Comp 388/424 - Client-side Web Design - notes

Spring Semester 2016 - Week 13

Dr Nick Hayward

Contents

Data visualisation - D3

Selections - single element

- select a single element within our page

```
d3.select("p");
```

- now select the first <p> element on the page, and then allow us to modify as necessary
  - *eg; we could simply add some text to this element*

```
d3.select("p")
.text("Hello World");
```

- selection could be a generic element, such as <p>
  - *or a specific element defined by targeting its ID*
- use additional modifier functions, such as `attr`, to perform a given modification on the selected element

```
//set an attribute for the selected element
d3.select("p").attr("foo");
//get the attribute for the selected element
d3.select("p").attr("foo");
```

- also add or remove classes on the selected element

```
//test selected element for specified class
d3.select("p").classed("foo")
//add a class to the selected element
d3.select("p").classed("goo", true);
//remove the specified class from the selected element
d3.select("p").classed("goo", function(){ return false; });
```

Data visualisation - D3

Selections - multiple elements

- also select all of the specified elements using D3

```
d3.selectAll("p")
.attr("class", "para");
```

- use and implement multiple element selection
  - *same as single selection pattern*
- also use the same modifier functions
- allows us to modify each element's attributes, style, class...

Data visualisation - D3

Selections - iterating through a selection

- D3 provides us with a selection iteration API
  - *allows us to iterate through each selection*
  - *then modify each selection relative to its position*
  - *very similar to the way we normally loop through data*

```
d3.selectAll("p")
.attr("class", "para")
.each(function (d, i) {
    d3.select(this).append("h1").text(i);
});
```

- D3 selections are essentially like arrays with some enhancements
  - *use the iterative nature of Selection API*

```
d3.selectAll('p')
.attr("class", "para2")
.text(function(d, i) {
    return i;
});
```

Data visualisation - D3

Selections - performing sub-selection

- for selections - often necessary to perform specific scope requests
  - *eg: selecting all <p> elements for a given element*

```
//direct css selector (selector level-3 combinators)
d3.select("div > p")
    .attr("class", "para");

//d3 style scope selection
d3.select("div")
    .selectAll("p")
    .attr("class", "para");
```

- both examples produce the same effect and output, but use very different selection techniques
  - *first example uses the CSS3, level-3, selectors*
  - *`div > p` is known as combinators in CSS syntax*

Data visualisation - D3

Selections - combinators

Example combinators..

1. descendant combinator

- uses the pattern of `selector selector` - describing loose parent-child relationship
- loose due to possible relationships - parent-child, parent-grandchild...

```
d3.select("div p");
```

- select the <p> element as a child of the parent element
  - *relationship can be generational*

2. child combinator

- uses same style of syntax, `selector > selector`
- able to describe a more restrictive **parent-child** relationship between two elements

```
d3.select("div > p");
```

- finds <p> element if it is a direct child to the element

Selections - D3 sub-selection

- sub-selection using D3's built-in selection of child elements
- a simple option to select an element, then chain another selection to get the child element
- this type of chained selection defines a scoped selection within D3
  - *eg: selecting a <p> element nested within our selected element*
  - *each selection is, effectively, independent*
- D3 API built around the inherent concept of function chaining
  - *can almost be considered a Domain Specific Language for dynamically building HTML/SVG elements*
- a benefit of chaining = easy to produce concise, readable code

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

body.append("div")
    .attr("id", "div1")
  .append("p")
    .attr("class", "para")
  .append("h5")
    .text("this is a paragraph heading...");
```

Data visualisation - D3

Data Intro - page elements

- generation of new DOM elements normally fits
  - *either circles, rectangles, or some other visual form that represents the data*
- D3 can also create generic structural elements in HTML, such as a <p>
  - *eg: we can append a standard p element to our new page*

```
d3.select("body").append("p").text("sample text...");
```

- used D3 to select `body` element, then append a new <p> element with text "new paragraph"
- D3 supports *chain syntax*
  - *allowed us to `select`, `append`, and add `text` in one statement*

Data visualisation - D3

Data Intro - page elements

```
d3.select("body").append("p").text("sample text...");
```

- `d3`
  - *references the D3 object, access its built-in methods*
- `.select("body")`
  - *accepts a CSS selector, returns first instance of the matched selector in the document's DOM*
  - *`.selectAll()`*
  - ***NB:** this method is a variant of the single `select()`*
  - *returns all of the matched CSS selectors in the DOM*
- `.append("p")`
  - *creates specified new DOM element*
  - *appends it to the end of the defined select CSS selector*
- `.text("new paragraph")`
  - *takes defined string, "new paragraph"*
  - *adds it to the newly created <p> DOM element*

Data visualisation - D3

Binding data - making a selection

- choose a selector within our document
  - *eg: we could select all of the paragraphs in our document*

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

- if the element we require does not yet exist
  - *need to use the method `enter()`*

```
d3.select("body").selectAll("p").data(dataset).enter().append("p").text("new paragraph");
```

- we get new paragraphs that match total number of values currently available in the **dataset**
  - *akin to looping through an array*
  - *outputting a new paragraph for each value in the array*
- create new, data-bound elements using `enter()`
  - *method checks the current DOM selection, and the data being assigned to it*
- if more data values than matching DOM elements
  - *`enter()` creates a new placeholder element for the data value*
  - *then passes this placeholder on to the next step in the chain, eg: `append()`*
- data from dataset also assigned to new paragraphs
- **NB:** when D3 binds data to a DOM element, it does not exist in the DOM itself
  - *it does exist in the memory*

Data visualisation - D3

Binding data - using the data

- change our last code example as follows,

```
d3.select("body").selectAll("p").data(dataset).enter().append("p").text(function(d) { retu
```

- then load our HTML, we'll now see dataset values output instead of fixed text
- anytime in the chain after calling the `data()` method
  - *we can then access the current data using d*
- also bind other things to elements with D3, eg: CSS selectors, styles...

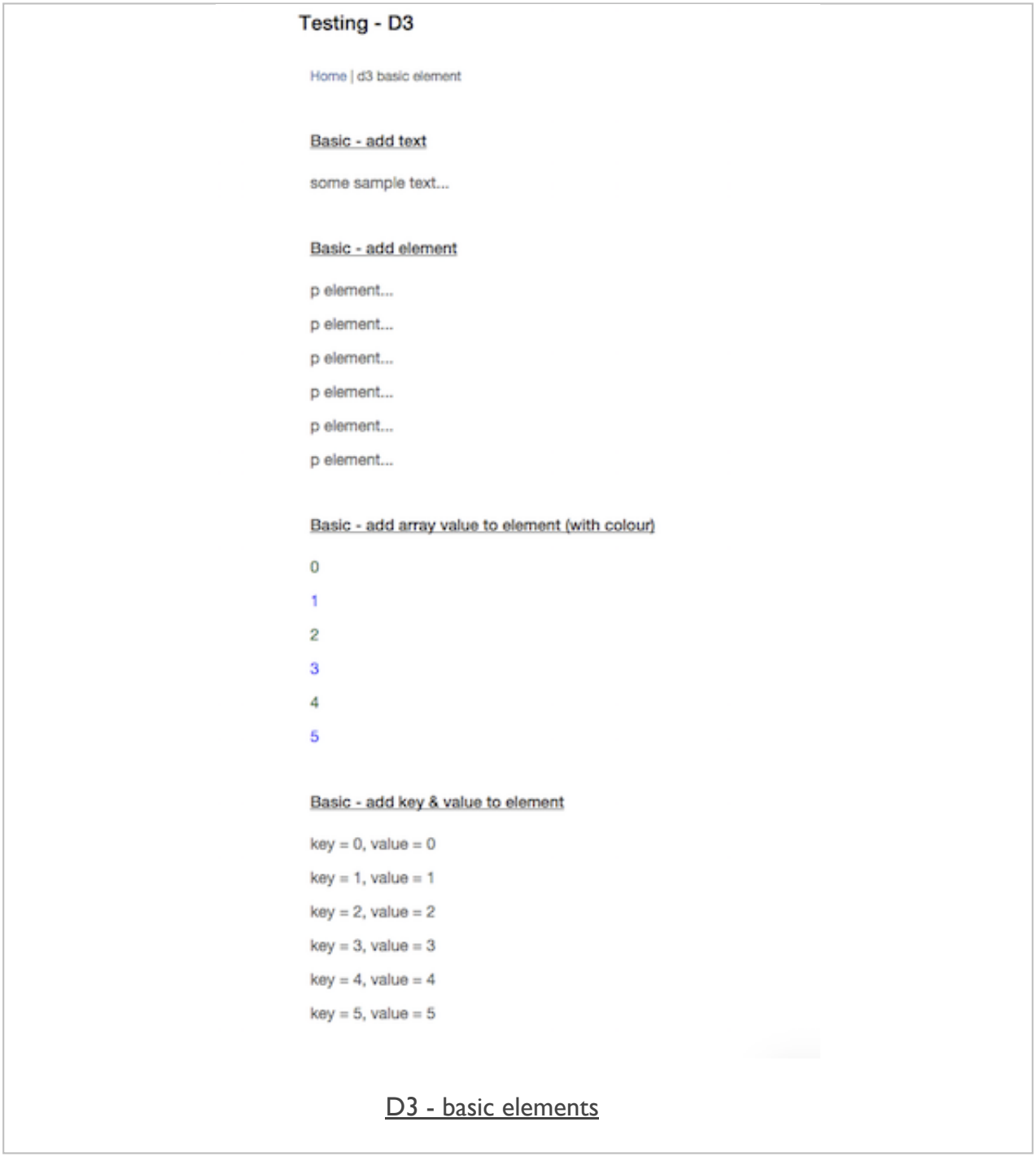
```
.style("color", "blue");
```

- chain the above to the end of our existing code
  - *now bind an additional css style attribute to each <p> element*
  - *turning the font colour blue*
- extend code to include a conditional statement that checks the value of the data
  - *eg: simplistic striped colour option*

```
.style("color", function(d) {
if (d % 2 == 0) {
return "green";
} else {
 return "blue";
}
});
```

- DEMO - D3 basic elements

Image - D3 Basic Elements



D3 - basic elements

Data visualisation - D3

Drawing - intro - part 1

1. drawing divs

- one of the easiest ways to draw a rectangle, for example, is with a HTML `<div>`
- an easy way to start drawing a bar chart for our stats
- start with standard HTML elements, then consider more powerful option of drawing with SVG
- semantically incorrect, we could use `<div>` to output bars for a bar chart
  - *use of an empty `<div>` for purely visual effect*
- using D3, add a class to an empty element using `selection.attr()` method

2. setting attributes

- `attr()` is used to set an HTML attribute and its value on an element
- After selecting the required element in the DOM
  - *assign an attributes as follows*

```
.attr("class", "barchart")
```

Data visualisation - D3

Drawing - intro - part 2

- use D3 to draw a set of bars in divs as follows

```
var dataset = [ 1, 2, 3, 4, 5 ];

d3.select("body").selectAll("div")
    .data(dataset)
    .enter()
    .append("div")
    .attr("class", "bar");
```

- above sample outputs the values from our dataset with no space between them
  - *effectively as a bar chart of equal height*
- modify the height of each representative bar
  - *by setting height of each bar as a function of its corresponding data value*
  - *eg: append the following to our example chain*

```
.style("height", function(d) {
    return d + "px";
});
```

- make each bar in our chart more clearly defined by modifying `style`

```
.style("height", function(d) {
    var barHeight = d * 3;
    return barHeight + "px";
});
```

Data visualisation - D3

Drawing - intro - part 3

1. drawing SVGs

- properties of SVG elements are specified as **attributes**
- represented as property/value pairs within each element tag

```
<element property="value">...</element>
```

- SVG elements exist in the DOM
  - *we can still use D3 methods `append()` and `attr()`*
  - *create new HTML elements and set their attributes*

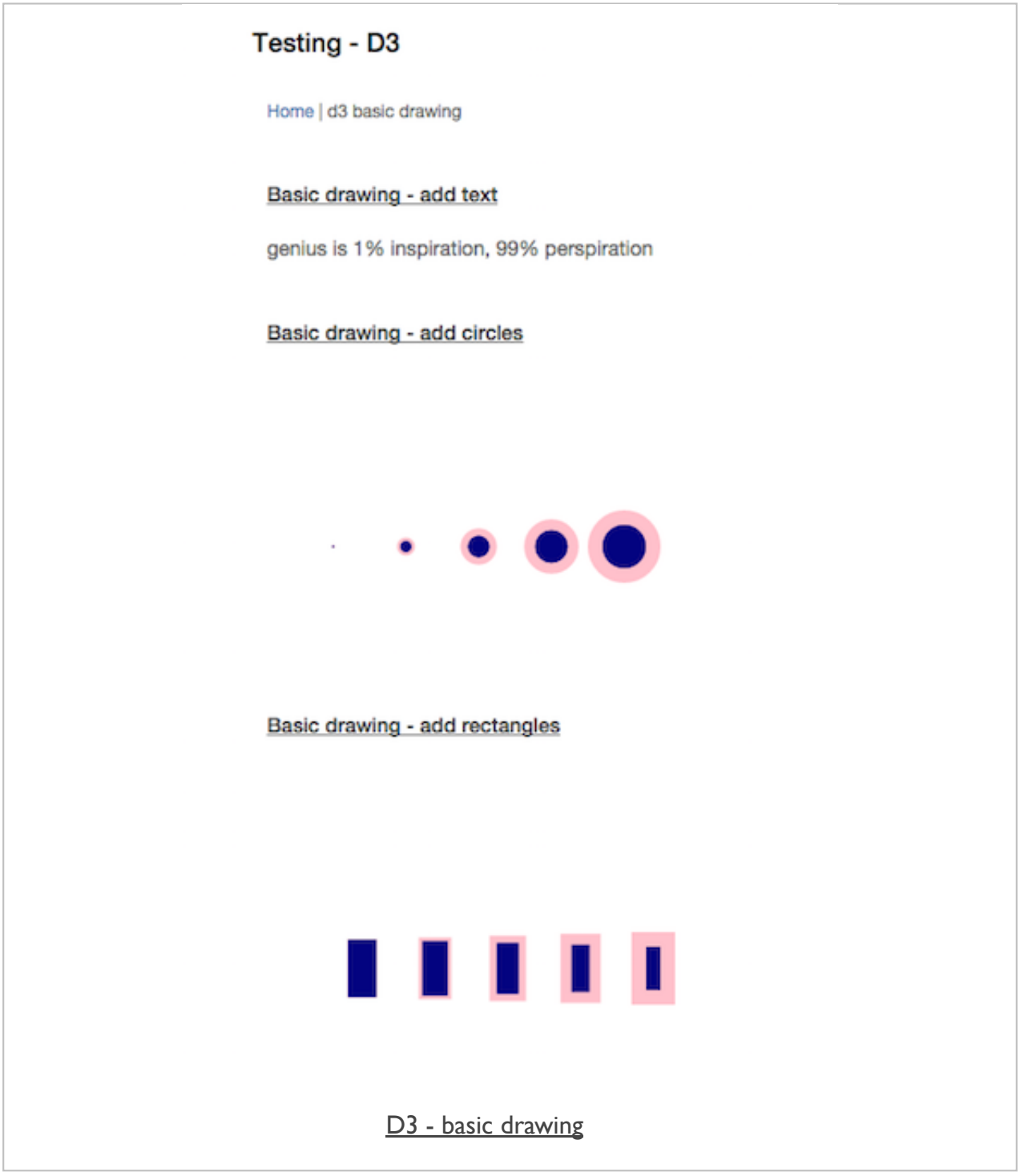
2. create SVG

- need to create an element for our SVG
- allows us to draw and output all of our required shapes

```
d3.select("body").append("svg");
```

- variable effectively works as a reference
  - *points to the newly created SVG object*
  - *allows us to use this reference to access this element in the DOM*
- DEMO - Drawing with SVG

Image - D3 Basic Drawing



D3 - basic drawing

Data visualisation - D3

Drawing - SVG barchart - part 1

- create a new barchart using SVG, need to set the required size for our SVG output

```
//width & height
var w = 750;
var h = 200;
```

- then use D3 to create an empty SVG element, and add it to the DOM

```
var svg = d3.select("body")
    .append("svg")
    .attr("width", w)
    .attr("height", h);
```

- instead of creating DIVs as before, we generate *rects* and add them to the *svg* element.

```
svg.selectAll("rect")
    .data(dataset)
    .enter()
    .append("rect")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width", 10)
    .attr("height", 50);
```

Drawing - SVG barchart - part 2

- this code selects all of the `rect` elements within `svg`
- initially none, D3 still needs to select them before creating them
- `data()` then checks the number of values in the specified dataset
  - *hands those values to the `enter` method for processing*
- `enter` method then creates a placeholder
  - *for each data value without a corresponding `rect`*
  - *also appends a rectangle to the DOM for each data value*
- then use `attr` method to set `x, y, width, height` values for each rectangle
- still only outputs a single bar due to an overlap issue
- need to amend our code to handle the width of each bar
  - *implement flexible, dynamic coordinates to fit available SVG width and height*
  - *visualisation scales appropriately with the supplied data*

```
.attr("x", function(d, i) {
    return i * (w / dataset.length);
})
```

Data visualisation - D3

Drawing - SVG barchart - part 3

- now linked the `x` value directly to the width of the SVG `w`
  - *and the number of values in the dataset, `dataset.length`*
  - *the bars will be evenly spaced regardless of the number of values*
- if we have a large number of data values
  - *bars still look like one horizontal bar*
  - *unless there is sufficient width for parent SVG and space between each bar*
- try to solve this as well by setting the bar width to be proportional
  - *narrower for more data, wider for less data*

```
var w = 750;
var h = 200;
var barPadding = 1;
```

- now set each bar's width
  - *as a fraction of the SVG width and number of data points, minus our padding value*

```
.attr("width", w / dataset.length - barPadding)
```

- our bar widths and x positions scale correctly regardless of data values

Data visualisation - D3

Drawing - SVG barchart - part 4

- encode our data as the *height* of each bar

```
.attr("height", function(d) {
    return d * 4;
});
```

- our bar chart will size correctly, albeit from the top down
  - *due to the nature of SVG*
  - *SVG adheres to a top left pattern for rendering shapes*
- to correct this issue
  - *need to calculate the top position of our bars relative to the SVG*
- top of each bar expressed as a relationship
  - *between the height of the SVG and the corresponding data value*

```
.attr("y", function(d) {
    //height minus data value
    return h - d;
})
```

- bar chart will now display correctly from the bottom upwards
- DEMO - Drawing with SVG - barcharts

Image - D3 Barcharts

D3 - drawing barcharts

Data visualisation - D3

Drawing - SVG barchart - part 5

1. add some colour

- adding a colour per bar simply a matter of setting an attribute for the fill colour

```
.attr("fill", "blue");
```

- set many colours using the data itself to determine the colour

```
.attr("fill", function(d) {
    return "rgb(0, 0, " + (d * 10) + ")";
});
```

2. add text labels

- also set dynamic text labels per bar, which reflect the current dataset

```
svg.selectAll("text")
.data(dataset)
.enter()
.append("text")
```

- extend this further by positioning our text labels

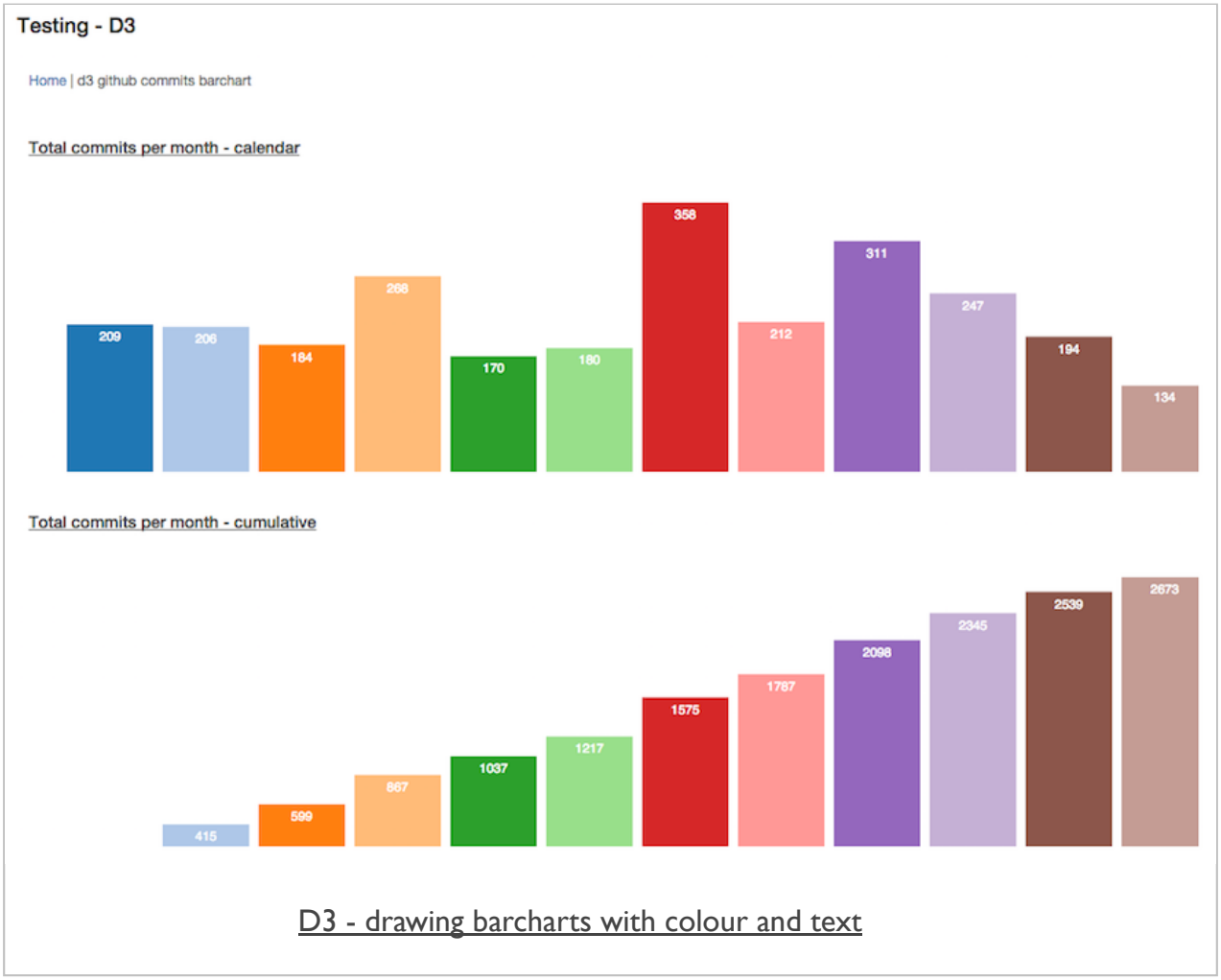
```
.attr("x", function(d, i) {
    return i * (w / dataset.length);
})
.attr("y", function(d, i) {
    return h - (d * 4);
});
```

- then position them relative to the applicable bars, add some styling, colours...

```
.attr("font-family", "sans-serif")
.attr("font-size", "11px")
.attr("fill", "white");
```

- DEMO - Drawing with SVG - barcharts, colour, and text labels

Image - D3 Barcharts



D3 - drawing barcharts with colour and text

Data visualisation - D3

Drawing - add interaction - listeners

- event listeners apply to any DOM element for interaction
  - *from a button to a <p> with the body of a HTML page*

```
<p>this is a HTML paragraph...</p>
```

- add a listener to this DOM element

```
d3.select("p")
    .on("click", function() {
    //do something with the element...
    });
```

- above sample code selects the <p> element
  - *then adds an event listener to that element*
- *event listener* is an anonymous function
  - *listens for* `.on` *event for a specific element or group of elements*
- in our example,
  - `on()` *function takes two arguments*

Data visualisation - D3

Drawing - add interaction - update visuals

- achieved by combining
  - *event listener*
  - *modification of the visuals relative to changes in data*

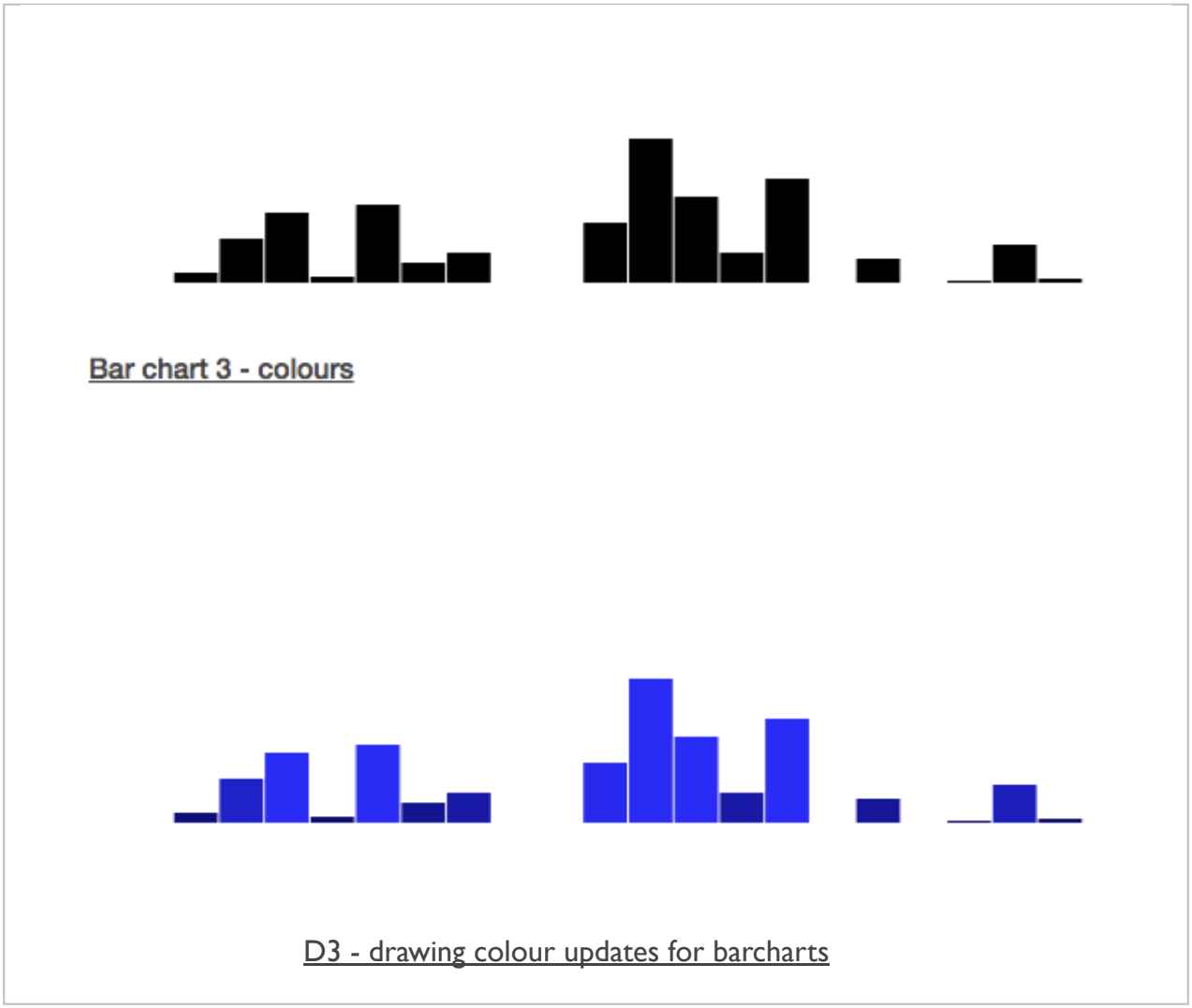
```
d3.select("p")
    .on("click", function() {

    dataset = [....];

    //update all of the rects
    svg.selectAll("rect")
    .data(dataset)
    .attr("y", function(d) {
    return h - yScale(d);
    });
    .attr("height", function(d) {
    return yScale(d);
    });
});
```

- above code triggers a change to visuals for each call to the event listener
- eg: change the colours
  - *add call to `fill()` to update bar colours*

```
.attr("fill", function( d) {
    return "rgb( 0, 0, " + (d * 10) + ")";
});
```

- DEMO - update bar colours

Image - D3 Barcharts



Data visualisation - D3

Drawing - add interaction - transitions

- adding a fun transition in D3 is as simple as adding the following,

```
.transition()
```

- add this to above code chain to get a fun and useful transition in the data
- animation reflects the change from the old to the new data
- add a call to the `duration()` function
  - *allows us to specify a time delay for the transition*
  - *quick, slow...we can specify each based upon time*
- chain the `duration()` function after `transition()`

```
.transition().duration(1000)
```

- if we want to specify a constant easing to the transition
  - *use `ease()` with a `linear` parameter*

```
.ease(linear)
```

- other built-in options, including
  - *circle - gradual ease in and acceleration until elements snap into place*
  - *elastic - best described as springy*
  - *bounce - like a ball bouncing, and then coming to rest...*

Drawing - add interaction - transitions

- add a delay using the `delay()` function

```
.transition()
.delay(1000)
.duration(2000)
```

- also set the `delay()` function dynamically relative to the data,

```
.transition()
.delay( function( d, i) {
return i * 100;
})
.duration( 500)
```

- when passed an anonymous function
  - *datum bound to the current element is passed into `d`*
  - *index position of that element is passed into `i`*
- in the above code example, as D3 loops through each element
  - *delay for each element is set to `i * 100`*
  - *meaning each subsequent element will be delayed 100ms more than preceding element*
- DEMO - transitions - interactive sort

Data visualisation - D3

Drawing - add interaction - adding values and elements

- select all of the bars in our chart
  - *we can rebind the new data to those bars*
  - *and grab the new update as well*

```
var bars = svg.selectAll("rect")
    .data(dataset);
```

- if more new elements, bars in our example, than original length
  - *use `enter()` to create references to those new elements that do not yet exist*
- with these reserved elements
  - *we can use `append()` to add those new elements to the DOM*
  - *now updates our bar chart as well*
- now made the new `rect` elements
  - *need to update all visual attributes for our `rects`*
  - *set x, and y position relative to new dataset length*
  - *set width and height based upon new xScale and yScale*
  - *calculated from new dataset length*

Data visualisation - D3

Drawing - add interaction - removing values and elements

- more DOM elements than provided data values
  - *D3's **exit** selection contains references to those elements without specified data*
  - ***exit** selection is simply accessed using the `exit()` function*
- grab the *exit* selection
- then transition exiting elements off the screen
  - *for example to the right*
- then finally remove it

```
bars.exit()
.transition()
.duration(500)
.attr("x", w)
.remove();
```

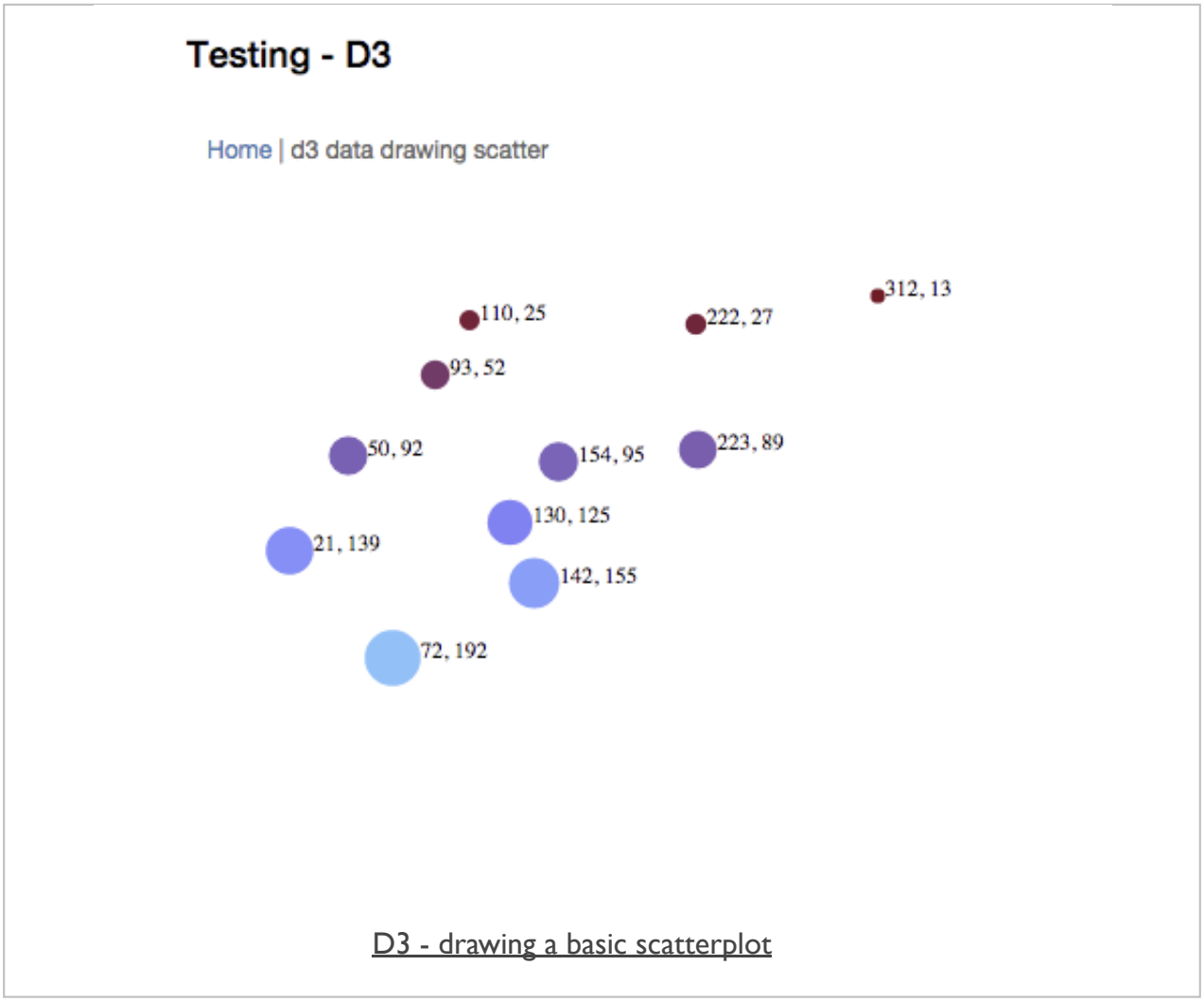
- `remove()` is a special transition method that awaits until transition is complete
- then deletes element from DOM forever
  - *to get it back, we'd need to rebuild it again*

Data visualisation - D3

Drawing - SVG scatterplot - intro

- scatterplot allows us to visualise two sets of values on two different axes
  - *one set of data against another*
- plot one set of data on *x* axis, and the other on the *y* axis
- often create dimensions from our data
  - *helps us define patterns within our dataset*
  - *eg: date against age, or age against fitness...*
- dimensions will also be represented relative to *x* and *y* axes
- create our scatterplot using SVG
  - *add our SVG to a selected element*

Image - D3 Scatterplot



D3 - drawing a basic scatterplot

Data visualisation - D3

Drawing - SVG scatterplot - data

- data for the scatterplot is normally stored as a multi-dimensional representation
  - *comparison x and y points*
- eg: we could store this data in a multi-dimensional array

```
var dataset = [
        [10, 22], [33, 8], [76, 39], [4, 15]
      ];
```

- in such a multi-dimensional array
  - *inner array stores the comparison data points for our scatterplot*
  - *each inner array stores x and y points for scatterplot diagram*
- we can also store such data in many different structures
  - *eg: JSON...*

Data visualisation - D3

Drawing - SVG scatterplot - create SVG

- need to create an element for our SVG
  - *allows us to draw and output all of our required shapes*

```
d3.select("body").append("svg");
```

- appends to the body an SVG element
  - *useful to encapsulate this new DOM element within a variable*

```
var svg = d3.select("body").append('svg');
```

- variable effectively works as a reference
  - *points to the newly created SVG object*
  - *allows us to use this reference to access element in the DOM*

Drawing - SVG scatterplot - build scatterplot

- as with our barchart, we can set the width and height for our scatterplot,

```
//width & height
var w = 750;
var h = 200;
```

- we will need to create circles for use with scatterplot instead of rectangles

```
svg.selectAll('circle')
   .data(dataset)
   .enter()
   .append('circle');
```

- corresponding to drawing circles
  - set `cx`, the x position value of the centre of the circle
  - set `cy`, the y position value of the centre of the circle
  - set `r`, the radius of the circle

Data visualisation - D3

Drawing - SVG scatterplot - adding circles

- draw circles for scatterplot

```
.attr('cx', function(d) {
    return d[0]; //get first index value for inner array
})
.attr('cy', function(d) {
    return d[1]; //get second index value for inner array
})
.attr('r', 5);
```

- outputs simple circle for each inner array within our supplied multi-dimensional dataset
- start to work with creating circle sizes relative to data quantities
- set a dynamic size for each circle
  - *representative of the data itself*
  - *modify the circle's area to correspond to its y value*
- as we create SVG circles, we cannot directly set the area
  - *so we need to calculate the radius* `r`
  - *then modify that for each circle*

Data visualisation - D3

Drawing - SVG scatterplot - calculate dynamic area

- assuming that `d[1]` is the original area value of our circles
  - *get the square root and set the radius for each circle*
- instead of setting each circle's radius as a static value
  - *now use the following*

```
.attr('r', function(d) {
    return Math.sqrt(d[1]);
});
```

- use the JavaScript `Math.sqrt()` function to help us with this calculation

Data visualisation - D3

Drawing - SVG scatterplot - add colour

- as with a barchart
- also set a dynamic colour relative to a circle's data

```
.attr('fill', function (d) {
    return 'rgb(125,' + (d[1]) + ', ' + (d[1] * 2) + ')';
});
```

Data visualisation - D3

Drawing - SVG scatterplot - add labels

```
//add labels for each circle
svg.selectAll('text')
   .data(dataset)
   .enter()
   .append('text')
   .text(function(d) {
    return d[0] + ', ' + d[1];//set each data point on the text label
   })
   .attr('x', function(d) {
    return d[0];
   })
   .attr('y', function(d) {
    return d[1];
   })
   .attr('font-family', 'serif')
   .attr('font-size', '12px')
   .attr('fill', 'navy');
```

- start by adding text labels for our data
  - *adding new text elements where they do not already exist*
- then set the text label itself for each circle
  - *using the data values stored in each inner array*
- make the label easier to read
  - *set x and y coordinates relative to data points for each circle*
- set some styles for the labels

Image - D3 Scatterplot

D3 - drawing a basic scatterplot 2

Data visualisation - D3

Drawing - SVG - scales

- in D3, *scales* are defined as follows,

> "*Scales are functions that map from an input domain to an output range*"
>
> *Bostock, M.*

- you can specify your own scale for the required dataset
  - *eg: to avoid massive data values that do not translate correctly to a visualisation*
  - *scale these values to look better within you graphic*
- to achieve this result, you simply use the following pattern.
  - *define the parameters for the scale function*
  - *call the scale function*
  - *pass a data value to the function*
  - *the scale function returns a scaled output value for rendering*
- also define and use as many scale functions as necessary for your visualisation
- important to realise that a scale has no direct relation to the visual output
  - *it is a mathematical relationship*
- need to consider scales and axes
  - *two separate, different concepts relative to visualisations*

Data visualisation - D3

Drawing - SVG - domains and ranges

- *input domain* for a scale is its possible range of input data values
  - *in effect, initial data values stored in your original dataset*
- *output range* is the possible range of output values
  - *normally use as the pixel representation of the data values*
  - *a personal consideration of the designer*
- normally set a minimum and maximum *output range* for our scaled data
- scale function then calculates the scaled output
  - *based upon original data and defined range for scaled output*
- many different types of scale available for use in D3
- three primary types
  - *quantitative*
  - *ordinal*
  - *time*
- *quantitative* scale types also include other built-in scale types
- many methods available for the scale types

Data visualisation - D3

Drawing - SVG - building a scale

- start building our scale in D3
  - *use `d3.scale` with our preferred scale type*

```
var scale = d3.scale.linear();
```

- to use the scale effectively, we now need to set our input domain

```
scale.domain([10, 350]);
```

- then we set the output range for the scale

```
scale.range([1, 100]);
```

- we can also chain these methods together

```
var scale = d3.scale.linear()
          .domain([10, 350])
              .range([1, 100]);
```

Data visualisation - D3

Drawing - SVG - adding dynamic scales

- we could pre-define values for our scale relative to a given dataset
- makes more sense to abstract these values relative to the defined dataset
- we can now use the D3 array functions to help us set these scale values
  - *eg; find highest number in array dataset*

```
d3.max(dataset, function(d) {
    return d[0];
});
```

- returns highest value from the supplied array
- getting minimum value in array works in the same manner
  - *with `d3.min()` being called instead*
- now create a scale function for `x` and `y` axes

```
var scaleX = d3.scale.linear()
        .domain([0, d3.max(dataset, function(d) { return d[0]; })])
        .range([0, w]);//set output range from 0 to width of svg
```

- Y axis scale modifies above code relative to provided data, `d[1]`
  - *range uses height instead of width*
- for a scatterplot we can use these values to set `cx` and `cy` values

Image - D3 Scatterplot

Data visualisation - D3

Drawing - SVG - adding dynamic scales

- a few data visualisation examples
- Tests 1
- Tests 2

React JavaScript Library

overview

- **React** began life as a port of a custom PHP framework called XHP
  - *developed internally at Facebook*
- XHP, as a PHP framework, was designed to render the full page for each request
- **React** developed from this concept
  - *creating a client-side implementation of loading the full page*
- **React** can, therefore, be perceived as a type of *state machine*
  - *control and manage inherent complexity of state as it changes over time*
- able to achieve this by concentrating on a narrow scope for development,
  - *maintaining and updating the DOM*
  - *responding to events*
- **React** is best perceived as a *view* library
  - *no definite requirements or restrictions on storage, data structure, routing...*
- allows developers freedom
  - *incorporate **React** code into a broad scope of applications and frameworks*

React JavaScript Library

why use React?

- React is often considered the *V* in the traditional *MVC*
- [React(http://facebook.github.io/react/docs/why-react.html) was designed to solve one problem
  - ***building large applications with data that changes over time***
- React can best be considered as addressing the core concerns
  - *simple, declarative, components*
- simple - define how your app should look at any given point in time
  - *React handles all UI changes and updates in response to data changes*
- declarative - as data changes, React effectively refreshes your app
  - *sufficiently aware to only update those parts that have changed*
- components - fundamental principle of React is building re-usable components
  - *components are encapsulated in their design and concepts*
  - *they make it simple for code re-use, testing...*
  - *in particular, the separation of design and app concerns in general*
- React leverages its built-in, powerful rendering system to produce
  - *quick, responsive rendering of DOM in response to received state changes*
- uses a virtual DOM
  - *enables React to maintain and update the DOM without the lag of reading it as well*

state changes

- as **React** is informed of a state change, it re-runs render functions
- enables it to determine a new representation of the page in its virtual DOM
- then automatically translated into the necessary changes for the new DOM
  - *reflected in the new rendering of the view*
- may, at first glance, appear inherently slow
  - *React uses an efficient algorithm*
  - *checks and determines differences*
  - *differences between current page in the virtual DOM and the new virtual one*
- from these differences it makes the minimal set of necessary updates to the rendered DOM
- creates speed benefits and gains
  - *minimises usual reflows and DOM manipulations*
- also minimises effect of cascading updates caused by frequent DOM changes and updates

component lifecycle

- in the lifecycle of a component
  - *its props or state might change along with any accompanying DOM representation*
- in effect, a component is a known state machine
  - *it will always return the same output for a given input*
- following this logic, React provides components with certain *lifecycle* hooks
  - *instantiation - mounting*
  - *lifetime - updating*
  - *teardown - unmounting*
- we may consider these hooks
  - *first through the instantiation of the component*
  - *then its active lifetime*
  - *finally its teardown*

React JavaScript Library

a few benefits

- one of the main benefits of this virtual approach
  - *avoidance of micro-managing any updates to the DOM*
- a developer simply informs React of any changes
  - *such as user input*
- React is able to process those passed changes and updates
- React has inherent benefit of delegating all events to a single event handler
  - *naturally gives React an associated performance boost*

React JavaScript Library

getting started - part 1

- React's starter kit
  - *gives us the required React JS file, the JSX transform JS file, and many examples and demos*
- choose whether we want to use
  - *JSX, plain JavaScript, or pre-compile the former into the latter before deploying our application*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <script src="build/JSXTransformer.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/jsx">
      React.render(
        <h1>Hello React World!</h1>,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

- uses the JSX Transformer to create plain JavaScript for rendering

React JavaScript Library

getting started - part 2

- perform offline transform using **react-tools**, available as an *npm* package

```
npm install -g react-tools
```

- using these tools, translate our React code file, `src/helloworld.js`, to plain JavaScript

```
jsx --watch src/ build/
```

- **NB:** `build/helloworld.js` autogenerated whenever we make a change in our React code
- updated HTML source file is as follows

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello React!</title>
        <!-- required react files -->
        <script src="build/react.js"></script>
        <!-- no need for JSX transformer if we use NPM package React tools... -->
        <!--<script src="build/JSXTransformer.js"></script>-->
    </head>
    <body>
        <div id="example"></div>
        <script src="build/helloworld.js"></script>
    </body>
</html>
```

React JavaScript Library

JSX - intro

- JSX stands for **JavaScript XML**
  - *follows an XML familiar syntax for developing markup within React components*
- JSX is not compulsory within React
  - *it makes components easier to read and understand*
  - *its structure is more succinct and less verbose*
- A few defining characteristics of JSX
  - *each JSX node maps to a function in JavaScript*
  - *JSX does not require a runtime library*
  - *JSX does not supplement or modify the underlying semantics of JavaScript*

React JavaScript Library

JSX - benefits

- why use JSX, in particular when it simply maps to JavaScript functions?
- many of the inherent benefits of JSX become more apparent
  - *as an application, and its code base, grows and becomes more complex*
- benefits can include
  - *a sense of familiarity - easier with experience of XML and DOM manipulation*
  - *eg: React components capture all possible representations of the DOM*
  - *JSX transforms an application's JavaScript code into semantic, meaningful markup*
  - *permits declaration of component structure and information flow using a similar syntax to HTML*
  - *permits use of pre-defined HTML5 tag names and custom components*
  - *easy to visualise code and components*
  - *considered easier to understand and debug*
  - *ease of abstraction due to JSX transpiler*
  - *abstracts process of converting markup to JavaScript*
  - *unity of concerns*
  - *no need for separation of view and templates*
  - *React encourages discrete component for each concern within an application*
  - *encapsulates the logic and markup in one definition*

JSX - composite components

- example React component might allow us to output a HTML heading

```
var heading = React.createClass({
  render: function() {
    return (
      <div className="heading">
      <h2>Welcome</h2>
      </div>
    );
  }
});
```

- currently fixed to `Welcome` heading
- now update this example to work with dynamic values
- JSX considers values dynamic if they are placed between curly brackets `{..}`
  - *treated as JavaScript context*

```
var heading = 'Welcome';
<h2>{heading}</h2>
```

JSX - more dynamic values

- also call functions
  - *move some logic for a component to a standard JavaScript function*
- then call this function, plus any supplied parameters
- React can also evaluate arrays, and then output each value

```
var heading = React.createClass({
  render: function() {
    var text = ['welcome', 'home'];
      return (
        <h3>{text}</h3>
      );
    }
});

React.render(<heading />, document.getElementById('example'));
```

JSX - conditionals

- a component's markup and its logic are inherently linked in React
- this naturally includes *conditionals*, *loops*...
- adding `if` statements directly to JSX will create invalid JavaScript

1. ternary operator

```
render: function() {
  return <div className={
    this.state.isComplete ? 'is-complete' : ''
  }>...</div>
}
```

2. variable

```
getIsComplete: function() {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function() {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>
}
```

3. function call

```
getIsComplete: function() {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function() {
  return <div className={this.getIsComplete()}>...</div>;
}
```

- to handle React's lack of output for *null* or *false* values
  - *use a boolean value and follow it with the desired output*

JSX - non-DOM attributes - part 1

- in JSX, there are currently three special attribute names
  - `key`
  - `ref`
  - `dangerouslySetInnerHTML`

1. `key`

- an optional unique identifier that remains consistent throughout render passes
- informs React so it can more efficiently select when to reuse or destroy a component
- helps improve the rendering performance of the application.
- eg: if two elements already in the DOM need to switch position
  - *React is able to match the keys and move them*
  - *does not require unnecessary re-rendering of the complete DOM*

JSX - non-DOM attributes - part 2

2. `ref`

- `ref` permits parent components to easily maintain a reference to child components
  - *available outside of the render function*
- to use `ref`, simply set the attribute to the desired reference name

```
render: function() {
  return <div>
    <input ref="myInput" ... />
    </div>;
}
```

- able to access this `ref` using the defined `this.refs.myInput`
  - *access anywhere in the component*
  - *object accessed through this `ref` known as a backing instance*
- **NB:** not the actual DOM
  - *a description of the component React uses to create the DOM when necessary*
- access DOM itself for this `ref`
  - *use `this.refs.myInput.getDOMNode()`, where mylnput is name of previously defined ref*

JSX - non-DOM attributes - part 3

3. `dangerouslySetInnerHTML`

- When absolutely necessary, React can set HTML content as a string using this attribute
- to correctly use this property set it as an object with key `__html`

```
render: function() {
  var htmlString = {
    __html: "<span>...your html string...</span>"
  };
  return <div dangerouslySetInnerHTML={htmlString}></div>;
}
```

JSX - reserved words

- JSX transforms to plain JavaScript functions
  - *means there are some reserved or special attributes*
  - *eg: we can't use `class` or `for`*
- to create a form label with the `for` attribute we can use `htmlFor` instead

```
<label htmlFor="text...">
```

- create a custom class we can use `className`

```
<div className={class...}>
```

data flow

- data flows in one direction in React
  - *namely from **parent to child***
- helps to make components nice and simple, and predictable as well
- components take *props* from the parent, and then render
- if a *prop* has been changed, for whatever reason
  - *React will update the component tree for that change*
  - *then re-render any components that used that property*
- Internal state also exists for each component
  - *state should only be updated within the component itself*
- we can think of data flow in React
  - *in terms of* `props` *and* `state`

data flow - `props` - part I

- `props` can hold any data and are passed to a component for usage
- set `props` on a component during instantiation

```
var classics = [{ title: 'Greek'}];
<ListClassics classics={classics}/>
```

- also use the `setProps` method on a given instance of a component

```
var ListClassics = React.createClass({
  render: function() {
    return (
      <li className="classic">{this.props.classics}</li>
    );
  }
});

var classics = [{ title: 'Greek'}];
var listClassics = React.render (
  <ListClassics/>,
    document.getElementById('example')
);

listClassics.setProps({ classics: classics });
```

data flow - `props` with JSX

- set `props` using { } syntax
  - *allows us to pass variables of any type via JavaScript injection*

```
<a href={'/classics/' + classic.id}>{classic.title}</a>
```

- also pass event handlers as `props`

```
var EditButton = React.createClass({
  render: function() {
    return (
      <a className='button edit' onClick={this.handleClick}>Edit</a>
      );
  },
  handleClick: function() {
  //handle click...
    alert('edit button clicked...');
  }
});
```

React JavaScript Library

state - intro - part 1

- a component in React is able to house *state*
- *State* is inherently different from `props` because it is internal to the component
- it is particularly useful for deciding a view state on an element
  - *eg: we could use state to track options within a hidden list or menu*
  - *track the current state*
  - *change it relative to component requirements*
  - *then show options based upon this amended state*
- **NB:** considered bad practice to update state directly using `this.state`
  - *use the method `this.setState`*
- try to avoid storing computed values or components directly in *state*
- focus upon using simple data
  - *directly required for given component to function correctly*
- considered good practice to perform required calculations in the `render` function
- try to avoid duplicating `prop` data into `state`
  - *use the `props` data instead*

state - intro - part 2

```
var EditButton = React.createClass({
  getInitialState: function() {
    return {
      editShow: true
    };
  },
  render: function() {
    if (this.state.editShow == false) {
      alert('edit button will be turned off...');
    }
    return (
      <button className="button edit" onClick={this.handleClick}>Edit</button>
    );
  },
  handleClick: function() {
  //handle click...
  alert('edit button clicked');
  //set state after button click
  this.setState({ editShow: false });
  }
});
```

state - intro - part 3

- when designing React apps, we often think about
  - ***stateless children*** *and a* ***stateful parent***

> *A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via props.*
>
> *React documentation*

- need to carefully consider how to identify and implement this type of component hierarchy

1. Stateless child components
   - *components should be passed data via* `props` *from the parent*
   - *to remain stateless they should not manipulate their* `state`
   - *they should send a callback to the parent informing it of a change, update etc*
   - *parent will then decide whether it should result in a* `state` *change, and a re-rendering of the DOM*
2. Stateful parent component
   - *can exist at any level of the hierarchy*
   - *does not have to be the root component for the app*
   - *instead can exist as a child to other parents*
   - *use parent component to pass* `props` *to its children*
   - *maintain and update state for the applicable components*

state - intro - part 4

1. props vs state
   - *in React, we can often consider two types of model data*
   - *includes `props` and `state`*
   - *most components normally take their data from `props`*
   - *allows them to render the required data*
   - *as we work with users, add interactivity, and query and respond to servers*
   - *we also need to consider the `state` of the application*
   - *`state` is very useful and important in React*
   - *also important to try and keep many of our components stateless*
2. state
   - *React considers user interfaces, UIs, as simple state machines*
   - *acting in various states and then rendering as required*
   - *in React, we simply update a component's state*
   - *then render the new corresponding UI*

state - intro - part 5

1. How state works

- if there is a change in data in the application
  - *perhaps due to a server update or user interaction*
  - *quickly and easily inform React by calling `setState(data, callback)`*
- this method allows us to easily merge `data` into `this.state`
  - *re-renders the component*
- as re-rendering is finished
  - *optional `callback` is available and is called by React*
- this `callback` will often be unnecessary
  - *it's still useful to know it is available*

state - intro - part 6

2. In state

- try to keep data in `state` to a minimum
  - *consider minimal possible representation of an application's state*
  - *helps build a stateful component*
- `state` should try to just contain minimal data
  - *data required by a component's event handlers to help trigger a UI update*
  - *if and when they are modified*
- such properties should also normally only be stored in `this.state`
- as we render the updated UI
  - *simply compute required information in the `render()` method based on this `state`*
  - *avoids need to keep computed values in sync in state*
  - *instead relying on React to compute them for us*

3. out of state

- in React, `this.state` should only contain minimal data
- minimum necessary to represent an application's UI state
- should contain
  - *computed value*
  - *React components*
  - *duplicated data from `props`*

state - an example app - part I

- a simple app to allow us to test the concept of stateful parent and stateless child components
- resultant app outputs two parallel `div` elements
- allow a user to select one of the available categories
- then view all of the available *authors*

```
//static test data...
var AUTHORS = [
  {id:1, category: 'greek', categoryId:1, author: 'Plato'},
  {id:2, category: 'greek', categoryId:1, author: 'Aristotle'},
  {id:3, category: 'greek', categoryId:1, author: 'Aeschylus'},
  {id:4, category: 'roman', categoryId:2, author: 'Livy'},
  {id:5, category: 'greek', categoryId:1, author: 'Euripides'},
  {id:6, category: 'roman', categoryId:2, author: 'Ptolemy'},
  {id:7, category: 'greek', categoryId:1, author: 'Sophocles'},
  {id:8, category: 'roman', categoryId:2, author: 'Virgil'},
  {id:9, category: 'roman', categoryId:2, author: 'Juvenal'}
];
```

- start with some static data to help populate our app
- `categoryId` used to filter unique categories
  - *again to help get all of our authors per category*

state - an example app - part 2

- for `stateless` child components
  - *need to output a list of filtered, unique categories*
  - *then a list of authors for each selected category*
- first child component is the `CategoryList`
  - *filters and renders our list of unique categories*
  - *`onClick` attribute is included*
  - *state is therefore passed via callback to the `stateful` parent*

state - an example app - part 3

```
//output unique categories from passed data...
var CategoryList = React.createClass({
 render: function() {
  var category = [];
   return (
    <div id="left-titles" className="col-6">
     <ul>
      {this.props.data.map(function(item) {
        if (category.indexOf(item.category) > -1) {
        } else {
         category.push(item.category);
          return (
           <li key={item.id} onClick={this.props.onCategorySelected.bind(null, item.catego
            {item.category}
           </li>);
          }}, this)}
      </ul>
     </div>
   );
  }
});
```

- the component is accepting `props` from the parent component
  - *then informing this parent of a required change in state*
  - *change reported via a callback to the `onCategorySelected` method*
  - *does not change `state` itself*
  - *it simply handles the passed data as required for a React app*

state - an example app - part 4

- need to consider our second `stateless` child component
  - *renders the user's chosen authors per category*
  - *user clicks on their chosen category*
  - *a list of applicable authors is output to the right side div*

```
var AuthorList = React.createClass({
 render: function() {
  return (
   <div id="right-titles" className="col-md-6 col-sm-6 col-xs-6">
    <ul>
     {this.props.authors.map(function(item) {
      return (
       <li key={item.id}>{item.author}</li>
      );
     })
    }
    </ul>
   </div>
  );
 }
});
```

- this component does not set any state
- simply rendering the passed `props` data for viewing

state - an example app - part 5

- to handle updates to the DOM, we need to consider our `stateful` parent
- this component passes the app's data as `props` to the children
- handles the setting and updating of the `state` for app as well
- as noted in the React documentation,

> *State should contain data that a component's event handler may change to trigger a UI update.*

- for this example app
  - *only need to store the `selectedCategoryAuthors` in `state`*
  - *enables us to update the UI for our app*

state - an example app - part 6

```
var Container = React.createClass({
    getInitialState: function() {
        return {
        selectedCategoryAuthors: this.getCategoryAuthors(this.props.defaultCategoryId)
        };
    },
  getCategoryAuthors: function(categoryId) {
        var data = this.props.data;
        return data.filter(function(item) {
            return item.categoryId === categoryId;
        });
    },
  render: function() {
    return (
      <div className="container col-md-12 col-sm-12 col-xs-12">
      <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected} />
      <AuthorList authors={this.state.selectedCategoryAuthors} />
      </div>
    );
  },
  onCategorySelected: function(categoryId) {
    this.setState({
      selectedCategoryAuthors: this.getCategoryAuthors(categoryId)
    });
  }
});
```

state - an example app - part 7

- our `stateful` parent component sets its initial state
  - *including passed data and app's selected category for authors*
- helps set a default state for the app
  - *we can then modify as a user selects their chosen category*
- callback for this user selected category is handled in the `onCategorySelected` method
  - *updates the app's state for the chosen `categoryId`*
  - *then leads to the app re-rendering the DOM for any changes*
- we still have computed data in the app's `state`
  - *as noted in the React documentation,*

> *`this.state` should only contain the minimal amount of data needed to represent your UIs state...*

- we should now move our computations to the `render` method of the parent component
  - *then update `state` accordingly*

state - an example app - part 8

```
var Container = React.createClass({
    getInitialState: function() {
    return {
      selectedCategoryId: this.props.defaultCategoryId
    };
  },
  render: function() {
    var data = this.props.data;
    var selectedCategoryAuthors = data.filter(function(item){
      return item.categoryId === this.state.selectedCategoryId;
    }, this);
    return (
        <div className="container col-md-12 col-sm-12 col-xs-12">
        <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected}
        <AuthorList authors={selectedCategoryAuthors} />
        </div>
    );
  },
    onCategorySelected: function(categoryId) {
    this.setState({selectedCategoryId: categoryId});
  }
});
```

- `state` is now solely storing the `categoryId` for our app
- can be modified and the DOM re-rendered correctly

state - an example app - part 9

- we can then load this application
  - *passing data as props to the `Container`*
  - *data from JSON Authors*

```
var buildLibrary = React.render (
 <Container data={AUTHORS} defaultCategoryId='1' />,
 document.getElementById('library')
);
```

- DEMO - state example

state - minimal state - part 1

- to help make our UI interactive
  - *use React's `state` to trigger changes to the underlying data model of an application*
  - *need to keep a minimal set of mutable state*
- **DRY**, or *don't repeat yourself*
  - *often cited as a good rule of thumb for this minimal set*
- need to decide upon an absolute minimal representation of the `state` of the application
  - *then compute everything else as required*
  - *eg: if we maintain an array of items*
  - *common practice to calculate array length as needed instead of maintaining a counter*

state - minimal state - part 2

- as we develop an application with React
  - *start dividing our data into logical pieces*
  - *then start to consider which is state*
- for example,
  - *is it from `props`?*
  - *if yes, this is probably not `state` in React*
  - *does it update or change over time? (eg: due to API updates etc)*
  - *if yes, this is probably not `state`*
  - *can you compute the data based upon other `state` or `props` in a component?*
  - *if yes, it is not state*
- need to decide upon our minimal set of components that mutate, or *own* state
  - *React is based on the premise of one-way data flow down the hierarchy of components*
  - *can often be quite tricky to determine*
- initially, we can check the following
  - *each component that renders something based on state*
  - *determine the parent component that needs the state in the hierarchy*
  - *a common or parent component should own the state*
    - *NB:* if this can't be determined
    - simply create a basic component to hold this state
    - add component at the top of the state hierarchy

component lifecycle - intro

- React components include a minimal lifecycle API
- provides the developer with enough without being overwhelming
  - *at least in theory*
- React provides what are known as *will* and *did* methods
  - *will - called right before something happens*
  - *did - called right after something happens*
- relative to the lifecycle, we can consider the following groupings of methods
  1. Instantiation (mounting)
  2. Lifetime (updating)
  3. Teardown (unmounting)
  4. Anti-pattern (calculated values)

component lifecycle - method groupings - Instantiation (mounting)

- includes methods called upon instantiation for the selected component class
- eg: `getDefaultProps` or `getInitialState`
  - *use such methods to set default values for new instances*
  - *initialise a custom state of each instance...*
- also have the important `render` method
  - *builds our application's virtual DOM*
  - *the only required method for a component*
- `render` method has rules it needs to follow
  - *such as accessible data*
  - *return values*
- `render` method must also remain *pure*
  - *cannot change the state or modify the DOM output*
  - *returned result is the virtual DOM*
  - *compared against actual DOM*
  - *helps determine if changes are required for the application*

component lifecycle - method groupings - Lifetime (updating)

- component has now been rendered to the user for viewing and interaction
- as a user interacts with the component
  - *they are changing the state of that component or application*
  - *allows us as developers to act on the relevant points in the component tree*
- State changes for the application
  - *those affecting the component*
  - *may result in update methods being called*
- we're telling the component how and when to update

component lifecycle - method groupings - Teardown (unmounting)

- as React is finished with a component
  - *it must be unmounted from the DOM and destroyed*
- there is a single hook for this moment
  - *provides opportunity to perform necessary cleanup and teardown*
- `componentWillUnmount`
  - *removes component from component hierarchy*
  - *this method cleans up the application before component removal*
  - *undo custom work performed during component's instantiation*

component lifecycle - method groupings - Anti-pattern (calculated values)

- React is particularly concerned with maintaining a single source of truth
- one point where `props` and `state` are derived, set...
- consider calculated values derived from `props`
  - *considered an anti-pattern to store these calculated values as state*
- if we needed to convert a `props` date to a string for rendering
  - *this is not state*
  - *it should simply be calculated at the time of `render`*

React JavaScript Library

Additional reading, material, and samples

- design thoughts
- event handling
- more composing components
- DOM manipulation
- forms
- intro to flux
- animations
- lots of samples...

Demos

- D3.js
  - *D3 basic elements*
  - *Drawing with SVG*
  - *Drawing with SVG - barcharts*
  - *Drawing with SVG - barcharts, colour, and text labels*

React

- state example

References

- D3.js
  - *D3 - API reference*
  - *D3 - Easing*
  - *D3 - Scales*
  - *D3 - Wiki*
- React
  - *React*
  - *React - API Reference*
  - *React - Starter Kit*
- W3 Selector API