Comp 324/424 - Client-side Web Design

Spring Semester 2019 - Week 14

Dr Nick Hayward

Final Demo and Presentation

- presentation and demo - live working app...
  - *due on Monday 22nd April 2019 @ 4.15pm*
  - ***NO** content management systems (CMSs) such as Drupal, Joomla, WordPress...*
  - ***NO** PHP, Python, Ruby, C# & .Net, Go, XML...*
  - ***NO** CSS frameworks, such as Bootstrap, Foundation, Materialize...*
- explain chosen app's logic and structure
  - *data store, API, deferred patterns &c.*
  - *any data visualisations used*
  - *...*
- must implement data from either
  - *self hosted (MongoDB, Redis...)*
  - *APIs*
  - *cloud services, storage (Firebase, Heroku, mLab &c.)*
  - ***NO** SQL...*
- explain design decisions
  - *describe patterns used in design of UI and interaction*
  - *layout choices...*
- show and explain implemented differences from DEV week
  - *where and why did you update the app?*
  - *perceived benefits of the updates?*
- how did you respond to peer review?
- anything else useful for final assessment...
- consider outline of content from final report outline
- ...

All project code must be pushed to a repository on GitHub.

n.b. present your own work contributed to the project, and its development...

Final Report

Report due on 29th April 2019 by 4.15pm

- final report outline - coursework section of website
  - *PDF*
  - *group report*
  - ***extra individual report*** *- optional*
- include repository details for project code on GitHub

JavaScript - Prototype

intro

- along with the following traits of JS (ES6 ...),
- functions as first-class *objects*
- versatile and useful structure of functions with closures
- combine generator functions with promises to help manage async code
- async & await...
- *prototype* object may be used to delegate the search for a particular property
- a *prototype* is a useful and convenient option for defining properties and functionality
- accessible to other objects
- a *prototype* is a useful option for replicating many concepts in traditional object oriented programming

JavaScript - Prototype

understanding prototypes

- in JS, we may create objects, e.g. using *object-literal* notation
- a simple value for the first property
- a function assigned to the second property
- another object assigned to the third object

```
let testObject = {
    property1: 1,
    prooerty2: function() {},
    property3: {}
}
```

- as a dynamic language, JS will also allow us to
- modify these properties
- delete any not required
- or simply add a new one as necessary
- this dynamic nature may also completely change the properties in a given object
- this issue is often solved in traditional object-oriented languages using inheritance
- in JS, we can use *prototype* to implement inheritance

JavaScript - Prototype

basic idea of prototypes

- every *object* can have a reference to its *prototype*
- a delegate object with properties - default for child objects
- JS will initially search the onject for a property
- then, search the *prototype*
- i.e. prototype is a fall back object to search for a given property &c.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);

console.log(object1.author);
```

- in the above example, we define two objects
- properties may be called with standard object notation
- can be modified and mutated as standard
- use `setPrototypeOf()` to set and update object's prototype
- e.g. `object1` as object to update
- `object2` as the object to set as prototype
- if requested property is not available on `object1`
- JS will search defined prototype...
- `author` available as property of prototype for `object1`
- demo - basic prototype

JavaScript - Prototype

prototype inheritance

- *Prototypes*, and their properties, can also be inherited
- creates a chain of inheritance...
- e.g.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };
const object3 = { genre: 'fiction' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);
Object.setPrototypeOf(object2, object3);

console.log(object1.author);
console.log(`genre from prototype chain = ${object1.genre}`); // use template lit
```

- `object1` has access to the prototype of its parent, `object2`
- a property search against `object1` will now include its own prototype, `object2`
- and its prototype as well, `object3`
- output for `object1.genre` will return the value stored in the property on `object3`
- demo - basic set prototype

JavaScript - Prototype

object constructor & prototypes

- object-oriented languages, such as Java and C++, include a class constructor
- provides known encapsulation and structuring
- constructor is initialising an object to a known initial state...
- i.e. consolidate a set of properties and methods for a class of objects in one place
- JS offers such a mechanism, although in a slightly different form to Java, C++ &c.
- JS still uses the `new` operator to instantiate new objects via constructors
- JS does not include a true class definition comparable to Java &c.
- ES6 `class` is syntactic sugar for the `prototype`...
- `new` operator in JS is applied to a constructor function
- this triggers the creation of a new object

JavaScript - Prototype

prototype object

- in JS, every function includes their own prototype object
- set automatically as the prototype of any created objects
- e.g.

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();

console.log(bookRecord.library);
```

- likewise, we may set a default method on an instantiated object's prototype
- demo - basic prototype object

JavaScript - Prototype

instance properties

- as JS searches an object for properties, values or methods
- instance properties will be searched before trying the prototype
- a known order of precedence will work.
- e.g.

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';

  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();

console.log(bookRecord.library);
```

- `this` refers directly to the newly created object
- properties in constructor created directly on instantiated object
- e.g. instance of `LibraryRecord()`
- search for `library` property against object
- do not need to search against prototype for this example
- known side-effect
- instantiate multiple objects with this constructor
- each object gets its own copy of the constructor's properties & access to same prototype
- may end up with multiple copies of same properties in memory
- if replication is required or likely
- more efficient to store properties & methods against the prototype

- demo - basic prototype object properties

JavaScript - Prototype

side effects of JS dynamic nature

- JS is a dynamic language
- properties can be added, removed, modified...
- dynamic nature is true for prototypes
- function prototypes
- object prototypes

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';
}

// create instance of LibraryRecord - call constructor with `new` operator
const bookRecord1 = new LibraryRecord();

// check output of value for library property from constructor
console.log(`this library = ${bookRecord1.library}`);

// add method to prototype after object created
LibraryRecord.prototype.updateLibrary = function() {
  return this.retreat = 'mariafels';
};

// check prototype updated with new method
console.log(`this retreat = ${bookRecord1.updateLibrary()}`);

// then overwrite prototype - constructor for existing object unaffected...
LibraryRecord.prototype = {
  archive: 'mariafels',
  order: 'benedictine'
};

// create instance object of LibraryRecord...with updated prototype
const bookRecord2 = new LibraryRecord();

// check output for second instance object
console.log(`updated archive = ${bookRecord2.archive} and order = ${bookRecord2.o
// check output for second instance object - library
console.log(`second instance object - library = ${bookRecord2.library}`);
// check if prototype updated for first instance object - NO
```

```
console.log(`first instance object = ${bookRecord1.order}`);
// manual update to prototype for first instance object still available
console.log(`this retreat2 = ${bookRecord1.updateLibrary()}`);

// check prototype has been fully overwritten - e.g. `updateLibrary()` no longer
try {
// updates to original prototype are overridden - error is returned for second in
console.log(`this retreat = ${bookRecord2.updateLibrary()}`);
 } catch(error) {
  console.log(`modified prototype not available for new object...\n ${error}`);
 }
```

- demo - basic prototype dynamic

JavaScript - Prototype

object typing via constructors

- check function used as a constructor to instantiate an object
- using `constructor` property

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

// create instance object for libraryRecord
const bookRecord = new LibraryRecord();

// output constructor for instance object
console.log(`constructor = ${bookRecord.constructor}`);

// check if function was constructor (use ternary conditional)
const check = bookRecord.constructor === LibraryRecord ? true : false;
// output result of check
console.log(check);
```

- demo - basic constructor check

JavaScript - Prototype

instantiate a new object using a constructor reference

- use a constructor to create a new instance object
- also use `constructor()` of new object to create another object
- second object is still an object of the original constructor

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();
const bookRecord2 = new bookRecord.constructor();
```

JavaScript - Prototype

achieving inheritance

- *Inheritance* enables re-use of an object's properties by another object
- helps us efficiently avoid repetition of code and logic
- improving reuse and data across an application
- in JS, a prototype chain to ensure inheritance works beyond simply copying prototype properties
- e.g. a book in a corpus, a corpus in an archive, an archive in a library...

JavaScript - Prototype

inheritance with prototypes - part 1

- *inheritance* in JS
- create a prototype chain using an instance of an object as prototype for another object
- e.g.

```
SubClass.prototype = new SuperClass()
```

- this pattern works as a prototype chain for inheritance
- prototype of `SubClass` instance as an instance of `SuperClass`
- prototype will have all the properties of `SuperClass`
- `SuperClass` may also have properties from its superclass...
- prototype chain created of expected inheritance

JavaScript - Prototype

inheritance with prototypes - part 2

- e.g. inheritance achieved by setting prototype of `Archive` to instance of `Library` object

```
//constructor for object
function Library() {
    // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// constructor for Archive object
function Archive(){
    // instance property
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
    // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
    // domain property from Archive
    console.log(`Archive domain = ${archiveRecord.domain}`);
}
```

JavaScript - Prototype

issues with overriding the constructor property

- setting `Library` object as defined prototype for `Archive` constructor

```
Archive.prototype = new Library();
```

- connection to `Archive` constructor **lost** - we may check constructor

```
// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
    // Library constructor output - due to prototype
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
}
```

- `Library` constructor will be returned
- *n.b.* may become an issue - constructor property may be used to check original function for instantiation
- demo - inheritance with prototype

JavaScript - Prototype

some benefits of overriding the constructor property

```
//constructor for object
function Library() {
    // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// extend prototype
Library.prototype.addArchive = function(archive) {
  console.log(`archive added to library - ${archive}`);
    // add archive property to instantiate object
    this.archive = archive;
    // add property to Library prototype
    Library.prototype.administrator = 'knechts';
}

// constructor for Archive object
function Archive(){
    // instance property
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();
// call addArchive on Library prototype
archiveRecord.addArchive('mariafels');

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
    console.log(`Archive domain = ${archiveRecord.domain}`);
    console.log(`Archive = ${archiveRecord.archive}`);
    console.log(`Archive admin = ${archiveRecord.administrator}`);
```

```
}

// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
    // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
    // domain property from Archive
    console.log(`Archive domain = ${archiveRecord.domain}`);
}

// instantiate another Archive object
const archiveRecord2 = new Archive();
// output instance object for second archive
console.log('Archive2 object = ', archiveRecord2);
// check if archiveRecord2 object has access to updated archive property...NO
console.log(`Archive2 = ${archiveRecord2.archive}`);
// check if archiveRecord2 object has access to updated adminstrator property...Y
console.log(`Archive2 administrator = ${archiveRecord2.administrator}`);
```

- demo - inheritance with prototype - updated

JavaScript - Prototype

configure object properties - part 1

- each object property in JS is described with a **property descriptor**
- use such descriptors to configure specific keys, e.g.
- *configurable* - boolean setting
  - *true = property's descriptor may be changed and the property deleted*
  - *false = no changes &c.*
- *enumerable* - boolean setting
  - *true = specified property will be visible in a `for-in` loop through object's properties*
- *value* - specifies value for property (default is undefined)
- *writable* - boolean setting
  - *true = the property value may be changed using an assignment*
- *get* - defines the getter function, called when we access the property
  - ***n.b.*** *can't be defined with value and writable*
- *set* - defines the setter function, used whenever an assignment is made to the property
  - ***n.b.*** *can't be defined with value and writable*
- e.g. create following property for an object

```
archive.type = 'private';
```

- `archive`
- will be *configurable*, *enumerable*, *writable*
- with a value of *private*
- *get* and *set* will currently be undefined

JavaScript - Prototype

configure object properties - part 2

- to update or modify a property configuration use built-in `Object.defineProperty()` method
- this method takes an object, which may be used to
- define or update the property
- define or update the name of the property
- define a property descriptor object
  - e.g.

```
// empty object
const archive = {};

// add properties to object
archive.name = "waldzell";
archive.type = "game";

// define property access, usage, &c.
Object.defineProperty(archive, "access", {
    configurable: false,
    enumerable: false,
    value: true,
    writable: true
});

// check access to new property
console.log(`${archive.access}, access property available on the object...`);

/*
* check we can't access new property in loop
* - for..in iterates over enumerable properties
*/
for (let property in archive) {
    // log enumerable
    console.log(`key = ${property}, value = ${archive[property]}`);
}

/*
* plain object values not iterable...
* - returns expected TyoeError - archive is not iterable
*/
```

```
for (let value of archive) {
    // value not logged...
    console.log(value);
}
```

- demo - configure object properties

JavaScript - Prototype

using ES Classes

- ES6 provides a new `class` keyword
- enables object creation and aida in inheritance
- it's *syntactic sugar* for the prototype and instantiation of objects
  - e.g.

```
// class with constructor & methods
class Archive {
  constructor(name, admin) {
    this.name = name;
      this.admin = admin;
  }
    // class method
  static access() {
    return false;
  }
    // instance method
    administrator() {
        return this.admin;
    }
}

// instantiate archive object
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with class
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// log archive name
console.log(`class archive name = ${nameCheck}`);
// call class method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
```

- demo - basic ES Class

JavaScript - Prototype

ES classes as syntactic sugar

- classes in ES6 are simply syntactic sugar for prototypes.
- a prototype implementation of previous `Archive` class, and usage...
- e.g.

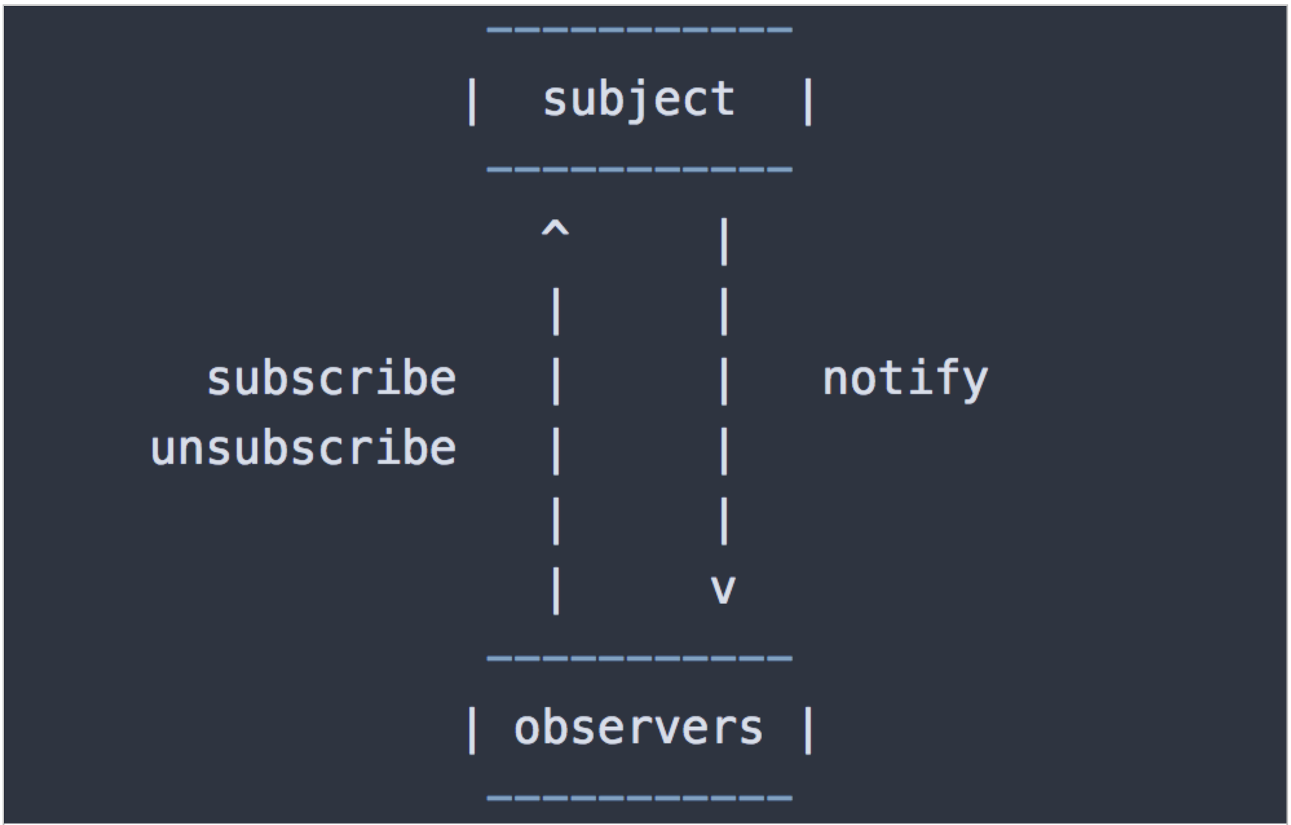
```
// constructor function
function Archive(name, admin) {
  this.name = name;
    this.admin = admin;

    // instance method
    this.administrator = function () {
        return this.admin;
    }

    // add property to constructor
    Archive.access = function() {
    return false;
    };
}

// instantiate object - pass arguments
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with ternary conditional...
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// output name check...
console.log(`prototype archive name = ${nameCheck}`);
// call constructor only method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
```

- demo - basic Prototype equivalent

Design Patterns - Observer - intro

- *observer* pattern is used to help define a *one to many* dependency between objects
- as **subject** (object) changes state
  - *any dependent **observers** (object/s) are then notified automatically*
  - *and then may update accordingly*
- managing changes in state to keep app in sync
- creating bindings that are event driven
  - *instead of standard push/pull*
- standard usage for this pattern with bindings
  - *one to many*
  - *one way*
  - *commonly event driven*

Image - Observer Pattern



Observer Pattern

Design Patterns - Observer - notifications

- observer pattern creates a model of event subscription with notifications
- benefit of this pattern
  - *tends to promote loose coupling in component design and development*
- pattern is used a lot in JavaScript based applications
  - *user events are a common example of this usage*
- pattern may also be referenced as *Pub/Sub*
  - *there are differences between these patterns - be careful...*

Design Patterns - Observer - Usage

The observer pattern includes two primary objects,

- **subject**
  - *provides interface for observers to subscribe and unsubscribe*
  - *sends notifications to observers for changes in state*
  - *maintains record of subscribed observers*
  - *e.g. a click in the UI*
- **observer**
  - *includes a function to respond to subject notifications*
  - *e.g. a handler for the click*

Design Patterns - Observer - Example

```
// constructor for subject
function Subject () {
  // keep track of observers
  this.observers = [];
}

// add subscribe to constructor prototype
Subject.prototype.subscribe = function(fn) {
  this.observers.push(fn);
};

// add unsubscribe to constructor prototype
Subject.prototype.unsubscribe = function(fn) {
  // ...
};

// add broadcast to constructor prototype
Subject.prototype.broadcast = function(status) {
  // each subscriber function called in response to state change...
  this.observers.forEach((subscriber) => subscriber(status));
};

// instantiate subject object
const domSubject = new Subject();

// subscribe & define function to call when broadcast message is sent
domSubject.subscribe((status) => {
  // check dom load
  let domCheck = status === true ? `dom loaded = ${status}` : `dom still loading.
  // log dom check
  console.log(domCheck)
});

document.addEventListener('DOMContentLoaded', () => domSubject.broadcast(true));
```

Design Patterns - Observer - Example

- Observer - Broadcast, Subscribe, & Unsubscribe

Design Patterns - Pub/Sub - intro

- variation of standard *observer* pattern is *publication and subscription*
  - *commonly known as PubSub pattern*
- popular usage in JavaScript
- *PubSub* pattern publishes a *topic* or *event* channel
- publication acts as a *mediator* or *event system* between
  - *subscriber objects wishing to receive notifications*
  - *and publisher object announcing an event*
- easy to define specific events with *event system*
- events may then pass custom arguments to a subscriber
- trying to avoid potential dependencies between objects
  - *subscriber objects and the publisher object*

Design Patterns - Pub/Sub - abstraction

- inherent to this pattern is the simple abstraction of responsibility
- publishers are unaware of nature or type of subscribers for messages
- subscribers are unaware of the specifics for a given publisher
- subscribers simply identify their interest in a given topic or event
  - *then receive notifications of updates for a given subscribed channel*
- primary difference with *observer* pattern
  - *PubSub abstracts the role of the subscriber*
- *subscriber* simply needs to handle data broadcasts by a *publisher*
- creating an abstracted *event system* between objects
  - *abstraction of concerns between publisher and subscriber*

Image - Publish/Subscribe Pattern

```
 -------------
|  publisher  |
 -------------
      |         publish topic/event
      v
 -------------
| topic/event |
|   channel   |
 -------------
      ^
      |         subscribe
 -------------
| subscriber  |
 -------------
```

PubSub Pattern

Design Patterns - Pub/Sub - benefits

- *observer* and *PubSub* patterns help developers
  - *better understanding of relationships within an app's logic and structure*
- need to identify aspects of our app that contain direct relationships
- many direct relationships may be replaced with patterns
  - *subjects and observers*
  - *publishers and observers*
- tightly coupled code can quickly create issues
  - *maintenance, scale, modification, clarity of code and logic...*
  - *semmingly minor changes may often create a cascade or waterfall effect in code*
- a known side effect of tightly couple code
  - *frequent need to mock usage &c. in testing*
  - *time consuming and error prone as app scales...*
- *PubSub* helps create smaller, loosely coupled blocks
  - *helps improve management of an app*
  - *promotes code reuse*

Design Patterns - Pub/Sub - basic example - part 1 - event system

```javascript
// constructor for pubsub object
function PubSub () {
  this.pubsub = {};
}

// publish - expects topic/event & data to send
PubSub.prototype.publish = function (topic, data) {
  // check topic exists
  if (!this.pubsub[topic]){
    console.log(`publish - no topic...`);
    return false;
  }
  // loop through pubsub for specified topic - call subscriber functions...
  this.pubsub[topic].forEach(function(subscriber) {
      subscriber(data || {});
    });
};

// subscribe - expects topic/event & function to call for publish notification
PubSub.prototype.subscribe = function (topic, fn) {
  // check topic exists
  if (!this.pubsub[topic]) {
    // create topic
    this.pubsub[topic] = [];
    console.log(`pubsub topic initialised...`);
  }
  else {
    // log output for existing topic match
    console.log(`topic already initialised...`);
  }
  // push subscriber function to specified topic
  this.pubsub[topic].push(fn);
};
```

Design Patterns - Pub/Sub - basic example - part 2 - usage

```
// basic log output
var logger = data => { console.log( `logged: ${data}` ); };

// test function for subscriber
var domUpdater = function (data) {
  document.getElementById('output').innerHTML = data;
}

// instantiate object for PubSub
const pubSub = new PubSub();

// subscriber tests
pubSub.subscribe( 'test_topic', logger );
pubSub.subscribe( 'test_topic2', domUpdater );
pubSub.subscribe( 'test_topic', logger );

// publisher tests
pubSub.publish('test_topic', 'hello subscribers of test topic...');
pubSub.publish('test_topic2', 'update notification for test topic2...');
```

- Demo - Pub/Sub

JavaScript - modular design

ES Module pattern - intro

- simpler and easier to work with than CommonJS
  - *in most examples...*
- JavaScript `strict` mode is enabled by default
- `strict` mode helps with language usage - check for poor usage
  - *stops hoisting of variables*
  - *variables must be declared*
  - *function parameters must have unique name*
  - *assignment to read-only properties throws errors*
  - ...
- modules are exported with `export` statements
- modules are imported with `import` statements

JavaScript - modular design

ES Module pattern - `export` statements

- ES6 modules are individual files
  - *expose an API using `export` statements*
- declarations are scoped to the local module
- e.g. variables declared inside a module
  - *not available to other modules*
  - *need to be explicitly exported in module API*
  - *need to be imported for usage in another module*
- export statements may only be added to *top-level* of a module
  - *e.g. not in function expression &c.*
- cannot dynamically define and expose API using methods
  - *unlike CommonJS module system - Node.js &c.*

JavaScript - modular design

ES Module pattern - `export default`

- common option is to export a default binding, e.g.

```javascript
export default `hello world`
```

```javascript
export default {
    name: 'Alice',
    place: 'Wonderland'
}
```

```javascript
export default [
    'Alice', 'Wonderland'
]
```

```javascript
export default function name() {
    ...
}
```

JavaScript - modular design

ES Module pattern - bindings

- ES modules export **bindings**
  - *not values or references*
- e.g. an export of `count` variable from a module
  - `count` *is exported as a binding*
  - *export is bound to* `count` *variable in the module*
  - *value is subject to changes of* `count` *in module*
- offers flexibility to exported API
  - *e.g.* `count` *might originally be bound to an object*
  - *then changed to an array...*
- other modules consuming this export
  - *they would see change as* `count` *is modified*
  - *modified in module and exported...*
- **n.b.** take care with this usage pattern
  - *useful for counters, logs &c.*
  - *can cause issues with API usage for a module*

JavaScript - modular design

ES Module pattern - named export

- we may define bindings for export
- instead of assigning properties to implicit export object
  - e.g.

```
export let counter = 0
export const count = () => counter++
```

- cannot refactor this example for named export
  - *syntax error will be thrown*
  - e.g.

```
let counter = 0
const count = () => counter++
export counter // this will return syntax error
export count
```

- rigid syntax helps with analysis, parsing
  - *static analysis for ES modules*

JavaScript - modular design

ES Module pattern - export lists

- lists provide a useful solution to previous refactor issue
- syntax for list export easy to parse
- export lists of named *top-level* declarations
  - *variables &c.*
- e.g.

```
let counter = 0
const count = () => counter++
export { counter, count }
```

- also rename binding for export, e.g.

```
let counter = 0
const count = () => counter++
export { counter, count as increment }
```

- define `default` with export list, e.g.

```
let counter = 0
const count = () => counter++
export { counter as default, count as increment }
```

JavaScript - modular design

ES Module pattern - `export from ...`

- expose another module's API using `export from...`
  - *i.e. a kind of pass through...*
- e.g.

```javascript
export { increment } from './myCounter.js'
```

- bindings are not imported into module's local scope
- current module acts as conduit, passing bindings along export/import chain...
- module does not gain direct access to `export from ...` bindings
  - *e.g. if we call `increment` it will throw a `ReferenceError`*
- aliases are also possible for bindings with `export from...`
  - e.g.

```javascript
export { increment as addition } from './myCounter.js'
```

JavaScript - modular design

ES Module pattern - `import` statements

- use `import` to load another module
- `import` statement are only allowed in top level of module definition
  - *same as `export` statements*
  - *helps compilers simplify module loading &c.*
- import default exports
  - *give default export a name as it is imported*
  - e.g.

```
import counter from './myCounter.js'
```

- importing binding to `counter`
- syntax different from declaring a JS variable

JavaScript - modular design

ES Module pattern - `import` named exports

- also imported any named exports
  - *import more than just default exports*
- named import is wrapped in braces
  - e.g.

```
import { increment } from './myCounter.js'
```

- also import multiple named exports
  - e.g.

```
import { increment, decrement } from './myCounter.js'
```

- import aliases are also supported
  - e.g.

```
import { increment as addition } from './myCounter.js'
```

- combine default with named
  - e.g.

```
import counter, { increment } from './myCounter.js'
```

JavaScript - modular design

ES Module pattern - `import` with wildcard

- we may also import using the *wildcard* operator
  - e.g.

```
import * as counter from './myCounter.js'
counter.increment()
```

- name for wildcard import acts like object for module
- call module exports on wildcard


```
import * as counter from './myCounter.js'
counter.increment()
```


- common pattern for working with libraries &c.

JavaScript - modular design

ES Module pattern - benefits & practical usage

- offers ability to explicitly publish an API
  - *keeps module content local unless explicitly exported*
- similar function to *getters* and *setters*
  - *explicit way in and out of modules*
  - *explicit options for reading and updating values...*
- code becomes simpler to write and manage
  - *module offers encapsulation of code*
- import binding to variable, function &c.
  - *then use it as normal...*
- removes need for encapsulation in main JS code
  - *e.g. with patterns such as IIFE...*
- *n.b.* need to be careful how we use modules
  - *e.g. priority for access, security, testing &c.*
  - *all now moved to individual modules...*

JavaScript - modular design

ES Module pattern - Lib structure

- Modules in JavaScript are not a new concept
  - *e.g. CommonJS is a popular option for modular development with Node.js*
- a built-in option for plain JavaScript, *ES Modules*.
- use this option to develop and structure custom module libraries
- e.g.
  - *abstract utility modules*
  - *custom draw libraries*
  - *game renderers*
  - ...

JavaScript - modular design

ES Module pattern - JS library

- an example JS library - define the following directory structure

```
.
|-- lib
|   |-- spire
|   |   |-- helpers
|   |   |   |__ log.js
|   |   |__ spire.js
|__ main.js
| ...
```

- `lib` directory contains custom JS libraries, which may then be imported for use within an app
- for app usage, we might structure it as follows

```
.
|-- lib
|   |-- spire
|   |   |-- helpers
|   |   |   |__ log.js
|   |   |__ spire.js
|__ index.html
|__ main.js
| ...
```

JavaScript - modular design

ES Module pattern - JS library - `main.js`

- `main.js` file is loaded from the `index.html` file
  - *acts as the loader file for JS in an example app*
- also import example *Spire* JS library into an app using this main loader file, e.g.

```
import Spire from './lib/spire/spire.js';
```

- `Spire` object is the access point to the exported methods and variables for custom JS library

JavaScript - modular design

ES Module pattern - JS library - basic usage

- a custom JS library may then be accessed using this `Spire` object
- e.g. we might call a method from the library

```
const greeting = 'greetings from the planet Earth';
// basic log to console
Spire.log(`${greeting}...we wish you well`);
```

- custom method `log()` provides a reusable method
  - *e.g. use for various logging options in the application*
- might also call the following method using the same pattern

```
Spire.dir({'name': 'test dir logger...'});
```

JavaScript - modular design

ES Module pattern - JS library - module usage

- sample usage might include such *helpers*
  - *we may package in the directory `spire/helpers/`*
  - *e.g., we currently have a `log.js` module for various custom loggers*

```
// basic logger to console
function log(value, ...values) {
  const logValue = console.log(value, ...values);

  return logValue;
}

// directory logger to console
function dir(value, ...values) {
  const dirValue = console.dir(value, ...values);

  return dirValue;
}
```

- we may then simply export these methods from the `log.js` module, e.g.

```
export {
    log,
    dir
}
```

- interface for this module has now been defined relative to the above exported modules

JavaScript - modular design

ES Module pattern - JS library - import modules

- import this module
  - *allow a module to use these exported methods*
  - *interact with the exposed interface*
- as part of the JS library structure we may define
  - *a root module for organising a unified interface for the overall library*
- e.g. use the module `spire.js` to import required modules and their interfaces

```
import * as loggers from './helpers/log.js';
```

- then define a `Spire` object for the overall library, e.g.

```
const Spire = {
    log: loggers.log,
    dir: loggers.dir,
}
```

- this is then exported as the general interface for the Spire JS library, e.g.

```
export default Spire;
```

Responsive Design & Development - Modular Designs

Fun Exercise

Three responsive designs,

- Modular designs - http://linode4.cs.luc.edu/teaching/cs/demos/424/gifs/modular/
  - *Home Design*
  - *Reminders*
  - *Watches*

For each design, consider the following

- define perceived modules for each app
  - *where might you use a module?*
- what type of modules can you define in each app?
  - *e.g. logical, structural, design, performance...*
- from a developer perspective
  - *consider primary modular groupings*
  - *does each module purpose help with testing?*
  - *can each module be decoupled from app?*
    - e.g. test and use outside of current app...

~ 10 minutes

Data visualisation

intro - part 1

- data visualisation - study of how to visually communicate and analyse data
- covers many disparate aspects
  - *including infographics, exploratory tools, dashboards...*
- already some notable definitions of data visualisation
- one of the better known examples,

> *"Data visualisation is the representation and presentation of data that exploits our visual perception in order to amplify cognition."*
>
> *(Kirk, A. "Data Visualisation: A successful design process." Packt Publishing. 2012.)*

- several variants of this general theme exist
  - *the underlying premise remains the same*
- simply, data visualisation is a visual representation of the underlying data
- visualisation aims to impart a better understanding of this data
  - *by association, its relevant context*

Data visualisation

intro - part 2

- an inherent flip-side to data visualisation
- without a correct understanding of its application
  - *it can simply impart a false perception, and understanding, on the dataset*
- run the risk of creating many examples of standard **areal unit** problem
  - *perception often based on creator's base standard and potential bias*
- inherently good at seeing what we want to see
- without due care and attention visualisations may provide false summations of the data

Data visualisation

types - part 1

- many different ways to visualise datasets
  - *many ways to customise a standard infographic*
- some standard examples that allow us to consider the nature of visualisations
  - *infographics*
  - *exploratory visualisations*
  - *dashboards*
- perceived that data visualisation is simply a variation between
  - *infographics, exploratory tools, charts, and some data art*

  1. infographics
     - *well suited for representing large datasets of contextual information*
     - *often used in projects more inclined to exploratory data analysis,*
     - *tend to be more interactive for the user*
     - *data science can perceive infographics as improper data visualisation because*
     - *they are designed to guide a user through a story*
     - *the main facts are often already highlighted*
     - ***NB:** such classifications often still only provide tangible reference points*

Data visualisation

types - part 2

2. exploratory visualisations
   - *more interested in the provision of tools to explore and interpret datasets*
   - *visualisations can be represented either static or interactive*
   - *from a user perspective these charts can be viewed*
   - *either carefully*
   - *simply become interactive representations*
   - *both perspectives help a user discover new and interesting concepts*
   - *interactivity may include*
   - *option for the user to filter the dataset*
   - *interact with the visualisation via manipulation of the data*
   - *modify the resultant information represented from the data*
   - *often perceived as more objective and data oriented than other forms*
3. dashboards
   - *dense displays of charts*
   - *represent and understand a given issue, domain...*
   - *as quickly and effectively as possible*
   - *examples of dashboards*
   - *display of server logs, website users, business data...*

Data visualisation

Dashboards - intro

- dashboards are dense displays of charts
- allow us to represent and understand the key **metrics** of a given issue
  - *as quickly and effective as possible*
  - *eg: consider display of server logs, website users, and business data...*
- one definition of a dashboard is as follows,

> *"A dashboard is a visual display of the most important information needed to achieve one or more objective; consolidated and arranged on a single screen so the information can be monitored at a glance."*
>
> *Few, Stephen. Information Dashboard Design: The Effective Visual Communication of Data. O'Reilly Media. 2006.*

- dashboards are visual displays of information
  - *can contain text elements*
  - *primarily a visual display of data rendered as meaningful information*

Data visualisation

Dashboards - intro

- information needs to be consumed quickly
- often simply no available time to read long annotations or repeatedly click controls
- information needs to be visible, and ready to be consumed
- dashboards are normally presented as a complementary environment
- an option to other tools and analytical/exploratory options
- design issues presented by dashboards include effective distribution of available space
- compact charts that permit quick data retrieval are normally preferred
- dashboards should be designed with a purpose in mind
- generalised information within a dashboard is rarely useful
- display most important information necessary to achieve their defined purpose
- a dashboard becomes a central view for collated data
- represented as meaningful information

Data visualisation

Dashboards - good practices

- to help promote our information
  - *need to design the dashboard to fully exploit available screen space*
- need to use this space to help users absorb as much information as possible
- some visual elements more easily perceived and absorbed by users than others
- some naturally convey and communicate information more effectively than others
- such attributes are known as **pre-attentive attributes of visual perception**
- for example,
  - *colour*
  - *form*
  - *position*

Data visualisation

Dashboards - visual perception

- pre-attentive attributes of visual perception

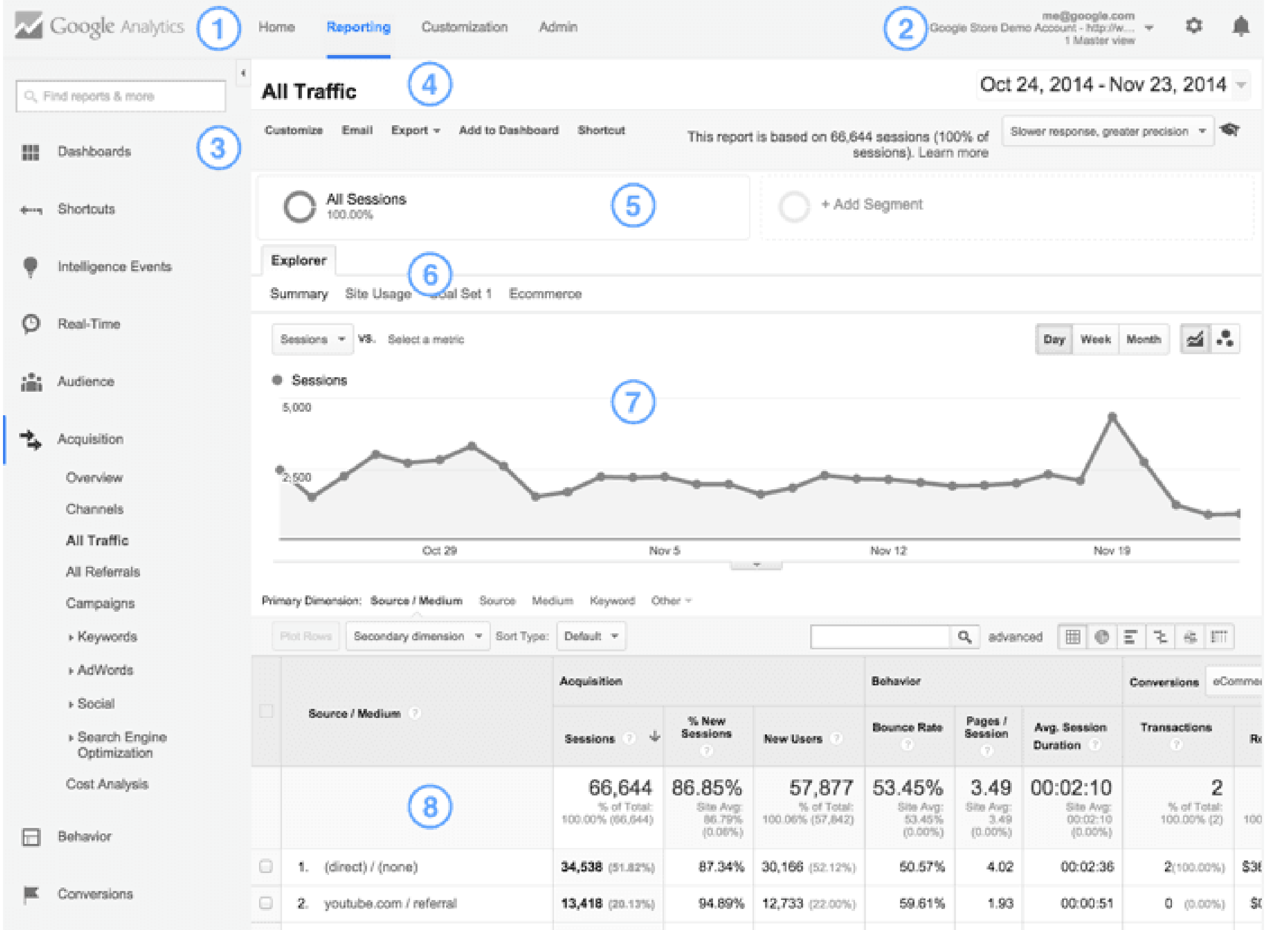
1. *Colour*
   - *many different colour models currently available*
   - *most useful relevant to dashboard design is the HSL model*
   - *this model describes colour in terms of three attributes*
     - *hue*
     - *saturation*
     - *lightness*
   - *perception of colour often depends upon context*
2. *Form*
   - *correct use of length, width, and general size can convey quantitative dimensions*
   - *each with varying degrees of precision*
   - *use the Laws of Prägnanz to manipulate groups of similar shapes and designs*
   - *thereby easily grouping like data and information for the user*
3. *Position*
   - *relative positioning of elements helps communicate dashboard information*
   - *laws of Prägnanz teach us*
   - *position can often infer a perception of relationship and similarity*
   - *higher items are often perceived as being better*
   - *items on the left of the screen traditionally seen first by a western user*

Data visualisation

Building a dashboard

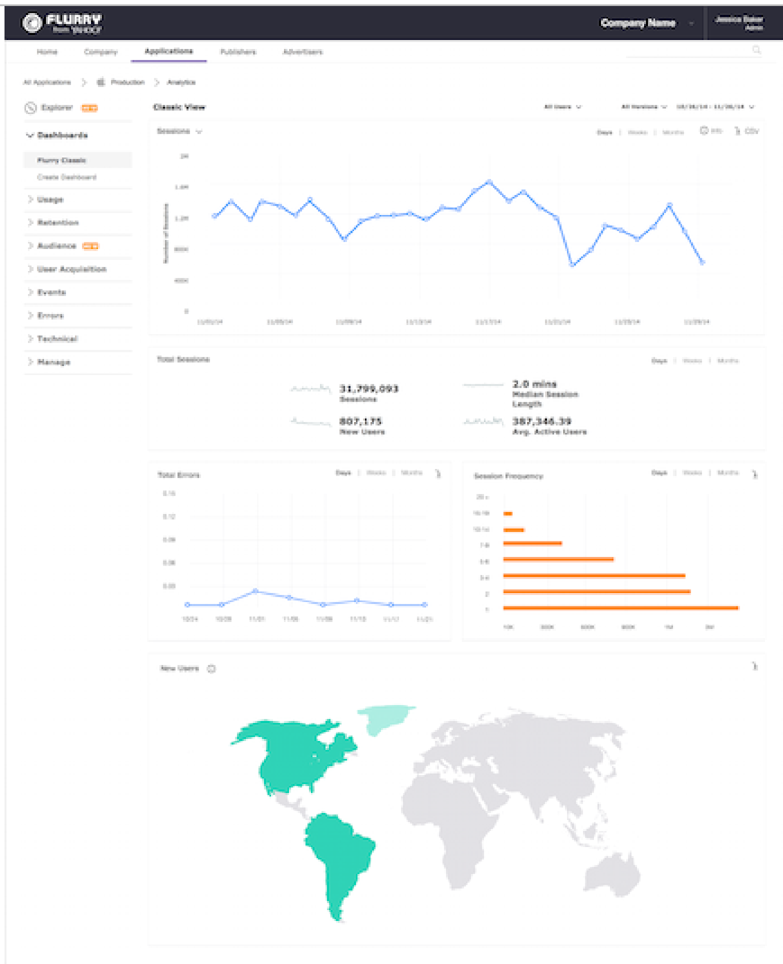
- need to clearly determine the questions that need to be answered
    - *given the information collated and presented within the dashboard*
- need to ensure that any problems can be detected on time
- be certain why we actually need a dashboard for the current dataset
- then begin to collect the requisite data to help us answer such questions
    - *data can be sourced from multiple, disparate datasets*
- chosen visualisations help us tell this story more effectively
- present it in a manner appealing to our users
- need to consider information visualisations familiar to our users
    - *helps reduce any potential user's cognitive overload*
- carefully consider organisation of data and information
- organise the data into logical units of information
    - *helps present dashboard information in a meaningful manner*
- dashboard sections should be organised
    - *to help highlight and detect any underlying or prevailing issues*
    - *then present them to the user*

Image - Google Analytics



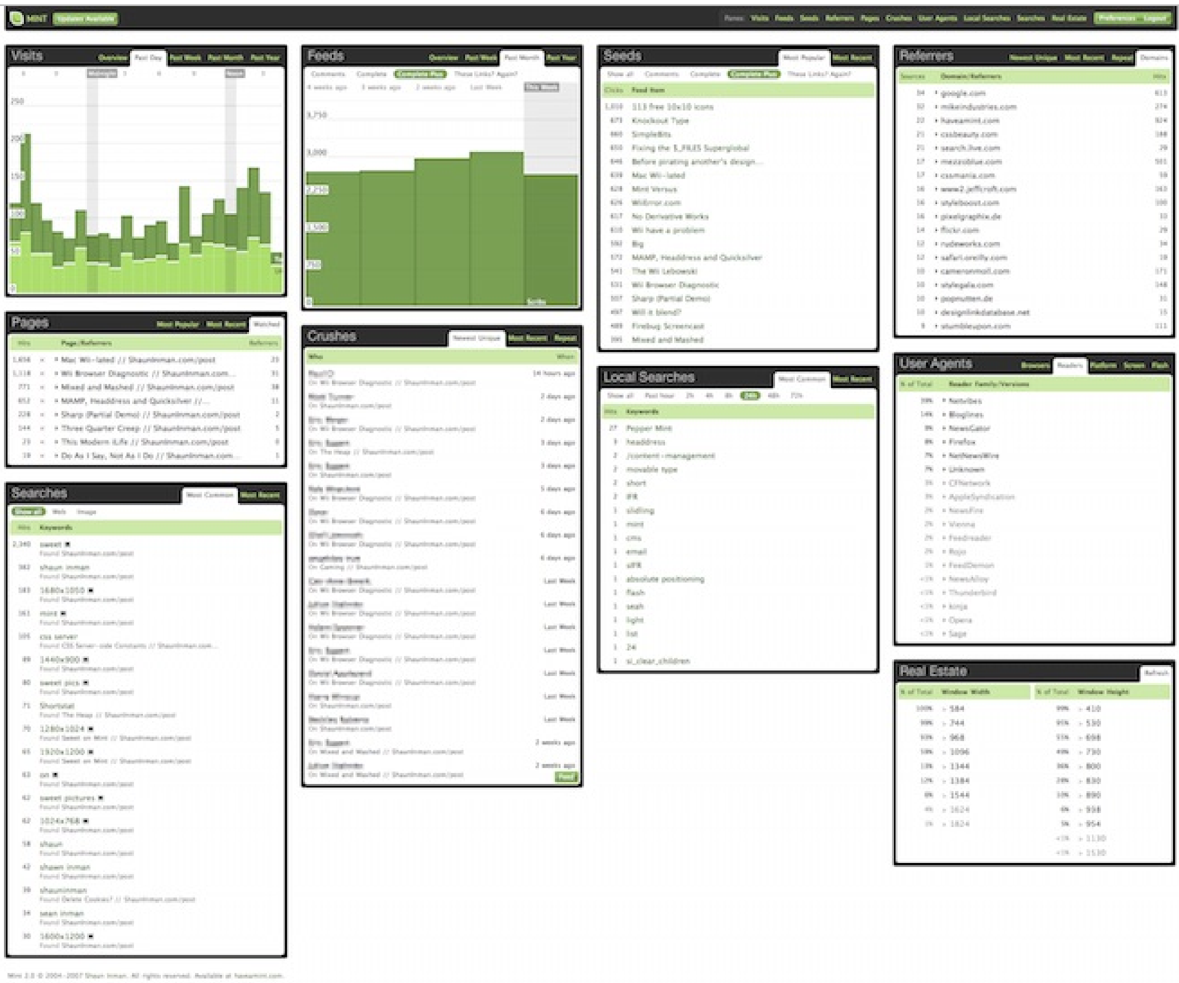
Dashboard - Google Analytics

Image - Yahoo Flurry



[Dashboard - Yahoo Flurry]

Image - Mint



Dashboard - Mint

Data visualisation - D3

Intro - part 1

- D3 is a custom JavaScript library
  - *designed for the manipulation of data centric documents*
  - *uses a custom library with HTML, CSS, and SVG*
  - *creates graphically rich, informative documents for the presentation of data*
- D3 uses a data-driven approach to manipulate the DOM
- Setup and configuration of D3 is straightforward
  - *most involved aspect is the configuration of a web server*
- D3.js works with standard HTML files
  - *requires a web server capable of parsing and rendering HTML...*
- to parse D3 correctly we need
  - *UTF-8 encoding reference in a meta element in the head section of our file*
  - *reference D3 file, CDN in standard script element in HTML*

Data visualisation - D3

intro - part 2

- D3 Wiki describes the underlying functional concepts as follows,

> *D3's functional style allows code reuse through a diverse collection of components and plugins.*
>
> *D3 Wiki*

- in JS, functions are objects
  - *as with other objects, a function is a collection of a name and value pair*
- real difference between a function object and a regular object
  - *a function can be invoked, and associated, with two hidden properties*
  - *include a function context and function code*
- variable resolution in D3 relies on variable searching being performed locally first
- if a variable declaration is not found
  - *search will continue to the parent object*
  - *continue recursively to the next static parent*
  - *until it reaches global variable definition*
  - *if not found, a reference error will be generated for this variable*
- important to keep this static scoping rule in mind when dealing with D3

Data visualisation - D3

Data Intro - part 1

- Data is structured information with an inherent perceived potential for meaning
- consider data relative to D3
  - *need to know how data can be represented*
  - *both in programming constructs and its associated visual metaphor*
- what is the basic difference between data and information?

> *Data are raw facts. The word raw indicates that the facts have not yet been processed >>> to reveal their meaning...Information is the result of processing raw data to reveal >>> its meaning.*
>
> *Rob, Morris, and Coronel. 2009*

- a general concept of data and information
- consider them relative to visualisation, impart a richer interpretation
- information, in this context, is no longer
  - *the simple result of processed raw data or facts*
  - *it becomes a visual metaphor of the facts*
- same data set can generate any number of visualisations
  - *may lay equal claim in terms of its validity*
- visualisation is communicating creator's insight into data...

Data visualisation - D3

Data Intro - part 2

- relative to development for visualisation
  - *data will often be stored simply in a text or binary format*
- not simply textual data, can also include data representing
  - *images, audio, video, streams, archives, models...*
- for D3 this concept may often simply be restricted to
  - *textual data, or text-based data...*
  - *any data represented as a series of numbers and strings containing alpha numeric characters*
- suitable textual data for use with D3
  - *text stored as a comma-separated value file (.csv)*
  - *JSON document (.json)*
  - *plain text file (.txt)*
- data can then be *bound* to elements within the DOM of a page using D3
  - *inherent pattern for D3*

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

- in D3, connection between data and its visual representation
  - *usually referred to as the **enter-update-exit** pattern*
- concept is starkly different from the standard imperative programming style
- pattern includes
  - *enter mode*
  - *update mode*
  - *exit mode*

Data visualisation - D3

Data Intro - Enter-Update-Exit Pattern

Enter mode

- `enter()` function returns all specified data that not yet represented in visual domain
- standard modifier function chained to a selection method
  - *create new visual elements representing given data elements*
  - *eg: keep updating an array, and outputting new data bound to elements*

Update mode

- `selection.data(data)` function on a given selection
  - *establishes connection between data domain and visual domain*
- returned result of intersection of data and visual will be a **data-bound** selection
- now invoke a modifier function on this newly created selection
  - *update all existing elements*
  - *this is what we mean by an **update** mode*

Exit mode

- invoke `selection.data(data).exit` function on a data-bound selection
  - *function computes new selection*
  - *contains all visual elements no longer associated with any valid data element*
- eg: create a bar chart with 25 data points
  - *then update it to 20, so we now have 5 left over*
  - ***exit mode** can now remove excess elements for 5 spare data points*

Data visualisation - D3

Data Intro - binding data - part 1

- consider standard patterns for working with data
- we can iterate through an array, and then bind the data to an element
  - *most common option in D3 is to use the* ***enter-update-exit*** *pattern*
- use same basic pattern for binding object literals as data
- to access our data we call the required attribute of the supplied data

```
var data = [
    {height: 10, width: 20},
    {height: 15, width: 25}
];

function (d) {
    return (d.width) + "px";
}
```

- then access the **height** attribute per object in the same manner
- we can also bind functions as data
  - *D3 allows functions to be treated as data...*

Data visualisation - D3

Data Intro - binding data - part 2

- D3 enables us to bind data to elements in the DOM
  - *associating data to specific elements*
  - *allows us to reference those values later*
  - *so that we can apply required mapping rules*
- use D3's `selection.data()` method to bind our data to DOM elements
  - *we obviously need some data to bind, and a selection of DOM elements*
- D3 is particularly flexible with data
  - *happily accepts various types*
- D3 also has a built-in function to handle loading JSON data

```
d3.json("testdata.json", function(json) {
    console.log(json); //do something with the json...
});
```

Data visualisation - D3

Data Intro - working with arrays - options

- min and max = return the min and max values in the passed array

```
d3.select("#output").text(d3.min(ourArray));
d3.select("#output").text(d3.max(ourArray));
```

- extent = retrieves both the smallest and largest values in the the passed array

```
d3.select("#output").text(d3.extent(ourArray));
```

- sum

```
d3.select("#output").text(d3.sum(ourArray));
```

- median

```
d3.select("#output").text(d3.median(ourArray));
```

- mean

```
d3.select("#output").text(d3.mean(ourArray));
```

- asc and desc

```
d3.select("#output").text(ourArray.sort(d3.ascending));
d3.select("#output").text(ourArray.sort(d3.descending));
```

- & many more...

Data visualisation - D3

Data Intro - working with arrays - nest

- D3's nest function used to build an algorithm
  - *transforms a flat array data structure into a hierarchical nested structure*
- function can be configured using the key function chained to **nest**
- nesting allows elements in an array to be grouped into a hierarchical tree structure
  - *similar in concept to the group by option in SQL*
  - ***nest** allows multiple levels of grouping*
  - *result is a tree rather than a flat table*
- levels in the tree are defined by the *key* function
- leaf nodes of the tree can be sorted by value
- internal nodes of the tree can be sorted by key

Data visualisation - D3

Selections - intro

- **Selection** is one of the key tasks required within D3 to manipulate and visualise our data
- simply allows us to target certain visual elements on a given page
- Selector support is now standardised upon the W3C specification for the Selector API
  - *supported by all of the modern web browsers*
  - *its limitations are particularly noticeable for work with visualising data*
- Selector API only provides support for selector and not selection
  - *able to select an element in the document*
  - *to manipulate or modify its data we need to implement a standard loop etc*
- D3 introduced its own selection API to address these issues and perceived shortcomings
  - *ability to select elements by ID or class, its attributes, set element IDs and class, and so on...*

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 <div> 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 <div> 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 <div> element

Data visualisation - D3

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 <div> 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 para
```

- 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(
```

- 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*

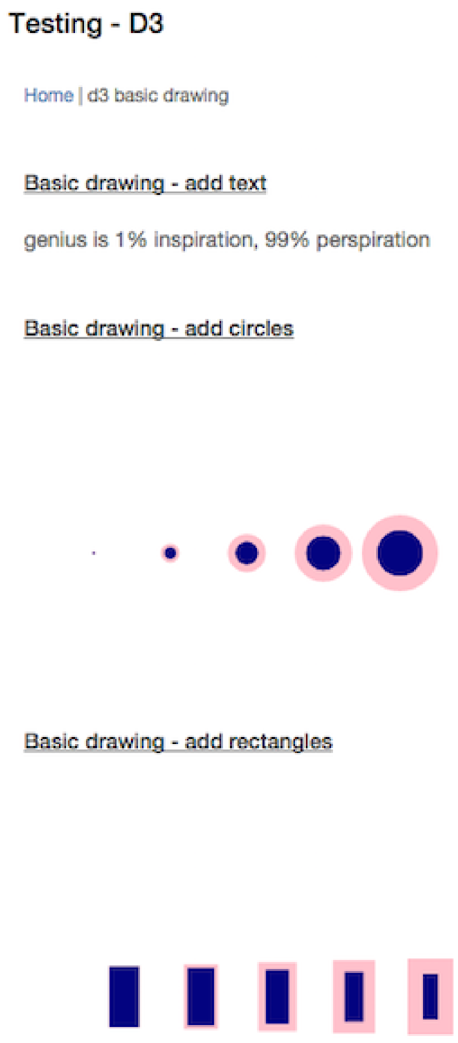
2. create SVG

- 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);
```

Data visualisation - D3

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

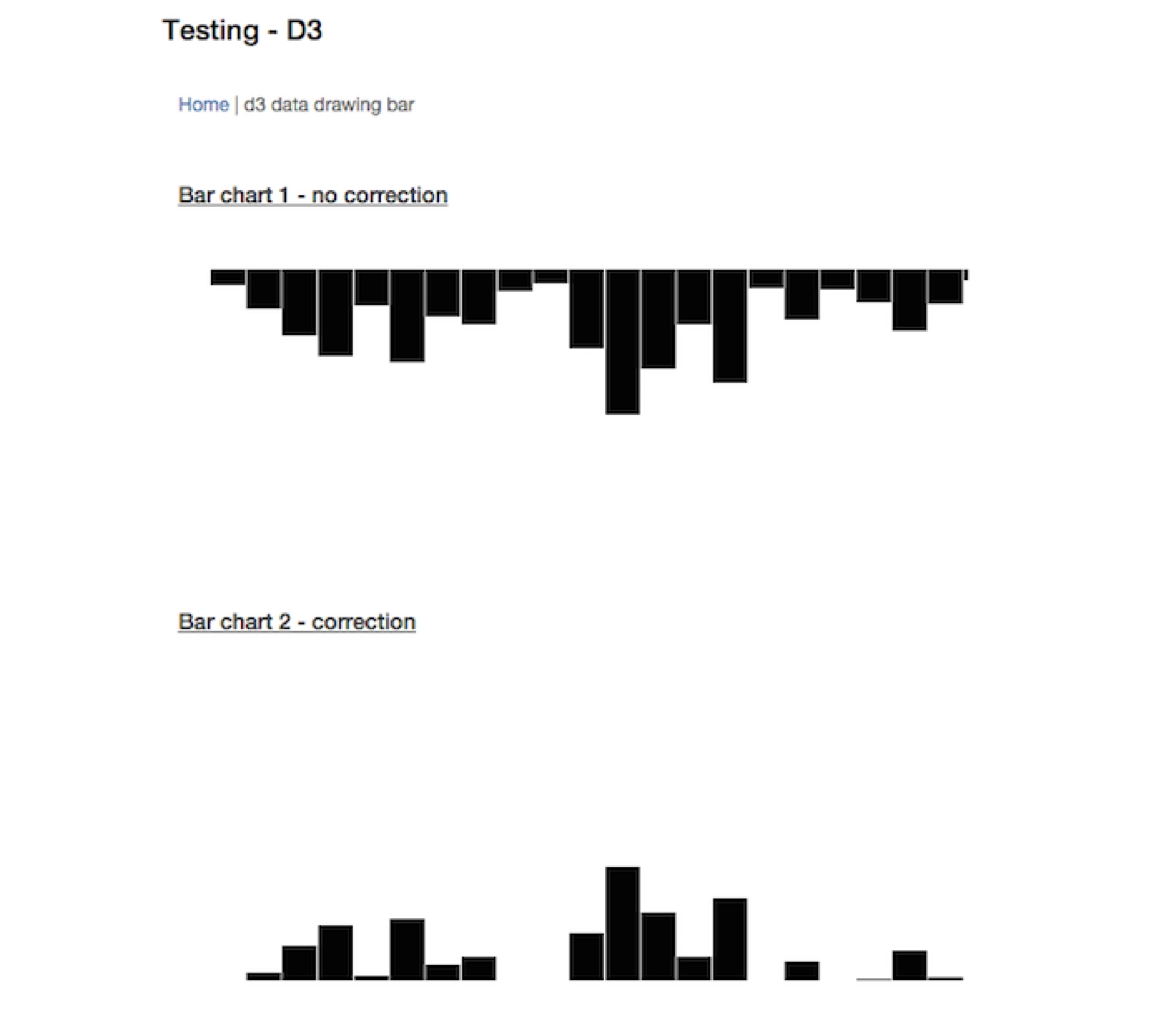
```
.attr("height", function(d) {
    return d * 4;
});
```

- 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

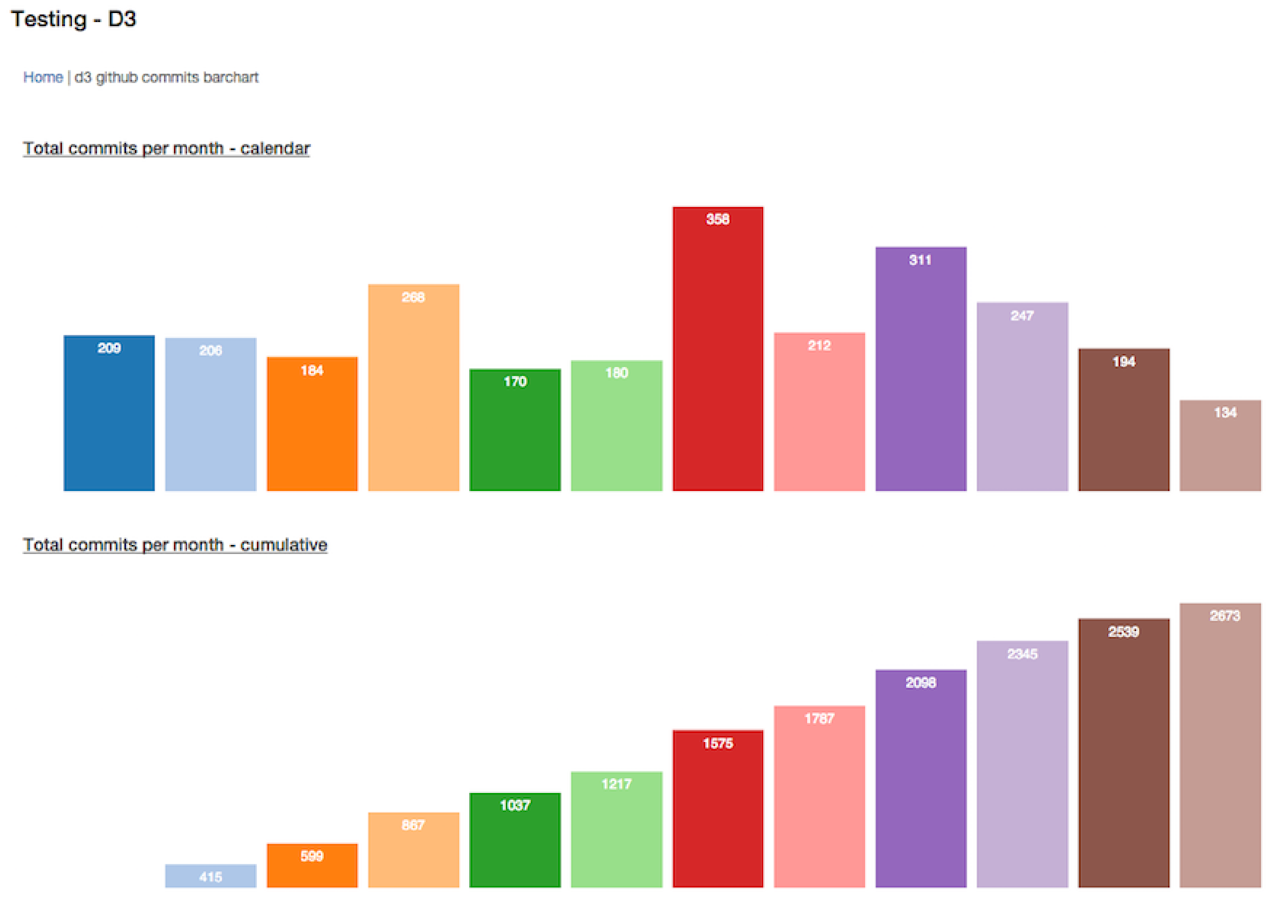
```
.attr("x", function(d, i) {
    return i * (w / dataset.length);
})
.attr("y", function(d, i) {
    return h - (d * 4);
});
```

- 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*

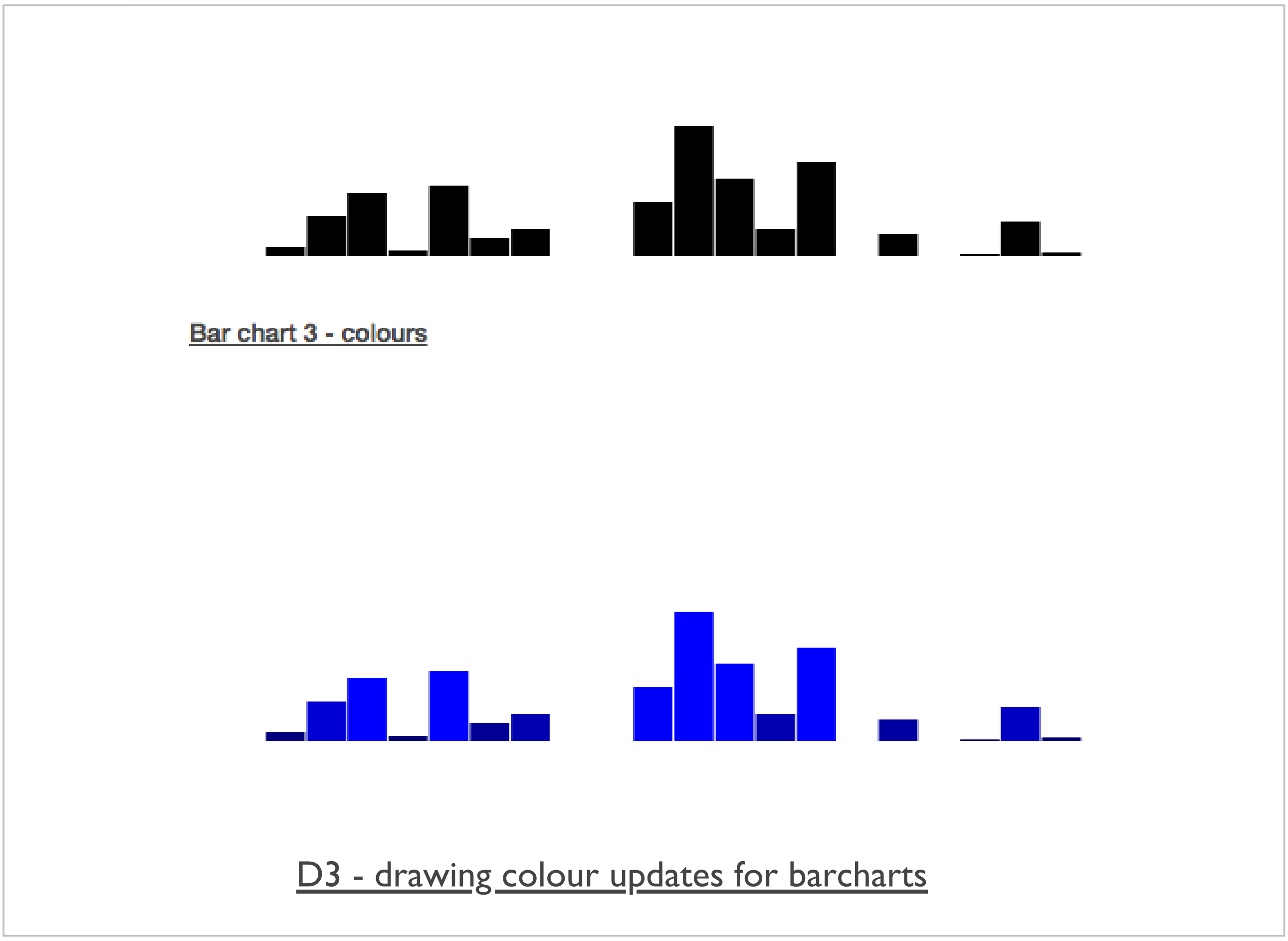
```
d3.select("p")
    .on("click", function() {

    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...*

Data visualisation - D3

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();
```

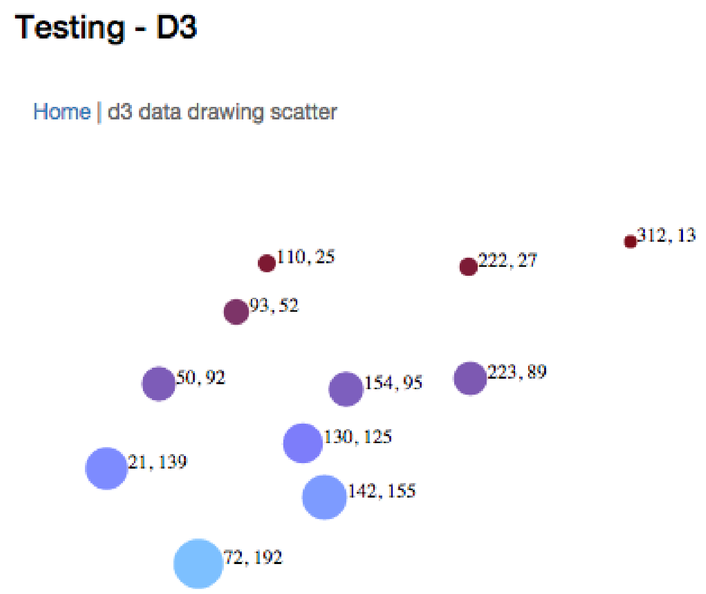
- `remove()` is a special transition method that awaits until transition is complete
- then deletes element from DOM forever
  - *to get it back, we'd need to rebuild it again*

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 stroe 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*

Data visualisation - D3

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) + ')';
});
```

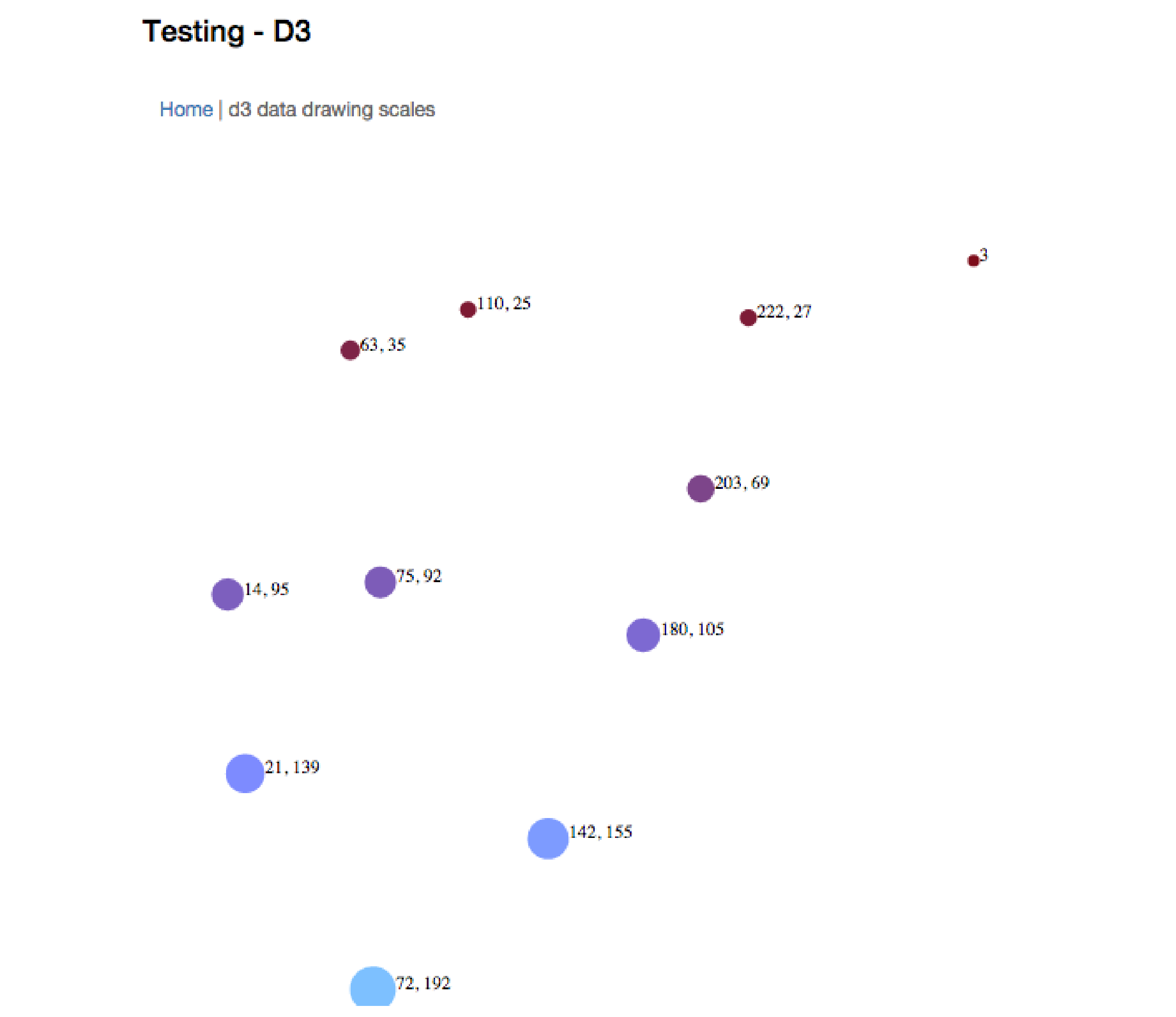
Data visualisation - D3

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*

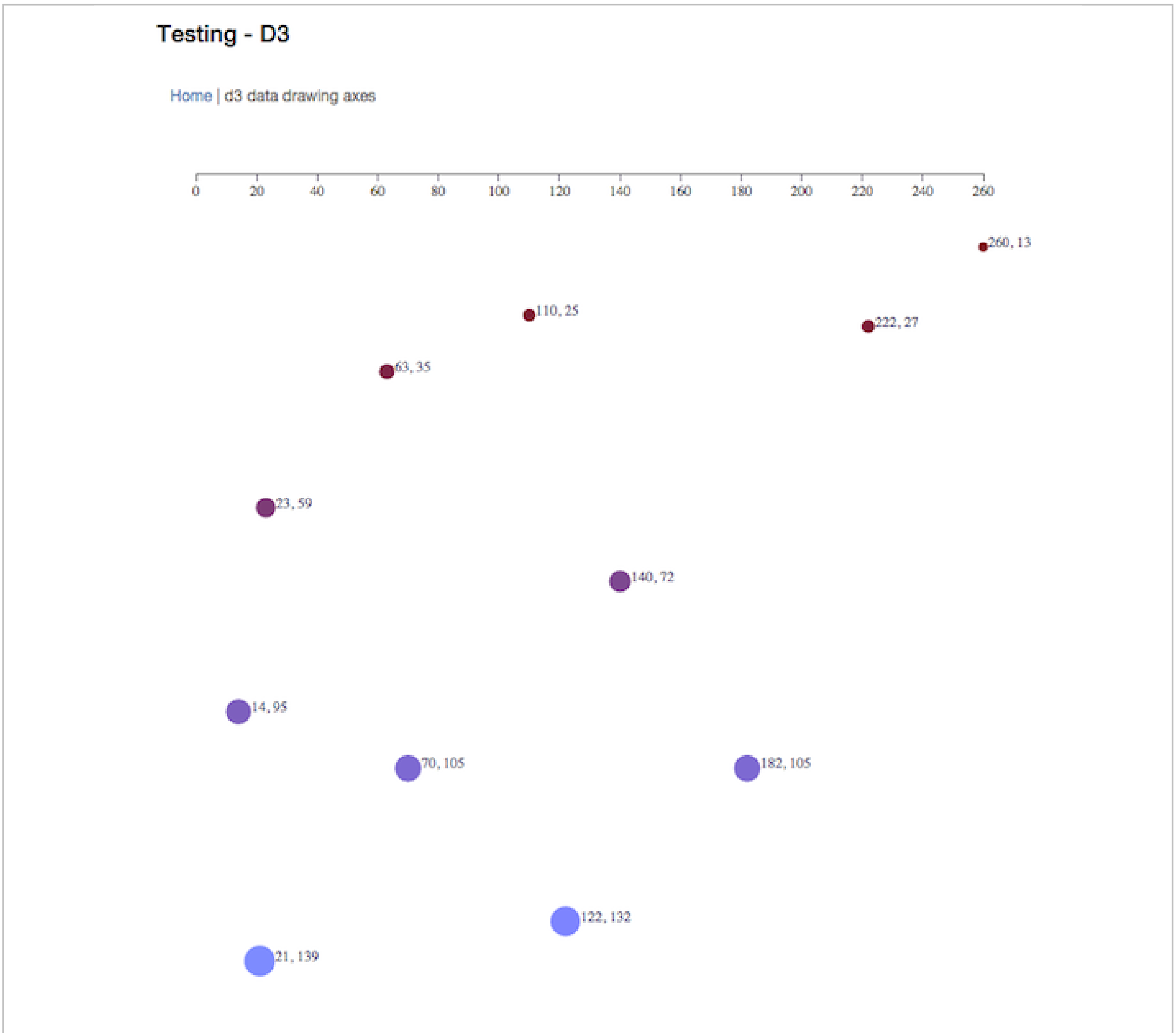
```
d3.max(dataset, function(d) {
    return d[0];
});
```

- 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



D3 - add axis

Data visualisation - D3

Drawing - SVG - adding dynamic scales

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

general examples

Sample dashboards and visualisations

- gaming dashboard
- schools and education
- students and grades
- D3 examples

Example datasets

- Chicago data portal

Article example

- dashboard designs
- replace jQuery with D3

Data Visualisation

projects examples

A few examples from recent projects,

- GitHub API tests
- check JSON return
- early test examples
- metrics test examples

Demos

- D3.js
- D3 basic elements
- Drawing with SVG
- Drawing with SVG - barcharts
- Drawing with SVG - barcharts, colour, and text labels
- JavaScript - Patterns
- Observer - Broadcast, Subscribe, & Unsubscribe
- Pub/Sub
- JavaScript - Prototype
- basic prototype
- basic set prototype
- basic prototype object
- basic prototype object properties
- basic prototype dynamic
- basic constructor check
- inheritance with prototype
- inheritance with prototype - updated
- configure object properties
- basic ES Class
- basic Prototype equivalent

Resources

- MDN
- Object Prototypes
- Inheritance and the prototype chain
- D3.js
- D3 - API reference
- D3 - Easing
- D3 - Scales
- D3 - Wiki
- Kirk, A. *Data Visualisation: A successful design process*. Packt Publishing. 2012.