in the [5 to 13](javascript:javascript:menu.property('value',%20'5%20to%2013%20Years'),change()) (15.1%) and [under 5](javascript:javascript:menu.property('value',%20'Under%205%20Years'),change()) (9.8%) brackets, while Florida is [popular with retirees](javascript:menu.property('value',%20'65%20Years%20and%20Over'),change()) (17.4%).

The chart shows multiple slices of a dataset, transitioning smoothly when the age bracket changes. The *x*-axis rescales to accommodate the change in maximum value, while bars reshuffle along the *y*-axis to preserve sorted order. Graphical elements enter and exit: Hawaii enters the top ten for the 65 and older age brackets, but fades out in younger ones. The axis ticks change suitably, from whole percentages to fifths. Old values fade-out while the new values fade-in, both translating to preserve a valid display across the transition.

Animated transitions are pretty, but they also serve a purpose: they make it easier to follow the data. This is known as *object constancy*: a graphical element that represents a particular data point (such as Ohio) can be tracked visually through the transition. This lessens the cognitive burden by using preattentive processing of motion rather than sequential scanning of labels.

## [#](http://bost.ocks.org/mike/constancy/" \l "key-functions)Key Functions

To achieve object constancy with [D3.js](http://d3js.org/), specify a **key function** as the second argument to[selection.data](https://github.com/mbostock/d3/wiki/Selections#wiki-data). This function takes a data point as input and returns a corresponding key: a string, such as a name, that uniquely identifies the data point. For example, the bar chart above defines data as objects:

{

"State": "ND",

"Total": 641481,

"Under 5 Years": 0.065,

"5 to 13 Years": 0.105,

"14 to 17 Years": 0.053,

"18 to 24 Years": 0.129,

"16 Years and Over": 0.804,

"18 Years and Over": 0.777,

"15 to 44 Years": 0.410,

"45 to 64 Years": 0.260,

"65 Years and Over": 0.147,

"85 Years and Over": 0.028

}

A suitable key function for this data returns the State property, a [FIPS code](http://www.itl.nist.gov/fipspubs/fip5-2.htm):

function key(d) {

return d.State;

}

When you join the top-ten states to the bars, three selections are returned:

var bar = svg.selectAll(".bar")

.data(top, function(d) { return d.State; });

The selection bar is the *update* selection: states that persist across the transition. The selectionsbar.enter() and bar.exit() are the *enter* and *exit* selections: states that are incoming or outgoing, respectively. For more on these three selections, see [Thinking with Joins](http://bost.ocks.org/mike/join/).

For example, when changing from the 18-24 bracket to 14-17, Alaska moves from spot #5 to #1. Since it is in the top ten in both age brackets, it is in the update selection. An update transition interpolates the transform attribute to translate Alaska smoothly to its new position. Simultaneous subtransitions resize the bar and reposition the associated label:

var barUpdate = d3.transition(bar)

.attr("transform", function(d) { return "translate(0," + y(d.State) + ")"; });

barUpdate.select("rect")

.attr("width", function(d) { return x(d[age]); });

barUpdate.select("text")

.attr("x", function(d) { return x(d[age]) - 3; })

.text(function(d) { return format(d[age]); });

Transitions are also used to fade entering and exiting elements. For the full code, [view source](https://github.com/mbostock/bost.ocks.org/blob/gh-pages/mike/constancy/index.html).

Key functions can be useful for improving performance independent of transitions. For example, if you filter a large table, you can use a key function to reduce the number of DOM modifications: reorder DOM elements in the update selection rather than regenerating them. We used this technique at Square to improve the performance of [merchant analytics](http://corner.squareup.com/2012/04/building-analytics.html), and it’s one of the reasons that D3 is faster than most template frameworks.

## [#](http://bost.ocks.org/mike/constancy/" \l "when-constancy-matter)When Constancy Matters

Above all, animation should be meaningful. While it may be visually impressive for bars to fly around the screen during transitions, animation should only be used when it enhances understanding. Transitions between unrelated datasets or dimensions (*e.g.*, from temperature to stock price) should use a simpler cross-fade or cut rather than gratuitous, nonsensical movement.

Use a key function whenever you want to follow graphical elements through animation and interaction: filtering (adding or removing elements), reordering (sorting), switching dimensions within multivariate data, *etc.* If you forget to specify a key function, the default join-by-index can be misleading! Assist your viewers by maintaining object constancy.

# A Bar Chart, Part 1

Say you have some data—a simple array of numbers:

1 **var** data = [4, 8, 15, 16, 23, 42];

One of the ways you might visualize this univariate data is a bar chart. This guide will examine how to create a simple bar chart using D3, first with basic HTML, and then a more advanced example with SVG.

## HTML

To get started with HTML, you’ll first need a container for the chart:

1 **var** chart = d3.select("body").append("div")

2 .attr("class", "chart");

This code selects the document body, which will be the parent of the new chart. Every visible node needs a parent, with the exception of the document’s root node. The chart container, a div element, is then created and appended to the body. The append operator adds the new node as the last child: the chart will appear at the end of the body.

D3 uses the [method chaining](http://en.wikipedia.org/wiki/Method_chaining) design pattern. Above, setting the attribute returns the current selection, and the chart variable thus refers to the chart container element. This approach minimizes the amount of code needed to apply many selections and transformations in sequence.

The attr operator sets the “class” attribute on the chart container, allowing stylesheets to be applied to the chart elements. This is convenient for static styles, such as the background color and font size. CSS code lives in a style element or an external stylesheet referenced by a link element, rather than the script element used for JavaScript:

1 **.chart** div {

2 **font**: 10px **sans-serif**;

3 **background-color**: steelblue;

4 **text-align**: **right**;

5 **padding**: 3px;

6 **margin**: 1px;

7 **color**: white;

8 }

Next, add some div elements to the container, setting the width by scaling the data:

1 chart.selectAll("div")

2 .data(data)

3 .enter().append("div")

4 .style("width", **function**(d) { **return** d \* 10 + "px"; })

5 .text(**function**(d) { **return** d; });

This code selects the child div elements of the chart container. This selection is empty because the container was just added. However, by binding this selection to the array of numbers via the dataoperator, you can obtain the entering selection—a set of placeholder nodes, one per data element, to which you can append the desired child nodes for each bar.

The text operator sets the text content of the bars. The identity function, function(d) { return d; }, causes each data value (number) to be formatted using JavaScript’s default string conversion, equivalent to the built-in String function. This may be ugly for some numbers (e.g., 0.12000000000000001). The d3.format class, modeled after Python’s [string formatting](http://docs.python.org/library/stdtypes.html#string-formatting), is available for more control over how the number is formatted, supporting comma-grouping of thousands and fixed precision.

The above code results in a rudimentary bar chart:

4

8

15

16

23

42

One weakness of the code so far is the [magic number](http://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants) 10, which scales the data value to the appropriate bar width. This number depends on the domain of the data (the maximum value, 42), and the width of the chart (420). To avoid hard-coding the x-scale of 10, you can use D3’s linear scale class, and compute the maximum value from the data:

1 **var** x = d3.scale.linear()

2 .domain([0, d3.max(data)])

3 .range(["0px", "420px"]);

Although it looks like an object, the x variable here is actually a function that converts data values (in the domain) to scaled values (in the range). For example, an input value of 4 returns “40px”, and an input value of 16 returns “160px”. The output range of the scale in this example are strings, with the appropriate px units for CSS. D3’s automatic interpolators detect the numbers within the strings, while retaining the constant remainder.

The new scale arguably still has a magic number: 420px, the width of the chart. If you want to make the chart resizable, you can inspect the width of the chart container, chart.style("width"). Or, use percentages rather than pixels. In any case, the reusable scale makes the chart specification easier to modify—for example, you can easily replace the linear scale with a log or root scale.

Using the new scale, you can simplify the width style definition:

1 chart.selectAll("div")

2 .data(data)

3 .enter().append("div")

4 .style("width", x)

5 .text(String);

Now, the HTML representation is very concise, but it’s not very flexible. Displaying reference lines in the background, or generating columns rather than bars, is difficult in pure HTML. Chart types such as pies and streamgraphs are practically impossible. Fortunately, there’s a convenient alternative:[Scalable Vector Graphics](http://en.wikipedia.org/wiki/Scalable_Vector_Graphics) (SVG)!

## SVG

You use SVG much the same way as HTML, but it offers substantially more flexibility. To start with SVG, create an svg container instead of a div:

1 **var** chart = d3.select("body").append("svg")

2 .attr("class", "chart")

3 .attr("width", 420)

4 .attr("height", 20 \* data.length);

An immediate difference you will notice with SVG is that the units are implicitly pixels, and thus do not need the “px” suffix. Even with pixels, you can rescale the SVG without losing image quality. You can use percentages for relative positioning, too. To use a numeric range for the x-scale:

1 **var** x = d3.scale.linear()

2 .domain([0, d3.max(data)])

3 .range([0, 420]);

Unlike HTML, SVG does not provide automatic flow layout. Shapes are positioned relative to the top-left corner, called the origin. Thus, by default, the bars would be drawn on top of each other. To fix this, set the y-coordinate and height explicitly:

1 chart.selectAll("rect")

2 .data(data)

3 .enter().append("rect")

4 .attr("y", **function**(d, i) { **return** i \* 20; })

5 .attr("width", x)

6 .attr("height", 20);

Also, the CSS changes slightly when using SVG. Rather than the background, the fill determines the bar color. You can also apply a white border to each bar by setting the stroke style:

1 **.chart** rect {

2 stroke: white;

3 fill: steelblue;

4 }

The SVG-based chart is now almost identical to our original. The chart is currently missing labels, but that will be fixed shortly:

Astute readers will notice that a magic number crept back into the chart description: the bar height of 20 pixels. Arguably, this number isn’t magic—twenty is a reasonable height for the bars, just as fourteen is a reasonable point size for text. However, if you prefer to set a height for the entire chart, use a second scale for the y-axis:

1 **var** y = d3.scale.ordinal()

2 .domain(data)

3 .rangeBands([0, 120]);

As with x previously, y is now a function. It takes as input values from the data array, and for each value returns the corresponding y-coordinate. For example, an input value of 4 returns 0, and an input value of 16 returns 60. This approach requires that the values in our dataset are unique; ordinal scales are often used to encode non-quantitative data, such as country names. Alternatively, you can use array indices as the ordinal domain: [0, 1, 2…].

The new scale plugs into the bar specification, replacing the “y” attribute:

1 chart.selectAll("rect")

2 .data(data)

3 .enter().append("rect")

4 .attr("y", y)

5 .attr("width", x)

6 .attr("height", y.rangeBand());

The new scales can also be applied to render labels, showing the associated value. This code centers labels vertically within each bar, and right-aligns text:

1 chart.selectAll("text")

2 .data(data)

3 .enter().append("text")

4 .attr("x", x)

5 .attr("y", **function**(d) { **return** y(d) + y.rangeBand() / 2; })

6 .attr("dx", -3) *// padding-right*

7 .attr("dy", ".35em") *// vertical-align: middle*

8 .attr("text-anchor", "end") *// text-align: right*

9 .text(String);

The formal [SVG Text](http://www.w3.org/TR/SVG/text.html) specification describes in detail the meaning of the “dx”, “dy” and “text-anchor” attributes. The full spec is dense, as SVG offers a level of control required by only the most ambitious typographers; fortunately, it’s not too hard to remember standard settings for alignment and padding!

The SVG chart now looks identical to the earlier HTML version:

4815162342

With the basic chart is in place, you can place additional marks to improve readability. As a first step, pad the SVG container to make space for labels:

1 **var** chart = d3.select("body").append("svg")

2 .attr("class", "chart")

3 .attr("width", 440)

4 .attr("height", 140)

5 .append("g")

6 .attr("transform", "translate(10,15)");

The g element is a [container element](http://www.w3.org/TR/SVG/struct.html), much like the div element in HTML. Setting a [transform](http://www.w3.org/TR/SVG/coords.html#TransformAttribute) on a container affects how its children are positioned. For padding, you need only to translate; however, for advanced graphical effects, you can use any affine transformation, such as scale, rotate and shear!

The linear scale, x, provides a ticks routine that generates values in the domain at sensible intervals. For a chart this size, about ten ticks is appropriate; for smaller or larger charts, you can vary the number of ticks to generate. These tick values serve as data for reference lines:

1 chart.selectAll("line")

2 .data(x.ticks(10))

3 .enter().append("line")

4 .attr("x1", x)

5 .attr("x2", x)

6 .attr("y1", 0)

7 .attr("y2", 120)

8 .style("stroke", "#ccc");

Positioning text above the reference lines reveals their value:

1 chart.selectAll(".rule")

2 .data(x.ticks(10))

3 .enter().append("text")

4 .attr("class", "rule")

5 .attr("x", x)

6 .attr("y", 0)

7 .attr("dy", -3)

8 .attr("text-anchor", "middle")

9 .text(String);

Note that the rule labels are assigned the class “rule”; this avoids a selector collision with the value labels on each bar. (Another way to disambiguate is to put reference labels in a separate gcontainer.) Lastly, add a single black line for the y-axis:

1 chart.append("line")

2 .attr("y1", 0)

3 .attr("y2", 120)

4 .style("stroke", "#000");

Et voilà!

05101520253035404815162342

This tutorial covered many of the core concepts in D3, including selections, dynamic properties, and scales. However, this only scratches the surface! Continue reading [part 2](http://mbostock.github.com/d3/tutorial/bar-2.html) to learn about transitions in dynamic visualizations. Or, explore the [examples gallery](http://mbostock.github.com/d3/ex/) to see more advanced techniques with D3.

# A Bar Chart, Part 2

The [previous part](http://mbostock.github.com/d3/tutorial/bar-1.html) of this tutorial covered the construction of a no-frills, static bar chart. This part will showcase some of the dynamic capabilities of D3, including transitions and data joins.

Say that, rather than a simple array of numbers, you want to visualize a [time series](http://en.wikipedia.org/wiki/Time_series)—a sequence of values sampled at regular time intervals. For example, say you run a website, and want to track how many visitors find your ideas intriguing? A bar chart could show the number of visitors that subscribe to your newsletter in realtime!

## Dynamic Data

Now typically, the subscription data would be downloaded to the client via an HTTP request. You can poll the server to refresh the latest data every minute, or use [web sockets](http://www.w3.org/TR/websockets/) to stream data incrementally, minimizing latency. To simplify this tutorial and focus on the task of visualization, we’ll construct a synthetic (i.e., fake) dataset by [random walk](http://en.wikipedia.org/wiki/Random_walk):

1 **var** t = 1297110663, *// start time (seconds since epoch)*

2 v = 70, *// start value (subscribers)*

3 data = d3.range(33).map(next); *// starting dataset*

4

5 **function** next() {

6 **return** {

7 time: ++t,

8 value: v = ~~Math.max(10, Math.min(90, v + 10 \* (Math.random() - .5)))

9 };

10 }

The exact mechanism of the random walk is unimportant, but you should understand the structure of the resulting data. Rather than a number, each data point is an object with time and value attributes:

1 {"time": 1297110663, "value": 56},

2 {"time": 1297110664, "value": 53},

3 {"time": 1297110665, "value": 58},

4 {"time": 1297110666, "value": 58},

Note that the values in the dataset are constrained to the domain [10, 90], which is convenient because it allows a fixed y-scale. This simplifies the implementation, as the old bars will not resize as new data arrives. You can use a dynamic scale, but keep in mind that rescaling old values while introducing new ones makes it harder for the user to perceive changes accurately. Also, you’ll need reference lines! Cushioning your scales to avoid sudden changes, or applying [hysteresis](http://en.wikipedia.org/wiki/Hysteresis) to delay changes, is recommended.

If you stream data from the server, you can redraw the bar chart whenever new data becomes available. In this case, we’ll cycle the data every 1.5 seconds:

1 setInterval(**function**() {

2 data.shift();

3 data.push(next());

4 redraw();

5 }, 1500);

The shift operation removes the first (oldest) element in the array, while the push appends after the last (newest) element. If you have a lot of data, a [circular buffer](http://en.wikipedia.org/wiki/Circular_buffer) will improve performance; with smaller data, the inefficiency of the shift operation is negligible and can be ignored. The redrawmethod is a function that you will define; we’ll get to that shortly.

## Dynamic Bars

For now, the next step is to construct two scales, based on our knowledge of the dataset and the desired chart size. To fix the maximum bar size to 80×20, construct two linear scales:

1 **var** w = 20,

2 h = 80;

3

4 **var** x = d3.scale.linear()

5 .domain([0, 1])

6 .range([0, w]);

7

8 **var** y = d3.scale.linear()

9 .domain([0, 100])

10 .rangeRound([0, h]);

The x-scale is a bit cheeky in that we’ve defined the domain as [0, 1], rather than the full time-domain of the dataset. That’s because we’ll assume (again, for simplicity) that the data is in chronological order and there are no missing data points. As such, we can use the index of the data to derive the x-position; x(i) is identical to w \* i. A more robust implementation would update the domain from the time attributes of the dataset whenever the data changes.

The y-scale uses rangeRound rather than range; the only difference is that the output values of the scale are rounded to the nearest integer to avoid antialiasing artifacts. If you prefer, you can instead use SVG’s [shape-rendering](http://www.w3.org/TR/SVG/painting.html#ShapeRenderingProperty) property. However, antialiasing is nice for smooth intermediate values during transition.

With the scales ready, construct the SVG container for the chart:

1 **var** chart = d3.select("body").append("svg")

2 .attr("class", "chart")

3 .attr("width", w \* data.length - 1)

4 .attr("height", h);

Add the initial bars:

1 chart.selectAll("rect")

2 .data(data)

3 .enter().append("rect")

4 .attr("x", **function**(d, i) { **return** x(i) - .5; })

5 .attr("y", **function**(d) { **return** h - y(d.value) - .5; })

6 .attr("width", w)

7 .attr("height", **function**(d) { **return** y(d.value); });

In SVG, rects are positioned relative to their top-left corner. For a vertical bar chart (also known as a column chart), the bars should be anchored by their bottom-left corner, so the “y” attribute flips the y-scale. Alternatively, you can use a transform to change the [coordinate system](http://www.w3.org/TR/SVG/coords.html). The .5 offset is to avoid antialiasing; the 1-pixel white stroke is centered on the given location, so a half-pixel offset will fill the pixel exactly. If you are not the Martha Stewart type, and don’t care for crisp edges, you may omit this step.

Add the y-axis last, so that it appears on top of the bars:

1 chart.append("line")

2 .attr("x1", 0)

3 .attr("x2", w \* data.length)

4 .attr("y1", h - .5)

5 .attr("y2", h - .5)

6 .style("stroke", "#000");

SVG draws shapes in the order they are specified, so to have the axis appear on top of the bars, the line must exist after the rects in the DOM. It is sometimes convenient to use g elements to group shapes into the desired z-order.

A little bit of CSS will set the bar colors:

1 **.chart** rect {

2 fill: steelblue;

3 stroke: white;

4 }

The code so far produces a static bar chart:

Now, what about that redraw function?

1 **function** redraw() {

2

3 *// Update…*

4 chart.selectAll("rect")

5 .data(data)

6 .transition()

7 .duration(1000)

8 .attr("y", **function**(d) { **return** h - y(d.value) - .5; })

9 .attr("height", **function**(d) { **return** y(d.value); });

10

11 }

Observe how the bars dance happily in response to changing data:

The redraw function is fairly trivial—reselect the rect elements, bind them to the new data, and then start a transition that updates the “y” and “height” attributes. No enter and exit selection is needed! Without a data join, the data are joined to nodes by index. As the length of the data array is fixed, the number of nodes never changes, and thus the enter and exit selections are always empty.

## Object Constancy

Yet, the above animation is poor because it lacks object constancy through the transition: it does not convey the changing data accurately. Rather than updating values in-place, the bars should slide to the left, so that each bar corresponds to the same point in time across the transition. Do this using a data join, to bind nodes to data by timestamp rather than index:

1 **function** redraw() {

2

3 **var** rect = chart.selectAll("rect")

4 .data(data, **function**(d) { **return** d.time; });

5

6 *// Enter…*

7 rect.enter().insert("rect", "line")

8 .attr("x", **function**(d, i) { **return** x(i) - .5; })

9 .attr("y", **function**(d) { **return** h - y(d.value) - .5; })

10 .attr("width", w)

11 .attr("height", **function**(d) { **return** y(d.value); });

12

13 *// Update…*

14 rect.transition()

15 .duration(1000)

16 .attr("x", **function**(d, i) { **return** x(i) - .5; });

17

18 *// Exit…*

19 rect.exit()

20 .remove();

21

22 }

With the new data join, we can no longer assume that the enter and exit selections are empty; instead, each contains exactly one bar upon redraw, as a new data point arrives and an old data point leaves. (If using real data, don’t assume regularity; multiple bars could enter and exit with each redraw.) So, the update is split to handle enter and exit separately. However, the update transition is actually simplified: we only transition the “x” attribute, as the “y” and “height” attributes do not change!

Note that operations on the entering or exiting selection do not affect the updating selection. Thus, the transition defined on rect on L14 above includes only the updating bars, not any of the entering bars that are appended on L7.

The bar chart now slides as desired, but the enter and exit are a bit clunky:

The above implementation enters new bars immediately, while old bars are removed immediately. A common alternative is to fade, but in this case the most intuitive transition is for new bars to enter from the right, and old bars to exit to the left. Enter and exit can have transitions, too, which you can use to offset the index i to the x-scale:

1 **function** redraw() {

2

3 **var** rect = chart.selectAll("rect")

4 .data(data, **function**(d) { **return** d.time; });

5

6 rect.enter().insert("rect", "line")

7 .attr("x", **function**(d, i) { **return** x(i + 1) - .5; })

8 .attr("y", **function**(d) { **return** h - y(d.value) - .5; })

9 .attr("width", w)

10 .attr("height", **function**(d) { **return** y(d.value); })

11 .transition()

12 .duration(1000)

13 .attr("x", **function**(d, i) { **return** x(i) - .5; });

14

15 rect.transition()

16 .duration(1000)

17 .attr("x", **function**(d, i) { **return** x(i) - .5; });

18

19 rect.exit().transition()

20 .duration(1000)

21 .attr("x", **function**(d, i) { **return** x(i - 1) - .5; })

22 .remove();

23

24 }

Note that the enter transition is staged; we initialize the values, and then start the transition. This is not needed with the exit transition because we’ll transition from the current state of the bar, regardless of value.

Et voilà!

This tutorial covered several core concepts in D3, including transitions, enter and exit, and data joins. However, this only scratches the surface! Explore the [examples gallery](http://mbostock.github.com/d3/ex/) to see more advanced techniques with D3.

**Proof of concept**

Say, you want to save your D3 application in a CouchDB database. This is just a proof of concept that this is possible. I use the »Focus + Context« diagram by Mike Bostock (<http://bl.ocks.org/1667367>) as an example.

Prerequisites:

* A running CouchDB (see <http://couchdb.apache.org/>; in my Windows installation CouchDb is started as a Windows service when Windows starts).
* An installation of couchapp (how to install couchapp, see <http://couchapp.org/page/index>).
* Your favorite editor.

The steps are quite simple.

* In your favorite workspace create a new folder and name it say **d3apps1**.
* Within this folder create a file named **.couchapprc**. This file will be used by couchapp to create the CouchDB database.
* Open this file in your text editor and add an empty object: **{}**. Nothing more is required for couchapp to create the database!
* In the **d3apps1** folder create a subfolder named **\_attachments**.
* Copy **index.html** and **sp500.csv** of the »Focus + Context« example into the subfolder **\_attachments**.

Now you should have the following folder/file structure:

d3apps1

\_attachments

index.html

sp500.csv

.couchappr

Open a command line window and navigate to your **d3apps1** folder. In the command line window issue the command (of course, couchapp must be on your path; otherwise you must supply the full path to couchapp.bat):

couchapp push d3apps1

That's it. Now in your browser, you can navigate to <http://127.0.0.1:5984/d3apps1/_design/d3apps1/index.html> and admire your d3 application.

Of course, this just shows that you can integrate and serve (!) D3 applications from a CouchDB without any middleware. There is still one external dependency: the d3 library, that is loaded from GitHub. This example doesn't use any features of CouchDB to store and deliver the data. This is just preliminary work for putting the data of my own applications into a CouchDB database. I thought this might be useful for others. How the d3 library and the data can be integrated into CouchDB is shown in [Integrating D3 with a CouchDB database 2](https://github.com/mbostock/d3/wiki/Integrating-D3-with-a-CouchDB-database-2).

**Integrating D3 with a CouchDB database 2**

[Part 1](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-1), [Part 3](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-3), [Part 4](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-4)

**Storing the D3 library in the CouchDB**

This one is easy. First, make a copy of your complete **d3apps1** folder and store it in your workspace using the name **d3apps2**. Then just move a copy of **d3.v2.min.js** into the **\_attachments** folder. Then it should look as follows:

d3apps2

\_attachments

d3.v2.min.js

index.html

sp500.csv

.couchapprc

Delete the references to the external d3 library files and insert a reference to the local **d3.v2.min.js** file. The beginning of your **index.html** file should look as follows (with or without the out-commented lines):

**<!DOCTYPE html>**

<meta charset="utf-8">

*<!--*

*<script src="http://mbostock.github.com/d3/d3.js?2.7.2"></script>*

*<script src="http://mbostock.github.com/d3/d3.csv.js?2.7.2"></script>*

*<script src="http://mbostock.github.com/d3/d3.time.js?2.7.2"></script>*

*-->*

<script src="d3.v2.min.js"></script>

From within your **d3apps2** folder, push your code into the CouchDB:

couchapp push d3apps2

Now D3 will be loaded from CouchDB.

**Storing the data in the CouchDB**

First, under **\_attachments** create a new file called **import.html**. Then add the following code to **import.html**:

<html>

<head>

<script src="/\_utils/script/jquery.js"></script>

<script src="/\_utils/script/jquery.couch.js"></script>

<script src="d3.v2.min.js"></script>

</head>

<body>

<script>

// Adapt the values to your needs

var file = "sp500.csv",

dbName = "d3apps2",

docId = "sp500",

source = "FocusAndContext";

d3.csv(file, function(data) {

var doc = {

\_id : docId,

souce : source,

data : data

// Add any fields you need

};

$.couch.db(dbName).saveDoc(doc, {

success:function (data) {

console.log("New doc created");

},

error:function (status) {

console.log("error:", status);

}

});

});

</script>

</body>

</html>

From within your **d3apps** folder, push your code into the CouchDB:

couchapp push d3apps2

Navigate to your import file:

http:**//**127.0.0.1:5984**/**d3apps2**/**\_design**/**d3apps2**/**import.html

That's it. The file is loaded and imports your data into CouchDB. You may control your import by navigating to

http:**//**127.0.0.1:5984**/**\_utils**/**database.html?d3apps2

There you find a new document named **"sp500"**. If you click on it, you can drill down to individual data.

**A Warning:** This import file is just a quick hack to get data into the CouchDB. There's no error checking. You can execute this import only once. If something goes wrong, you have to delete "sp500" manually using Futon, the rudimentary database manager of CouchDB, fix the error and try again. After the data has been imported successfully, it's no longer needed in this application. But don't yet delete it. You need it for the third part. And keep the **import.html** file, because you might want to import additional datasets into your CouchDB database, using the same import pattern.

How you can access the data in CouchDB from your D3 application, is shown in [Integrating D3 with a CouchDB database 3](https://github.com/mbostock/d3/wiki/Integrating-D3-with-a-CouchDB-database-3).

Last edited by drsm79, 5 months ago

**Integrating D3 with a CouchDB database 3**

[Part 1](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-1), [Part 2](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-2), [Part 4](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-4)

**Accessing data in a CouchDB database from a D3 application**

First, make a copy of your complete **d3apps2** folder and store it in your workspace using the name **d3apps3**. Your folder structure should look as follows:

d3apps3

\_attachments

d3.v2.min.js

import.html

index.html

sp500.csv

.couchappr**<**br**>**

To access the data in your **d3apps3** CouchDB, you need to change your **index.html** file. At the beginning you must insert two **jquery** scripts:

**<!DOCTYPE html>**

<meta charset="utf-8">

<script src="/\_utils/script/jquery.js"></script>

<script src="/\_utils/script/jquery.couch.js"></script>

<script src="d3.v2.min.js"></script>

<style>

The **d3.csv** function is no longer needed. Instead you retrieve the data from the database. Replace the function

d3.csv("sp500.csv", **function**(data) {

...

});

completely by following code:

**//** This **function** replaces the d3.csv **function**.

$.couch.db("d3apps3").openDoc("sp500", {

success : **function** (doc) {

var data = doc.data;

data.forEach(**function**(d) {

d.date = formatDate.parse(d.date);

d.price = **+**d.price;

});

x.domain(d3.extent(data.map(**function**(d) { **return** d.date; })));

y.domain([0, d3.max(data.map(**function**(d) { **return** d.price; }))]);

x2.domain(x.domain());

y2.domain(y.domain());

focus.append("path")

.data([data])

.attr("clip**-**path", "url(#clip)")

.attr("d", area);

focus.append("g")

.attr("class", "x axis")

.attr("transform", "translate(0," **+** height **+** ")")

.call(xAxis);

focus.append("g")

.attr("class", "y axis")

.call(yAxis);

context.append("path")

.data([data])

.attr("d", area2);

context.append("g")

.attr("class", "x axis")

.attr("transform", "translate(0," **+** height2 **+** ")")

.call(xAxis2);

context.append("g")

.attr("class", "x brush")

.call(brush)

.selectAll("rect")

.attr("y", **-**6)

.attr("height", height2 **+** 7);

},

error : **function** (status) {

console.log("Doc not found");

console.log("**\*\*\*** error:", status);

}

});

**Don't forget:** Change **dbName** in your **import.html** file to **d3apps3**!

From within your **d3apps3** folder, push your code into the CouchDB:

couchapp push d3apps3

Import your data into the CouchDB database by navigating to:

http:**//**127.0.0.1:5984**/**d3apps3**/**\_design**/**d3apps3**/**import.html

If the import has been successful, the **sp500.csv** file in the **\_attachments** folder is no longer needed. If you haven't done so, delete it now.

That's it. Now your data will be loaded from the CouchDB database.

On your local machine, navigate to

http:**//**127.0.0.1:5984**/**d3apps3**/**\_design**/**d3apps3**/**index.html

You can watch the result [here](https://rengel.iriscouch.com:6984/d3apps3/_design/d3apps3/index.html).

**Conclusion**

If you followed this tutorial, you now should have three CouchDB databases: **d3apps1**, **d3apps2**, **d3apps3**. They demonstrate different degrees of integration of an D3 application into a CouchDB database. The last one (\*\*d3apps3\*\*) has no dependencies on external files whatsoever (no libraries, no data files). Everything is contained in the database. Neither middleware nor a framework is required to access and display the data. The application and the data can be moved around, copied, replicated, deployed, etc. just by applying these operation to the database file. The only requirement is that the recipient has CouchDB running (which is available for all major operating systems).

**Roadmap**

This tutorial is the result of about a week of researching and figuring out, how d3 can be integrated with a database. Of course, this is a simple application, and the data is contained in a single document. The application has been taken verbatim from Mick Bostock's example, and no 'real' database features (records, selections, filters, etc.) are used. To show that this is not a toy application, some issues must be solved. It must be shown

* that one can work with many documents (records). Because I'm interested in (historical) timelines, my target at the moment is around 50,000 to 100,000 documents.
* that one can integrate richer content (pictures, videos, audio files, etc.). There are many applications that use CouchDB to deliver content of this type.
* that one can build a richer user interface. Again, there are many application that feature rich interfaces with CouchDB.

Stay tuned!

**Integrating D3 with a CouchDB database 4**

[Part 1](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-1), [Part 2](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-2), [Part 3](https://github.com/mbostock/d3/wiki/Integrating-d3-with-a-CouchDB-database-3)

Loading static data from a csv is fine, but really to use the power of CouchDB you'll want to be using views as the data source for your visualisation. This allows you to easily modify your dataset by adding/removing/editing documents.

**Changes to the couchapp**

First copy the app from a previous step over into d3apps4:

cp -r d3apps1 d3apps4

Next you'll need to make a view. Execute couchapp generate view d3apps4 pricetimeseries to create the view boiler plate. Remove the reduce function as we're not going to use that in this example (e.g. rm d3apps4/views/pricetimeseries/reduce.js). The reduce function is actually ideal for this kind of thing, you could have daily records being aggregated into single monthly records, but for simplicities sake we'll skip it for now.

The map function (d3apps4/views/pricetimeseries/map.js) should look like:

function(doc) {

if (doc.date && doc.price){

emit(doc.date, doc.price);

}

}

Next we need to change the javascript in d3apps4/\_attachments/index.html to use the json from the view instead of the static csv. There are lots of ways to do this (jquery.couch.js, sag.js to name a couple) but d3 has json support baked in, so lets use that directly. Once online, the view is available at \_view/pricetimeseries (in relation to index.html) so we can replace the d3.csv() function with:

d3.json("\_view/pricetimeseries", function(viewdata) {

// We just want rows from the view in the visualisation

data = viewdata["rows"];

data.forEach(function(d) {

// the key holds the date, in seconds

d.date = new Date(d.key);

d.price = +d.value;

});

// rest of the visalisation code

As you can see we've had to pick out the rows from the view and change the data parsing to take into account the different format of the view data (see below). You can see the full index.html in this [gist](https://gist.github.com/3690530).

Once you've made these changes push the couchapp up into the database with

couchapp push d3apps4

**Data import**

If you visit your app now your visualisation won't have any data in it. We need to import the csv into CouchDB documents so that the view can process them. The [gist](https://gist.github.com/3690530) contains a python script, import\_csv.py, to do just this. You'll need the excellent [requests](http://python-requests.org/) library installed to use it. Copy the import\_csv.py into d3apps4 and go into that directory. Run the script via python import\_csv.py, if everything works you should see<Response [201]>.

The script reads and uploads the csv data with each row being an individual document. It converts the data ("Sep 2012") into milliseconds during the import. This is to make sorting the date simpler - we could do that in the view or the d3 code but it's simpler to do it at acquisition time.

Once the data is uploaded go visit the application at <http://127.0.0.1:5984/d3apps4/_design/d3apps4/index.html>. You should see the visualisation, rendering data from the view.

Last edited by drsm79, 5 months ago

[**D3.js is Not a Graphing Library, Let's Design a Line Graph**](http://dealloc.me/2011/06/24/)

—Friday, June 24 2011

Working with graphing libraries can be tedious. Designing them can be downright frustrating. Each one of them slightly different, but most of them share two common flaws: a design-by-configuration and template design approach. A bar graph can be just a few bars with labels and tick marks…until it isn’t. Want to change the background color? New option. Want to change a bar color? New option. Want to hide the x-axis labels? New option. Want to highlight a specific bar with a different color? Good luck doing that.

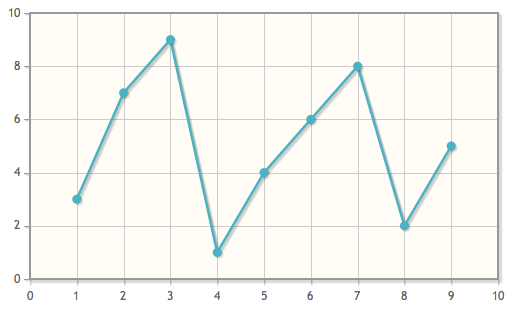
As long as you stay within the confines of the template, it’s simple, but, anytime you want customize a specific aspect of the original template, more configuration options are added to the library. **You should avoid “design by configuration.”**

I’m going to use [jqplot](http://www.jqplot.com/), a pretty popular graphing library, in the following examples. This isn’t an attempt to single out jqplot. Many graphing libraries use this template approach. I’ve [went down the rabbit hole](https://github.com/Caged/javascript-bits/tree/master/canvaslab)myself. With that out of the way, lets take a look at a simple line graph in jqplot.

**Before we begin:** **This is *not* a LOC comparison**. This is about writing code that makes sense. I’m using [CoffeeScript](http://jashkenas.github.com/coffee-script/) in most places and JavaScript where appropriate. If you’re not familiar with CoffeeScript, you can follow along with the [JavaScript](http://dealloc.me/demos/d3line/index.js) code. The complete CoffeeScript file can be[found here](http://dealloc.me/demos/d3line/index.coffee).

**var** plot1 **=** $.jqplot ('chart1', [[3,7,9,1,4,6,8,2,5]]);

This looks simple enough. First we include the proper renderers and then we create the default graph by specifying an element and passing in a nested array of data points. Once this renders, we end up with labels, grids, colors and shadows which were never specified. We’ll need to configure the graph to adjust or remove these items.



What happens when we need to tweak a few things? We end up with a crazy mix of nested hashes that become really hard to parse. And if you need more than one axis, things can really get crazy. In this jqplot example, “up to 9 y axes are supported”. I’ve never seen a graph with 9 y axes! The larger question is, why would you stop at 9?

1 **var** plot **=** $.jqplot ('chart2', [[3,7,9,1,4,6,8,2,5]], {

2 *// Give the plot a title.*

3 title**:** 'Plot With Options',

4 *// You can specify options for all axes on the plot at once with*

5 *// the axesDefaults object. Here, we're using a canvas renderer*

6 *// to draw the axis label which allows rotated text.*

7 axesDefaults**:** {

8 labelRenderer**:** $.jqplot.CanvasAxisLabelRenderer

9 },

10 *// An axes object holds options for all axes.*

11 *// Allowable axes are xaxis, x2axis, yaxis, y2axis, y3axis, ...*

12 *// Up to 9 y axes are supported.*

13 axes**:** {

14 *// options for each axis are specified in seperate option objects.*

15 xaxis**:** {

16 label**:** "X Axis",

17 *// Turn off "padding". This will allow data point to lie on the*

18 *// edges of the grid. Default padding is 1.2 and will keep all*

19 *// points inside the bounds of the grid.*

20 pad**:** 0

21 },

22 yaxis**:** {

23 label**:** "Y Axis"

24 }

25 }

26 });

**D3: Layers, Shapes, Text and Scales**

[](http://www.flickr.com/photos/windkoh/2743959440/)

What about [D3](http://mbostock.github.com/d3/)? D3 is a relatively new visualization library. Created by the extremely talented and proactive, [mbostock](https://twitter.com/" \l "!/mbostock). D3 can do many things (that we’ll get into later), but for now, let’s define what a “graph” is.

At its core, a graph is just layers of paths, primitives, color and text—something SVG is perfectly suited for. Lets take a look at a simple SVG line graph in d3.

1 data = [3,7,9,1,4,6,8,2,5]

2 w = 700

3 h = 300

4 max = d3.max(data)

5

6 *# Scales*

7 x = d3.scale.linear().domain([0, data.length **-** 1]).range [0, w]

8 y = d3.scale.linear().domain([0, max]).range [h, 0]

9

10 *# Base vis layer*

11 vis = d3.select('#chart')

12 .append('svg:svg')

13 .attr('width', w)

14 .attr('height', h)

15

16 *# Add path layer*

17 vis.selectAll('path.line')

18 .data([data])

19 .enter().append("svg:path")

20 .attr("d", d3.svg.line()

21 .x(**(d,i) ->** x(i))

22 .y(y))

[See example in action →](http://dealloc.me/demos/d3line/#1)

**Note:** Many methods in d3 (data, attr, x, y, etc.) will evaluate the first function passed to it using the current data point. For example: .y(function(d) { return d; }) is the same thing as .y(yfunc) Also, every attr call is part of the [SVG spec](http://www.w3.org/TR/SVG/). There are no “magic” attr values.

Lines 7 and 8 are where the magic happens. **These two lines return functions** that accept a single argument represented in the input domain. The x scale above is saying: *“I want a linear scale that represents data between 0 and 8 and I want the values returned to fit in the pixel range of 0 and the width (w).”* Lets see that expanded:

x = d3.scale.linear().domain([0, 8]).range [0, 700]

*# If given a value in the domain, should always return a number between 0 and 700*

console.log x(8), x(4)

Lines 11-14 are pretty straight forward. They are responsible for appending the svg element to the DOM and setting up some basic dimensions for our document.

Lets look at lines 17-22, which can be thought of as the first layer in our graph. We supply the data here and generate [the resulting path data](http://www.w3.org/TR/SVG/paths.html#PathData). If we were drawing more than one line, our data might look like.data([data1, data2]). In this example, we’re only plotting one dataset so we have an array with a single array entry. Moving along, we then get to the oddly named enter method. I like to think of enter asfind\_or\_create, but be sure to [check out the docs](https://github.com/mbostock/d3/wiki/Selections#enter) for an expanded explanation. Finally, we append the svg path element and using a [path data generator](https://github.com/mbostock/d3/wiki/SVG-Shapes#line) to generate the path data. Lets take a look at the path data generator:

d3.svg.line().x(**(d,i) ->** x(i)).y(y)

We’re using the scale functions we defined earlier. The i argument passed through to svg.line’s x method is the current index. So x and y will be called for each piece of data in our dataset passing the current piece of data as the first argument and the current index as the second argument. After the code above is executed, we end up with a pretty simple svg.

<svg width="700" height="300"><path d="M0,200L87.5,66.66666666666669L175,0L262.5,266.6666666666667L350,166.66666666666669L437.5,100L525,33.33333333333337L612.5,233.33333333333334L700,133.33333333333331"></path></svg>

**Note:** If you find yourself writing for loops with d3, you could be doing it wrong. Very rarely will you need to do this.

**CSS: It Just Works**

Many existing libraries are implementing using [Canvas](http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html). It’s not possible to style objects drawn to a canvaselement using CSS, but that hasn’t stopped [people from trying](http://tjholowaychuk.com/post/6339741902/styling-canvas-drawings-with-css). SVG on the other hand? SVG has excellent [CSS support](http://www.w3.org/TR/SVG/styling.html). Lets style the path we’ve drawn above:

path {

stroke**:** #c00;

stroke**-width:** 3px;

}

That’s it! We can put this in the same stylesheets we’re already using to style DOM elements. If a designer comes along and wants to tweak a few things, there is no need to go digging around in JavaScript.

**Caveat:** If you want to open SVG renders in a program such as Adobe Illustrator and retain styling, you’ll need to use attributes instead of externally defined CSS.

**Grouping: Labels and Grids**

When you have elements that share a similar position, it’s really handy to use the <g> element. All children of a parent <g> element will share the same relative position. For instance, a tick mark and its accompanying label should be positioned pretty close to each other. We don’t want to run this computation more than we have to, so we use a group.

1 *# Add tick groups*

2 ticks = vis.selectAll('.tick')

3 .data(y.ticks(7))

4 .enter().append('svg:g')

5 .attr('transform', **(d) ->** "translate(0, #{y(d)})")

6 .attr('class', 'tick')

7

8 *# Add y axis tick marks*

9 ticks.append('svg:line')

10 .attr('y1', 0)

11 .attr('y2', 0)

12 .attr('x1', 0)

13 .attr('x2', w)

14

15 *# Add y axis tick labels*

16 ticks.append('svg:text')

17 .text(**(d) ->** d)

18 .attr('text-anchor', 'end')

19 .attr('dy', 2)

20 .attr('dx', **-**4)

[See step 2 in action →](http://dealloc.me/demos/d3line#2)

We’ve already covered most of what is going on in the code above. The interesting part can be found on line 3: .data(y.ticks(7)). Remember those linear scales we defined earlier? ticks is [a method](https://github.com/mbostock/d3/wiki/Quantitative-Scales#linear_ticks) which will return a *“uniformly spaced, human readable values guaranteed to be within the extent of the input domain.”* In layman’s terms, if the linear range is 0-10, y.ticks(5) will attempt to return 5 numbers, evenly spaced between 0-10. Try it yourself in the console:

y **=** d3.scale.linear().domain([0, 10])

y.ticks(10) *// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*

y.ticks(5) *// [0, 2, 4, 6, 8, 10]*

y **=** d3.scale.linear().domain([0, 500])

y.ticks(10) *// [0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500]*

y.ticks(5) *// [0, 100, 200, 300, 400, 500]*

Moving on, you’ll notice the transform attribute. This is the [svg transform attribute](http://www.w3.org/TR/SVG/coords.html" \l "SVGGlobalTransformAttribute). In this particular case, we’re using our y scale to position the group. Can you guess what d is here? It’s each entry in the array of tick marks we talked about above.

Further down, you’ll notice we don’t have to position the svg:line elements because they are being positioned relative their parent group. Finally, we add the actual svg:text label, and use [dx and dy](http://www.w3.org/TR/SVG/text.html#TSpanElementDXAttribute) to make small tweaks to the text relative to the current x and y locations.

**Selectively Highlighting Data**

Sometimes we want to make a particular datapoint stand out. Whether we use color, visibility, or another method, it can be very difficult or impossible with template-based graphing libraries. D3 makes this very easy given most methods allow you to pass along a function to evaluate with the current piece of data. For example, I chose to highlight only the max datapoint when plotting a 30 day trend over at [Portland Crime](http://portlandcrime.com/offenses/aggravated-assault).

Let’s add circles to each data point along our line and highlight the largest data value using color and size.

1 *# Add point circles*

2 vis.selectAll('.point')

3 .data(data)

4 .enter().append("svg:circle")

5 .attr("class", **(d, i) ->** **if** d **==** max **then** 'point max' **else** 'point')

6 .attr("cx", **(d, i) ->** x(i))

7 .attr("cy", **(d) ->** y(d))

8 .attr("r", **(d, i) ->** **if** d **==** max **then** 6 **else** 4)

[See step 3 in action →](http://dealloc.me/demos/d3line#3)

Lines 5 and 8 are what we’re looking at. For every data point passed in, we check if it is equal to the max, if it is, we add a new class and increase its radius (r) value.

**Events: Making it Interactive**

Interactivity is a great way to liven up a graph. In our case, we’re going to tweak the radius of the points when someone hovers or leaves one. You could also use the same technique for adding tooltips or updating another DOM element. It doesn’t have to be inside the SVG document. Let’s modify the code above.

1 *# Add point circles*

2 vis.selectAll('.point')

3 .data(data)

4 .enter().append("svg:circle")

5 .attr("class", **(d, i) ->** **if** d **==** max **then** 'point max' **else** 'point')

6 .attr("r", **(d, i) ->** **if** d **==** max **then** 6 **else** 4)

7 .attr("cx", **(d, i) ->** x(i))

8 .attr("cy", **(d) ->** y(d))

9 .**on**('mouseover', **->** d3.select(**this**).attr('r', 8))

10 .**on**('mouseout', **->** d3.select(**this**).attr('r', 4))

11 .**on**('click', **(d, i) ->** console.log d, i)

[See final graph →](http://dealloc.me/demos/d3line)

**CSS & Attribute Precedence:** Be careful! If you define a style such as stroke-width in a stylesheet, it will take precedence over the *attribute* stroke-width. If you’re attempting to overwrite a style of an element, use the [.style](https://github.com/mbostock/d3/wiki/Selections#style) method.

Lines 9-11 are what we’re looking at. The on method allows us to attach an event to a node. When the event is triggered, a callback is fired, passing in the current piece of data and an index. All we’re doing is increasing and decreasing the radius of the current point.

Inside most methods, this will refer to the current element in the SVG document. It isn’t *“prewrapped”* withd3 selection methods, so you’ll need to run d3.select(this) if you intend to use them.

**Lesson for the Reader**

1. Those two 0s at the bottom left look pretty cramped. I think both of them can be implied. Can you get rid of them while keeping the other numbers intact?
2. The line looks a little rigid. Try experimenting with the [interpolation](https://github.com/mbostock/d3/wiki/SVG-Shapes#line_interpolate) and [tension](https://github.com/mbostock/d3/wiki/SVG-Shapes#line_tension).

**Final Thoughts**

D3 isn’t a graphing library. It’s so much more. It’s [Cartograms](http://mbostock.github.com/d3/ex/cartogram.html), [Choropleth Maps](http://mbostock.github.com/d3/ex/choropleth.html), [Chord Diagrams](http://mbostock.github.com/d3/ex/chord.html) and a[whole bunch of other stuff](https://github.com/mbostock/d3/tree/master/examples). On a camping trip, it’s the tool you never leave behind. If you’re fishing, it’s your tackle box. If you’re designing a line graph for the web? It’s your graphing library.

# Towards Reusable Charts

I’d like to propose a convention for encapsulating reusable charts in [D3](http://d3js.org/). Wait for it…

function chart() {

// generate chart here

}

You could infer the dimensions from the containing element, but most charts require some configuration.

A function; the standard unit of code reuse!

## [#](http://bost.ocks.org/mike/chart/" \l "configuration)Configuration

I jest; not any function will do. In truth we need a *configurable* function, since most charts require customization of their appearance or behavior. For example, you may need to specify the width and height, or the color palette. A simple method of configuration is passing arguments:

function chart(width, height) {

// generate chart here, using `width` and `height`

}

Yet this is cumbersome for the caller: they must store the chart’s various arguments, and pass them whenever an update is needed. A simple function is insufficient for highly-configurable charts. You could try a configuration object instead, as is done by many charting libraries:

function chart(config) {

// generate chart here, using `config.width` and `config.height`

}

However, the caller must then manage both the chart function (assuming you have multiple types of charts to pick from) and the configuration object. To bind the chart configuration to the chart function, we need a [closure](http://jibbering.com/faq/notes/closures/):

A conventional object-oriented approach as Chart.​proto­type.​render would also work, but then you must manage the thiscontext when calling the function.

function chart(config) {

return function() {

// generate chart here, using `config.width` and `config.height`

};

}

Now, the caller need merely say:

var myChart = chart({width: 720, height: 80});

And subsequently, myChart() to update. Simple!

## [#](http://bost.ocks.org/mike/chart/" \l "reconfiguration)Reconfiguration

But what if you want to change the configuration after construction? Or if you want to inspect the configuration of an existing chart? The configuration object is trapped by the closure and inaccessible to the outside world. Fortunately, JavaScript [functions are objects](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function), so we can store configuration properties on the function itself!

var myChart = chart();

myChart.width = 720;

myChart.height = 80;

The chart implementation changes slightly so that it can reference its configuration:

The inner function (my) can be named whatever you like; the name is only visible internally. You can even use the namechart, which would mask the outer function!

function chart() {

return function my() {

// generate chart here, using `my.width` and `my.height`

};

}

With a little bit of syntactic sugar, we can replace raw properties with getter-setter methods that allow [method chaining](http://en.wikipedia.org/wiki/Method_chaining). This gives the caller a more elegant way of constructing charts, and also allows the chart to manage side-effects when a configuration parameter changes. The chart may also provide default configuration values. Here we create a new chart and set two properties:

var myChart = chart().width(720).height(80);

Modifying an existing chart is similarly easy:

myChart.height(500);

As is inspecting it:

myChart.height(); // 500

Internally, the chart implementation becomes slightly more complex to support getter-setter methods, but convenience for the user merits additional developer effort! (And besides, this pattern becomes natural after you’ve used it for a while.)

function chart() {

var width = 720, // default width

height = 80; // default height

function my() {

// generate chart here, using `width` and `height`

}

my.width = function(value) {

if (!arguments.length) return width;

width = value;

return my;

};

my.height = function(value) {

if (!arguments.length) return height;

height = value;

return my;

};

return my;

}

To sum up: implement charts as **closures with getter-setter methods**. Conveniently, this is the same pattern used by D3’s other reusable objects, including [scales](https://github.com/mbostock/d3/wiki/Scales), [layouts](https://github.com/mbostock/d3/wiki/Layouts), [shapes](https://github.com/mbostock/d3/wiki/SVG-Shapes), [axes](https://github.com/mbostock/d3/wiki/SVG-Axes), *etc.*

## [#](http://bost.ocks.org/mike/chart/" \l "implementation)Implementation

The chart can now be configured, but two essential ingredients are still missing: the DOM element into which to render the chart (such as a particular div or document.body), and the data to display. These could be considered configuration, but D3 provides a more natural representation for data and elements: the **selection**.

By taking a selection as input, charts have greater flexibility. For example, you can render a chart into multiple elements simultaneously, or easily move a chart between elements without explicitly unbinding and rebinding. You can control exactly when and how the chart gets updated when data or configuration changes (for example, using a transition rather than an instantaneous update). In effect, the chart becomes a rubber stamp for rendering data visually.

The simplest way of invoking our chart function on a selection, then, is to pass the selection as an argument:

myChart(selection);

From the API reference: [call] invokes the specified function once, passing in the current selection… The call operator is identical to invoking a function by hand; but it makes it easier to use method chaining.

Or equivalently, using [selection.call](https://github.com/mbostock/d3/wiki/Selections" \l "wiki-call):

selection.call(myChart);

Internally, a call-based chart implementation looks something like this:

You could also design your chart function to work directly with[selection.each](https://github.com/mbostock/d3/wiki/Selections#wiki-each), but selection.call is more general and has precedent with the brush and axis components.

function my(selection) {

selection.each(function(d, i) {

// generate chart here; `d` is the data and `this` is the element

});

}

## [#](http://bost.ocks.org/mike/chart/" \l "examples)Examples

To make this proposal concrete, consider a simple yet ubiquitous use case: time-series visualization. A *time series* is a variable that changes over time. We can visualize this as an area chart, where the *x*- and *y*-axes respectively encode time and value as position:

20002001200220032004200520062007200820092010

To hold the chart, this page has an initially-empty p (paragraph) element:

<p id="example">

For data, there’s an external [CSV](http://en.wikipedia.org/wiki/Comma-separated_values) file ([sp500.csv](http://bost.ocks.org/mike/chart/sp500.csv)), the first few lines of which looks like this:

date,price

Jan 2000,1394.46

Feb 2000,1366.42

Mar 2000,1498.58

Apr 2000,1452.43

May 2000,1420.6

Jun 2000,1454.6

Jul 2000,1430.83

Aug 2000,1517.68

To create the chart, we first call timeSeriesChart to make a new chart instance, defining accessors for the *x* (date, a [Date](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date) object) and *y* (price, a number) dimensions for our data:

var chart = timeSeriesChart()

.x(function(d) { return formatDate.parse(d.date); })

.y(function(d) { return +d.price; });

var formatDate = d3.time.format("%b %Y");

The price is coerced to a number using the unary + operator; JavaScript is weakly-typed, so it’s a good idea to convert the loaded data from string to number. The date requires a [d3.time.format](https://github.com/mbostock/d3/wiki/Time-Formatting) for parsing; %b refers to the abbreviated month name, and %Y refers to the four-digit year. We might want to configure other options as well, but in this simple example the default width, height and margin are suitable, and no other configuration is needed.

Lastly, we load the data via [d3.csv](https://github.com/mbostock/d3/wiki/CSV). When available, we bind the data to the empty paragraph element, and use selection.call to render the chart!

The chart implementation source is in [time-series-chart.js](http://bost.ocks.org/mike/chart/time-series-chart.js), if you want to see how the chart works internally.

d3.csv("sp500.csv", function(data) {

d3.select("#example")

.datum(data)

.call(chart);

});

See the [axis component](http://bl.ocks.org/1166403) and[brush component](http://bl.ocks.org/1667367) for examples of chart components supporting interaction and animation automatically.

## [#](http://bost.ocks.org/mike/chart/" \l "further-considerations)Further Considerations

We now have a **strawman convention** for reusable visualization components. Yet there is far more to cover as to what should be considered configuration or even a chart. Is a traditional chart typology the best choice? Consider drawing inspiration from the [Grammar of Graphics](http://www.amazon.com/Grammar-Graphics-Leland-Wilkinson/dp/0387987746) (see[Polychart](http://polychart.com/) and [ggplot2](http://had.co.nz/ggplot2/)); there are more modular units for composition. Even with traditional chart types, should you expose the underlying scales and axes, or encapsulate them with chart-specific representations? Should your chart support interaction and animation automatically? Should the user be able to reach into your chart and tweak some aspect of its behavior? All of these are possible using the proposed convention, so have at it!