Comp 324/424 - Client-side Web Design

Spring Semester 2019 - Week 6

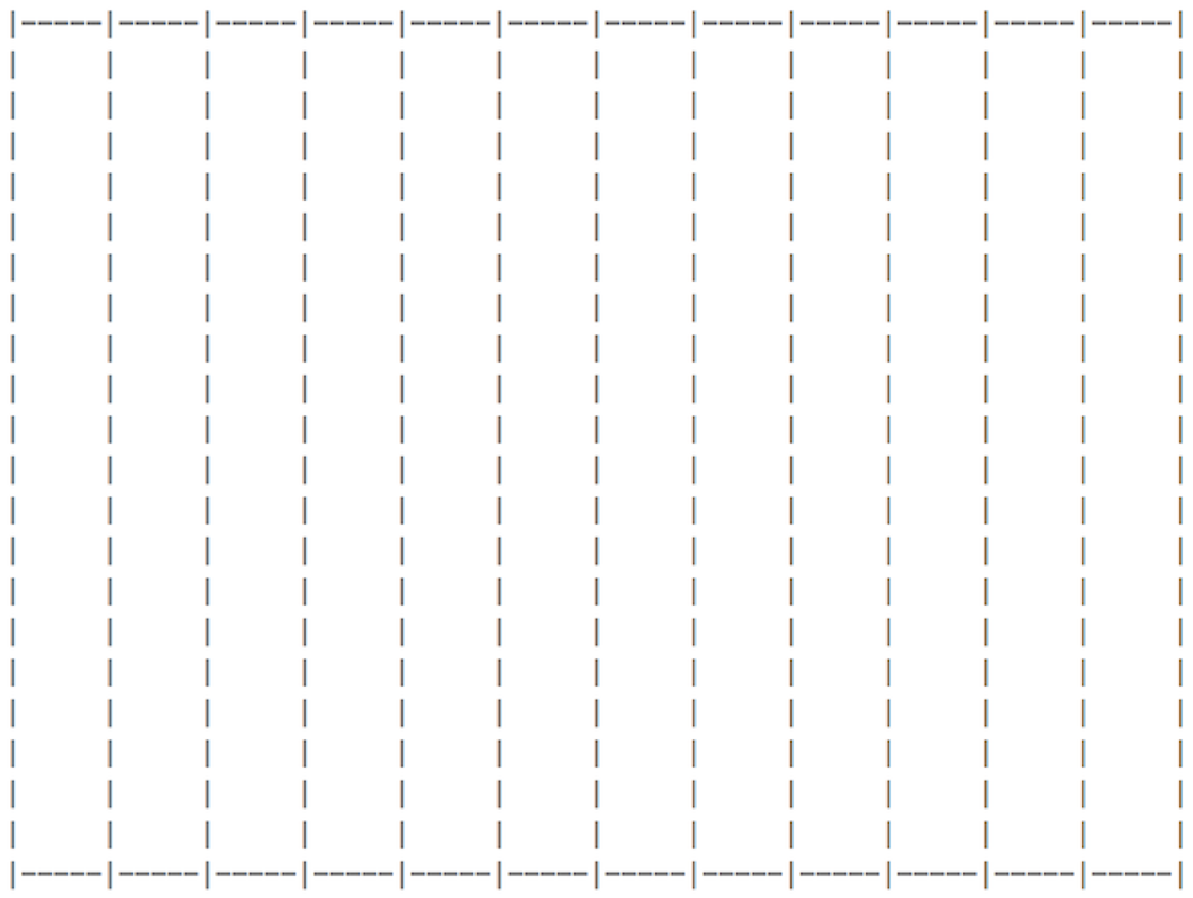
Dr Nick Hayward

CSS grid layout - part 1

intro

- grid designs for page layout, components...
  - *increasingly popular over the last few years*
  - *useful for creating responsive designs*
- quick and easy to layout a scaffolding framework for our structured content
- create boxes for our content
  - *then position them within our grid layout*
- content can be stacked in a horizontal and vertical manner
  - *creating most efficient layout for needs of a given application*
- another benefit of CSS grids is that they are framework and project agnostic
  - *thereby enabling easy transfer from one to another*
- concept is based upon a set number of columns per page with a width of 100%
- columns will increase and decrease relative to the size of the browser window
- also set break points in our styles
  - *helps to customise a layout relative to screen sizes, devices, aspect ratios...*
  - *helps us differentiate between desktop and mobile viewers*

Image - Grid Layout



Grid Layout - Columns and rows

CSS grid layout - part 2

grid.css

- build a grid based upon 12 columns
  - *other options with fewer columns as well*
- tend to keep our grid CSS separate from the rest of the site
  - *maintain a CSS file just for the grid layout*
- helps abstract the layout from the remaining styles
  - *makes it easier to reuse the grid styles with another site or application*
- add a link to this new stylesheet in the `head` element of our pages

```
<link rel="stylesheet" type="text/css" href="assets/styles/grid.css">
```

or

```
<link rel="stylesheet" href="assets/styles/grid.css">
```

- ensure padding and borders are included in total widths and heights for an element
  - *reset `box-sizing` property to include the `border-box`*
  - *resetting box model to ensure padding and borders are included*

```
* {
box-sizing: border-box;
}
```

CSS grid layout - example - part 3

grid.css

- set some widths for our columns, 12 in total
  - *each representing a proportion of the available width of a page*
  - *from a 12th to the full width of the page*

```
.col-1 {width: 8.33%;}
.col-2 {width: 16.66%;}
.col-3 {width: 25%;}
.col-4 {width: 33.33%;}
.col-5 {width: 41.66%;}
.col-6 {width: 50%;}
.col-7 {width: 58.33%;}
.col-8 {width: 66.66%;}
.col-9 {width: 75%;}
.col-10 {width: 83.33%;}
.col-11 {width: 91.66%;}
.col-12 {width: 100%;}
```

- classes allow us to set a column span for a given element
  - *from 1 to 12 in terms of the number of grid columns an element may span*

CSS grid layout - example - part 4

grid.css

- then set some further styling for each abstracted `col-` class

```
[class*="col-"] {
  position: relative;
  float:left;
  padding: 20px;
  border: 1px solid #333;
}
```

- create columns by wrapping our content elements into rows
- each row always needs 12 columns

```
<div class="row">
  <div class="col-6">left column</div>
  <div class="col-6">right column</div>
</div>
```

CSS grid layout - example - part 5

grid.css

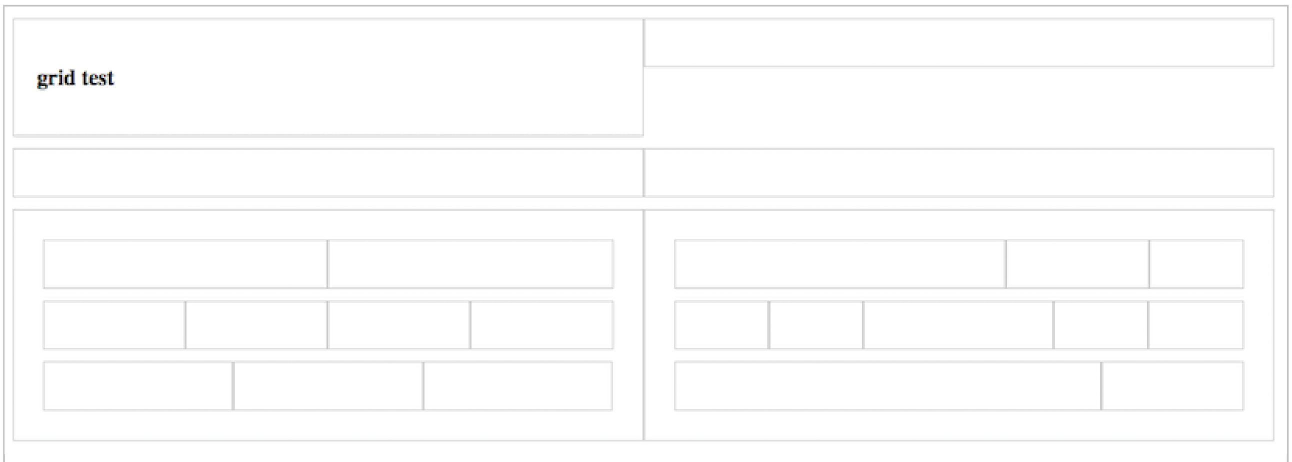
- due to the initial CSS of float left, each column is floated to the left
- columns are interpreted by subsequent elements in the hierarchy as non-existent
  - *initial placement will reflect this design*
- prevent this issue in layout, add the following CSS to grid stylesheet

```
.row:before, .row:after {
  content: "";
  clear: both;
  display: block;
}
```

- benefit of the clearfix, `clear: both`
  - *make row stretch to include columns it contains*
  - *without the need for additional markup*

DEMO - Grid Layout 1 - no gutters

Image - Grid Layout 1



Grid Layout - No Gutters

CSS grid layout - example - part 6

grid.css

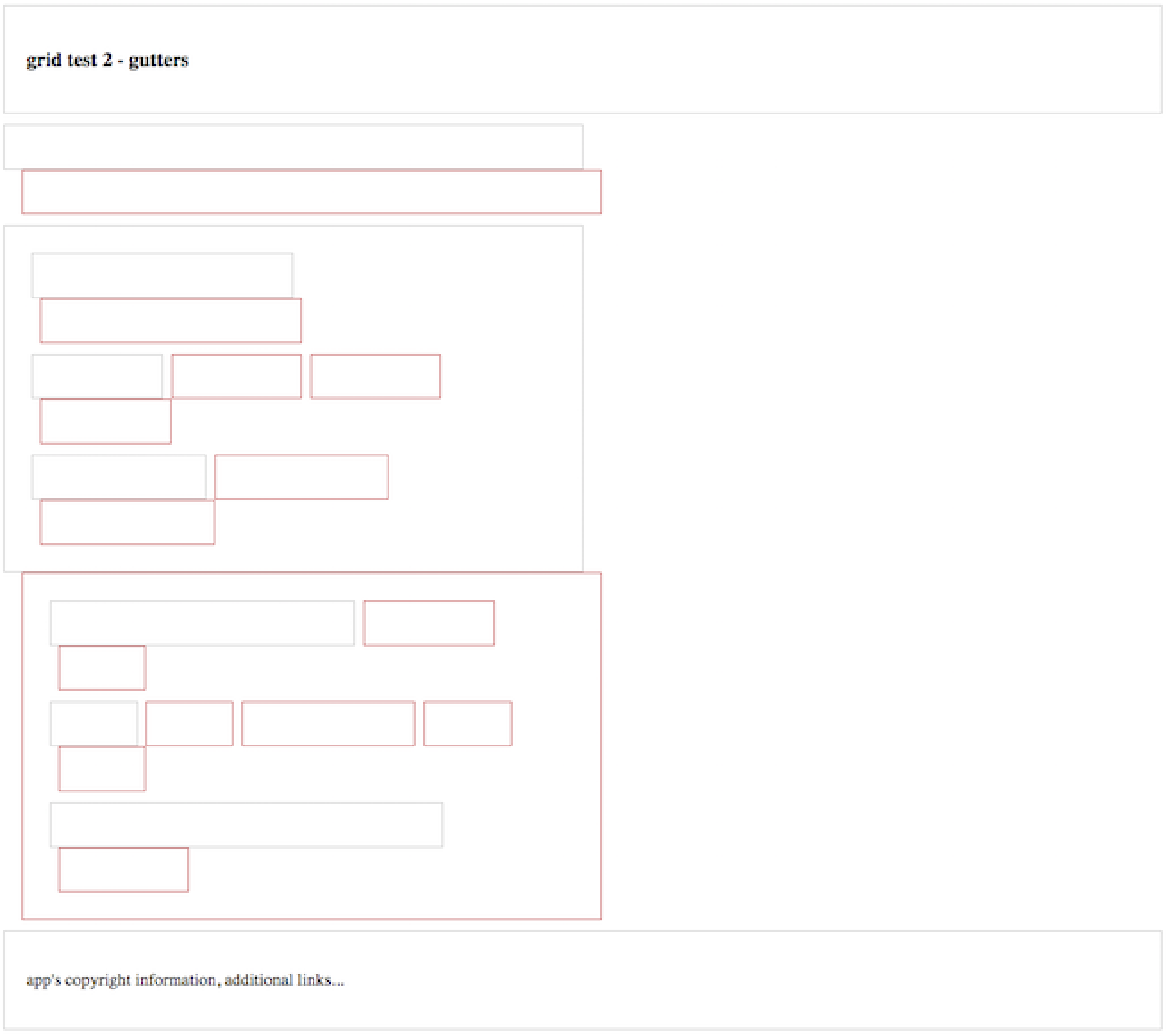
- add gutters to our grid to help create a sense of space and division in the content
- simplest way to add a gutter to the current grid css is to use padding
  - *rows can use padding, for example*

```css
.row {
  padding: 5px;
}
```

- issue with simply adding padding to the columns
  - *margins are left in place, next to each other*
  - *column borders next to each with no external column gutter*
- fix this issue by targeting columns that are a sibling to a preceding column
- means we do not need to modify the first column, only subsequent siblings

```css
[class*="col-"] + [class*="col-"] {
  margin-left: 1.6%;
}
```

Image - Grid Layout 2


grid test 2 - gutters

app's copyright information, additional links...

Grid Layout - Gutters Overflow

CSS grid layout - part 7

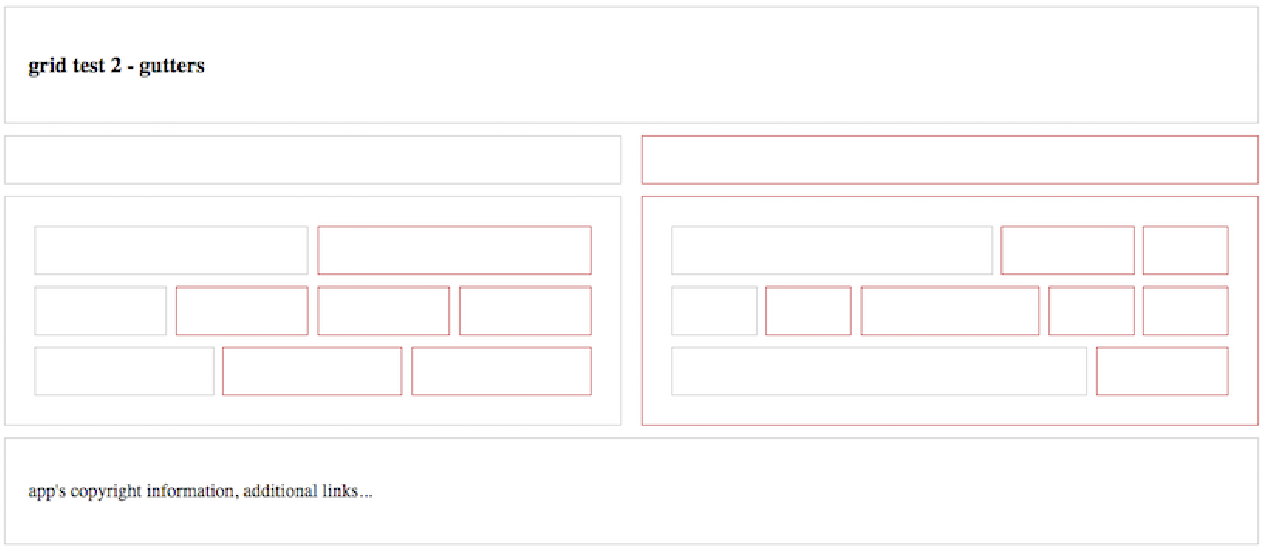
grid.css

- to fix this issue we recalculate permitted % widths for our columns in the CSS
  - *we now have % widths as follows*

```
.col-1 {width: 6.86%;}
.col-2 {width: 15.33%;}
.col-3 {width: 23.8%;}
.col-4 {width: 32.26%;}
.col-5 {width: 40.73%;}
.col-6 {width: 49.2%;}
.col-7 {width: 57.66%;}
.col-8 {width: 66.13%;}
.col-9 {width: 74.6%;}
.col-10 {width: 83.06%;}
.col-11 {width: 91.53%;}
.col-12 {width: 100%;}
```

- DEMO - Grid Layout 2 - gutters

Image - Grid Layout 3


grid test 2 - gutters

app's copyright information, additional links...

Grid Layout - Gutters

CSS grid layout - part 8

media queries

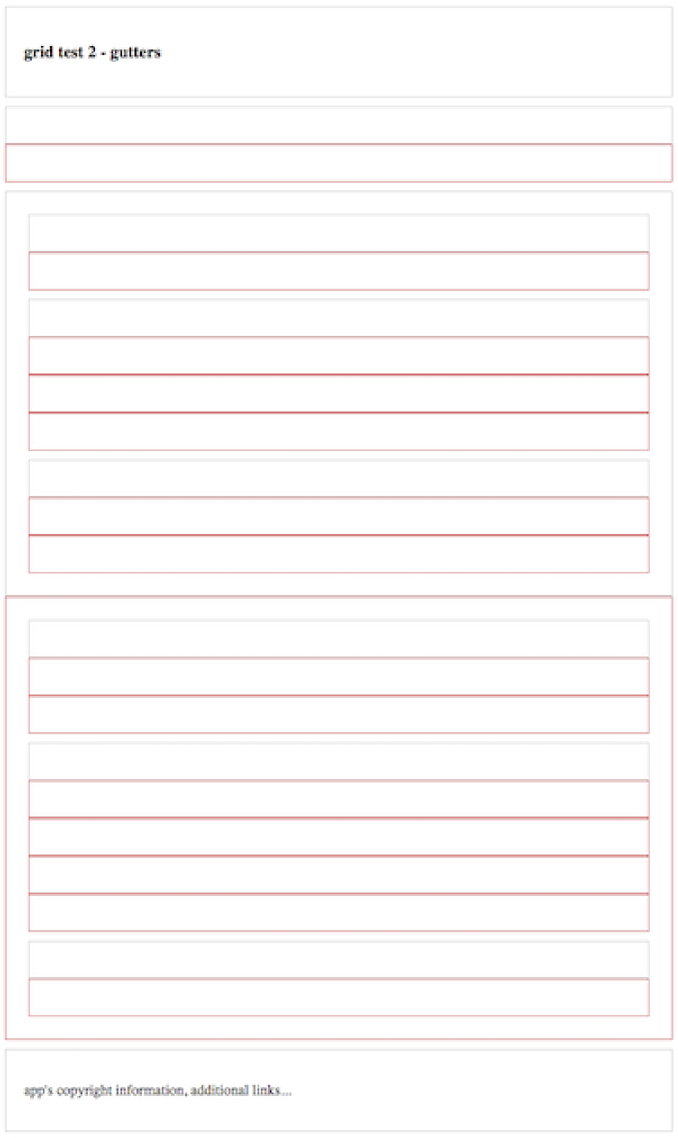
- often need to consider a mobile-first approach
- introduction of CSS3, we can now add **media queries**
- modify specified rulesets relative to a given condition
  - *eg: screen size for a desktop, tablet, and phone device*
- media queries allow us to specify a breakpoint in the width of the viewport
  - *will then trigger a different style for our application*
- could be a simple change in styles
  - *such as colour, font etc*
- could be a modification in the grid layout
  - *effective widths for our columns per screen size etc...*

```
@media only screen and (max-width: 900px) {
  [class*="col-"] {
  width: 100%;
  }
}
```

- gutters need to be removed
  - *specifying widths of 100% for our columns*

```
[class*="col-"] + [class*="col-"] {
  margin-left:0;
}
```

Image - Grid Layout 4



grid test 2 - gutters

app's copyright information, additional links...

Grid Layout - Media Queries

CSS3 Grid - intro

- gid layout with CSS is useful for structure and organisation
  - *applied to HTML page*
- usage similar to table for structuring data
- in its basic form
  - *enables developers to add columns and rows to a page*
- grid layout also permits more complex, interesting layout options
  - *e.g. overlap and layers...*
- further information on MDN website,
  - *MDN - CSS Grid Layout*

CSS3 Grid - general concepts & usage

- grid may be composed of rows and columns
  - *thereby forming an intersecting set of horizontal and vertical lines*
- elements may be added to the grid with reference to this structured layout

Grid layout in CSS includes the following general features,

- additional tracks for content
  - *option to create more columns and rows as needed to fit dynamic content*
- control of alignment
  - *align a grid area or overall grid*
- control of overlapping content
  - *permit partial overlap of content*
  - *an item may overlap a grid cell or area*
- placement of items - explicit and implicit
  - *precise location of elements &c.*
  - *use line numbers, names, grid areas &c.*
- variable track sizes - fixed and flexible, e.g.
  - *specify pixel size for track sizes*
  - *or use flexible sizes with percentages or new* `fr` *unit*

CSS3 Grid - grid container

- define an element as a grid container using
  - *`display: grid`* or *`display: inline-grid`*
- any children of this element become *grid items*
  - e.g.

```css
.wrapper {
  display: grid;
}
```

- we may also define other, child nodes as a `grid` container
  - *any direct child nodes to a grid container are now defined as grid items*

CSS3 Grid - what is a grid track?

- rows and columns defined with
  - *`grid-template-rows` and `grid-template-columns` properties*
- in effect, these define *grid tracks*
- as MDN notes,
  - *"a grid track is the space between any two lines on the grid.""*
  - *(https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Basic_Concepts_of_Grid_Layout)*
- so, we may create both row and column tracks, e.g.

```
.wrapper {
  display: grid;
  grid-template-columns: 200px 200px 200px;
}
```

- `wrapper` class now includes three defined columns of width 200px
  - *thereby creating three tracks*
- *n.b.* a track may be defined using any valid length unit, not just `px`...

CSS3 Grid - `fr` unit for tracks - part 1

- CSS Grid now introduces an additional length unit for tracks, `fr`
- `fr` unit represents fractions of the space available in the current grid container
  - e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

- we may also apportion various space to tracks, e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: 2fr 1fr 1fr;
}
```

- creates three tracks in the grid
  - *but overall space effectively now occupies four parts*
  - *two parts for `2fr`, and one part each for remaining two `1fr`*

CSS3 Grid - `fr` unit for tracks - part 2

- we may also be specific in this sub-division of parts in tracks, e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: 200px 1fr 1fr;
}
```

- first track will occupy a width of `200px`
  - *remaining two tracks will each occupy 1 fraction unit*

CSS3 Grid - `repeat()` notation for `fr` - part I

- for larger, repetitive grids, easier to use `repeat()`
  - *helps define multiple instances of the same track*
  - e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
}
```

- this creates four separate tracks - each defined as `1fr` unit's width

CSS3 Grid - `repeat()` notation for `fr` - part 2

- `repeat()` notation may also be used as part of the track definition
  - e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: 200px repeat(4, 1fr) 100px;
}
```

- this example will create
  - *one track of `200px` width*
  - *then four tracks of `1fr` width*
  - *and finally a single track of `100px` width*
- `repeat()` may also be used with multiple track definitions
  - *thereby repeating multiple times*
  - e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: repeat(4, 1fr 2fr);
}
```

- this will now create eight tracks
  - *the first four of width `1fr`*
  - *and the remaining four of `2fr`*

CSS3 Grid - implicit and explicit grid creation

- in the above examples
  - *we simply define tracks for the columns*
  - *and CSS grid will then apportion content to required rows*
- we may also define an explicit grid of columns and rows
  - e.g.

```css
.wrapper {
  display: grid;
  grid-template-columns: repeat(2 1fr);
  grid-auto-rows: 150px;
}
```

- this slightly modifies an implicit grid to ensure each row is `200px` tall

CSS3 Grid - track sizing

- a grid may require tracks with a minimum size
  - *and the option to expand to fit dynamic content*
- e.g. ensuring a track does not collapse below a certain height or width
  - *and that it has the option to expand as necessary for the content...*
- CSS Grid provides a `minmax()` function, which we may use with rows
  - e.g.

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(2 1fr);
  grid-auto-rows: minmax(150px, auto);
}
```

- ensures each row will occupy a minimum of `150px` in height
  - *still able to stretch to contain the tallest content*
  - *whole row will expand to meet the `auto` height requirements*
  - *thereby affecting each track in the row*

CSS3 Grid - grid lines

- a grid is defined using *tracks*
  - *and not lines in the grid*
- created grid also helps us with positioning by providing numbered lines
- e.g. in a three column, two row grid we have the following,
  - *four lines for the three vertical columns*
  - *three lines for the two horizontal rows*
- such lines start at the left for columns, and at the top for rows
- *n.b.* line numbers start relative to written script
  - *e.g left to right for western, right to left for arabic...*

CSS3 Grid - positioning against lines

- when we place an item in a grid
  - *we use these lines for positioning, and not the tracks*
- reflected in usage of
  - *`grid-column-start`, `grid-column-end`, `grid-row-start`, and `grid-row-end` properties.*
- items in the grid may be positioned from one line to another
  - *e.g. column line 1 to column line 3*
- *n.b.* default span for an item in a grid is one track,
  - *e.g. define column start and no end - default span will be one track...*
  - e.g.

```css
.content1 {
  grid-column-start: 1;
  grid-column-end: 4;
  grid-row-start: 1;
  grid-row-end: 3;
}
```

CSS3 Grid - grid cell & grid area

grid cell

- a *cell* is the smallest unit on the defined grid layout
- it is conceptually the same as a cell in a standard table
- as content is added to the grid, it will be stored in one cell

grid area

- we may also store content in multiple cells
  - *thereby creating grid areas*
- grid areas must be rectangular in shape
- e.g. a grid area may span multiple row and column tracks for required content

CSS3 Grid - add some gutters

- gutters may be created using the *gap* property
  - *available for either column or row*
  - `column-gap` and `row-gap`
  - e.g.

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(4, 1fr 2fr);
  column-gap: 5px;
  row-gap: 10px;
}
```

- *n.b.* any space used for gaps will be determined prior to assigned space for `fr` tracks

CSS3 Grid - structure and layout

fun exercise

Choose one of the following app examples,

- **sports** website for latest scores and updates
  - *e.g. scores for current matches, statistics, team data, player info &c.*
- **shopping** website
  - *product listings and adverts, cart, reviews, user account page &c.*
- **restaurant** website
  - *introductory info, menus, sample food images, user reviews &c.*

Then, consider the following

- use of a **grid** to layout your example pages
  - *where is it being used?*
  - *why is it being used for a given part of the UI?*
- how is the defined **grid** layout working with the **box model**?
- rendering of **box model** in the main content relative to **grid** usage
  - *i.e. box model updates due to changes in content*

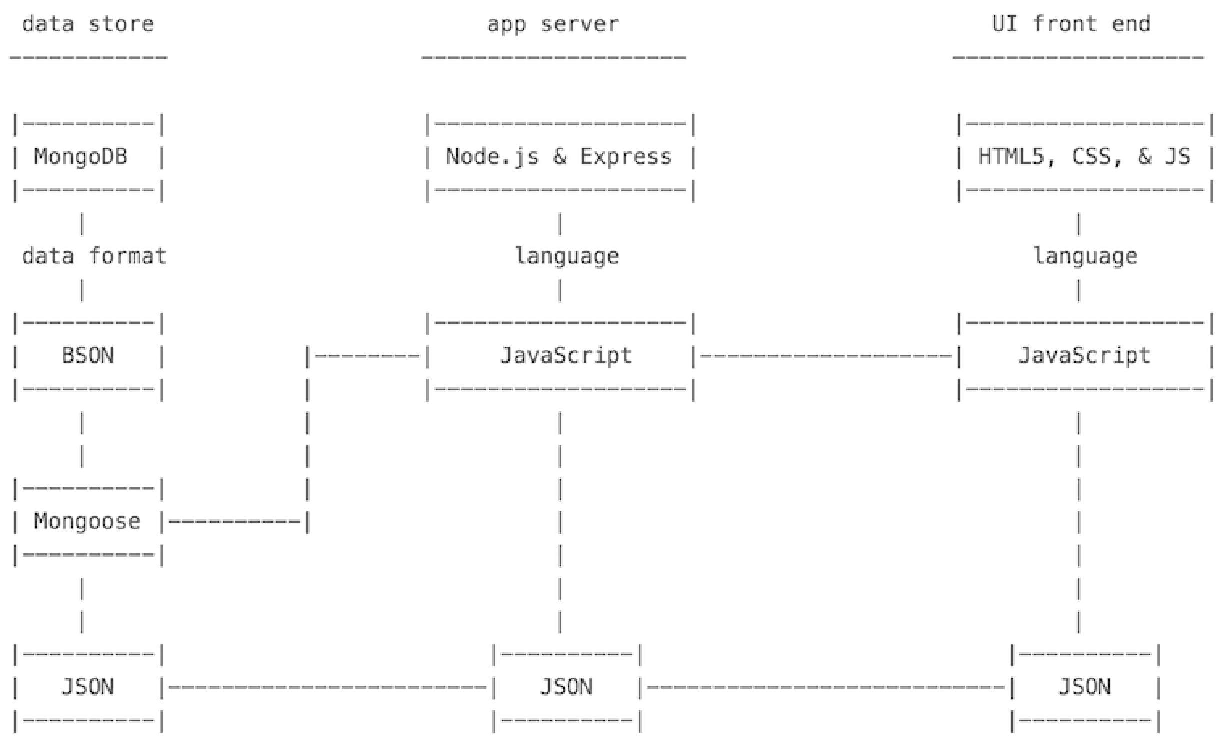
CSS3 Grid - working examples

- grid basic - page zones and groups
- grid basic - article style page
- grid layout - articles with scroll

building a web app - sample outline of underlying structure

- apps developed using a full JavaScript stack
- using and incorporating JS into each part of app's development
  - *UI front-end*
  - *app server and management*
  - *data store and management*
- Technologies will include
  - *front-end: HTML5, CSS, JS...*
  - *app server: Node.js, Express...*
  - *data store: MongoDB, Redis, Mongoose...*
- Data format is JSON

Image - building a web app - sample outline



JS full-stack outline

n.b. *I've explicitly omitted any arrows for flow within this diagram. This is something we'll return to as we start to work with Node.js, Mongoose, and MongoDB.*

JS Intro

- JavaScript (JS) a core technology for client-side design and development
- now being used as a powerful technology to help us
  - *rapidly prototype and develop web, mobile, and desktop apps*
- libraries such as jQuery, React, AngularJS, and Node.js
- helps develop cross-platform apps
  - *Apache Cordova*
  - *Electron*
- Embedded systems
  - *Espruino - http://www.espruino.com/*
  - *Tessel - https://tessel.io/*

JS Basics - operators

- operators allow us to perform
  - *mathematical calculations*
  - *assign one thing to another*
  - *compare and contrast...*
- simple * operator, we can perform multiplication

```
2 * 4
```

- we can add, subtract, and divide numbers as required
- mix mathematical with simple assignment

```
a = 4;
b = a + 2;
```

JS Basics - some common operators - part I

Assignment

- `=`
- eg: `a = 4`

Comparison

- `<, > <=, >=`
- eg: `a <= b`

Compound assignment

- `+=, -=, *=, /=`
- compound operators are used to combine a mathematical operation with assignment
- same as `result = result + expression`
- eg: `a += 4`

Equality

operator	**description**
==	loose equals
===	strict equals
!=	loose not equals
!==	strict not equals

- eg: `a != b`

JS Basics - some common operators - part 2

Increment/Decrement

- increment or decrement an existing value by 1
  - *`++`, `--`*
  - *eg: `a++` is equal to `a = a + 1`*

Logical

- used to express compound conditionals - **and**, **or**
  - *`&&`, `||`*
  - *eg: `a || b`*

Mathematical

- `+`, `-`, `*`, `/`
  - *eg: `a * 4` or `a / 4`*

Object property access

- properties in objects are specific named locations for holding values and data
- effectively, values within values
  - *`.`*
  - *eg: `a.b` means object `a` with a property of `b`*

JS Basics - values and types

- able to express different representations of values
  - *often based upon need or intention*
  - *known as **types***
- JS has built-in types
  - *allow us to represent **primitive** values*
  - *eg: **numbers**, **strings**, **booleans***
- such values in the source code are simply known as **literals**
- **literals** can be represented as follows,
  - *string literals use double or single quotes eg:* `"some text"` *or* `'some more text'`
  - *numbers and booleans are represented without being escaped eg:* `49, true;`
- also consider arrays, objects, functions...

JS Basics - type conversion

- option and ability to convert types in JS
  - *in effect,* ***coerce*** *our values and types from one type to another*
- convert a number, or coerce it, to a string
- built-in JS function, `Number()`, is an explicit coercion
  - *explicit coercion, convert any type to a number type*
- implicit coercion, JS will often perform as part of a comparison

```
"49" == 49
```

- JS implicitly coerces left string to a matching number
  - *then performs the comparison*
- often considered bad practice
  - *convert first, and then compare*
- implicit coercion still follows rules
  - *can be very useful*

JS Basics - variables - part 1

- **symbolic** container for values and data
- applications use containers to keep track and update values
- use a **variable** as a container for such values and data
  - *allow values to vary over time*
- JS can emphasize types for values, does not enforce on the variable
  - ***weak typing*** *or* ***dynamic typing***
  - *JS permits a variable to hold a value of any type*
- often a benefit of the language
- a quick way to maintain flexibility in design and development

JS Basics - variables - part 2

- declare a variable using the keyword `var`
- declaration does not include **type** information

```
var a = 49;
//double var a value
var a = a * 2;
//coerce var a to string
var a = String(a);
//output string value to console
console.log(a);
```

- `var` a maintains a running total of the value of a
- keeps record of changes, effectively **state** of the value
- **state** is keeping track of changes to any values in the application

JS Basics - variables - part 3

- use variables in JS to enable central, common references to our values and data
- better known in most languages simply as **constants**
- JS is similar
  - *creates a read-only reference to a value*
  - *value itself is not immutable, e.g. an object...*
  - *it's simply the identifier that cannot be reassigned*
  - *JS constants are also bound by scoping rules*
- allow us to define and declare a variable with a value
  - *not intended to change throughout the application*
- **constants** are often declared together
  - *uppercase is standard practice - although not a rule...*
- form a store for values abstracted for use throughout an app
- JS normally defines constants using uppercase letters,

```
var NAME = "Philae";
```

- ECMAScript 6, ES6, introduces additional variable keywords
  - *e.g.* `const`

```
const TEMPLE_NAME = "Philae";
```

- benefits of abstraction, ensuring value is not accidentally changed
  - *change rejected for a running app*
  - *in* `strict` *mode, app will fail with an error for any change*

JS Basics - comments

- JS permits comments in the code
- two different implementations

single line

```
//single line comment
var a = 49;
```

multi-line

```
/* this comment has more to say...
we'll need a second line */
var b = "forty nine";
```

JS Basics - logic - blocks

- simple act of grouping contiguous and related code statements together
  - *known as **blocks***
- block defined by wrapping statements together
  - *within a pair of curly braces, { }*
- **blocks** commonly attached to other forms of control statement

```
if (a > b) {
...do something useful...
}
```

JS Basics - logic - conditionals - part 1

- conditionals, conditional statements require a decision to be made
- code statement, application, consults **state**
  - *answer will predominantly be a simple **yes** or **no***
- JS includes many different ways we can express **conditionals**
- most common example is the `if` statement
  - *if this given condition is true, do the following...*

```
if (a > b) {
console.log("a is greater than b...");
}
```

- `if` statement requires an expression between the parentheses
  - *evaluates as either true or false*

JS Basics - logic - conditionals - part 2

- additional option if this expression returns false
  - *using an **else** clause*

```
if (a > b) {
console.log("a is greater than b...");
} else {
console.log("no, b is greater...");
}
```

- for an `if` statement, JS expects a `boolean`
- JS defines a list of values that it considers *false*
  - *eg: 0...*
- any value not on this list of *false* values will be considered true
  - *coerced to true when defined as a `boolean`*
- conditionals in JS also exist in another form
  - *the `switch` statement*
  - *more to come...*

JS Basics - logic - loops

- loops allow repetition of sets of actions until a condition fails
- repetition continues whilst the requested condition holds
- loops take many different forms and follow this basic behaviour
- a loop includes the *test condition* as well as a *block*
  - *normally within curly braces*
  - *block executes, an iteration of the loop has occurred*
- good examples of this behaviour include `while` and `do...while` loops
- basic difference between these loops, `while` and `do...while`
  - *conditional tested is before the first iteration (`while` loop)*
  - *after the first iteration (`do...while`) loop*
- if the condition is initially false
  - *a `while` loop will never run*
  - *a `do...while` will run through for the first time*
- also stop a JS loop using the common `break` statement
- `for` loop has three clauses, including
  - *initialisation clause*
  - *conditional test clause*
  - *update clause*

JS Basics - logic - functions - part 1

- functions are a type of object
  - *may also have their own properties*
  - *define once, then re-use as needed throughout our application*
- **function** is a named grouping of code
  - *name can be called, and code will be run each time*
- JS functions can be designed with optional arguments
  - *known as **parameters***
  - *allow us to pass values to the function*
- functions can also optionally return a value

```
function outputTotal(total) {
  console.log(total);
}
var a = 49;
a = a * 3; // or use a *= 3;

outputTotal(a);
```

JS Basics - logic - functions - part 2

```
function outputTotal(total) {
  console.log(total);
}

function calculateTotal(amount, times) {
  amount = amount * times;
  return amount;
}

var a = 49;
a = calculateTotal(a, 3);
outputTotal(a);
```

- JSFiddle Demo

JS Basics - logic - scope

- scope or **lexical scope**
  - *collection of variables, and associated access rules by name*
- in JS each function gets its own scope
- variables within a function's given **scope**
  - *can only be accessed by code inside that function*
- variable name has to be unique within a function's scope
- same variable name could appear in different scopes
- nest one scope within another
  - *code in inner scope can access variables from either inner or outer scope*
  - *code in outer scope cannot, by default, access code in the inner scope*

JS Basics - logic - scope example

```
function outerScope() {
  var a = 49;
  //scope includes outer and inner
  function innerScope() {
    var b = 59;
    //output a and b
    console.log(a + b); //returns 108
  }
  innerScope();

  //scope limited to outer
  console.log(a); //returns 49
}

//run outerScope function
outerScope();
```

- JSFiddle Demo

JS Basics - strict mode

- intro of ES5 - JS now includes option for **strict** mode
  - *ensures tighter code and better compliance...*
  - *often helps ensure greater compatibility, safer use of language...*
  - *can also help optimise code for rendering engines*
- add **strict** at different levels within our JS code
  - *eg: single function level or enforce for whole file*

```
function outerScope() {
  "use strict";
  //code is strict

  function innerScope() {
  //code is strict

  }
}
```

- if we set **strict** mode for complete file - set at top of file
  - *all functions and code will be checked against **strict** mode*
  - *eg: check against auto-create for global variables*
  - *or missing `var` keyword for variables...*

```
function outerScope() {
  "use strict";
  a = 49; // `var` missing - ReferenceError
}
```

JS Core - values and types

- JS has typed values, not typed variables
- JS provides the following built-in types
  - *boolean*
  - *null*
  - *number*
  - *object*
  - *string*
  - *symbol (new in ES6)*
  - *undefined*
- more help provided by JS's `typeof` operator
  - *examine a value and return its type*

```
var a = 49;
console.log(typeof a); //result is a number
```

- as of ES6, there are 7 possible return types in JS
- **NB:** JS variables do not have types, mere containers for values
  - *values specify the type*

```
var a = null;
console.log(typeof a); //result is object - known bug in JS...
```

JS Core - objects - part 1

Objects

- **object** type includes a compound value
  - *JS can use to set properties, or named locations*
- each of these properties holds its own value
  - *can be defined as any type*

```
var objectA = {
  a: 49,
  b: 59,
  c: "Philae"
};
```

- access these values using either **dot** or **bracket** notation

```
//dot notation
objectA.a;
//bracket notation
objectA["a"];
```

JS Core - objects - example

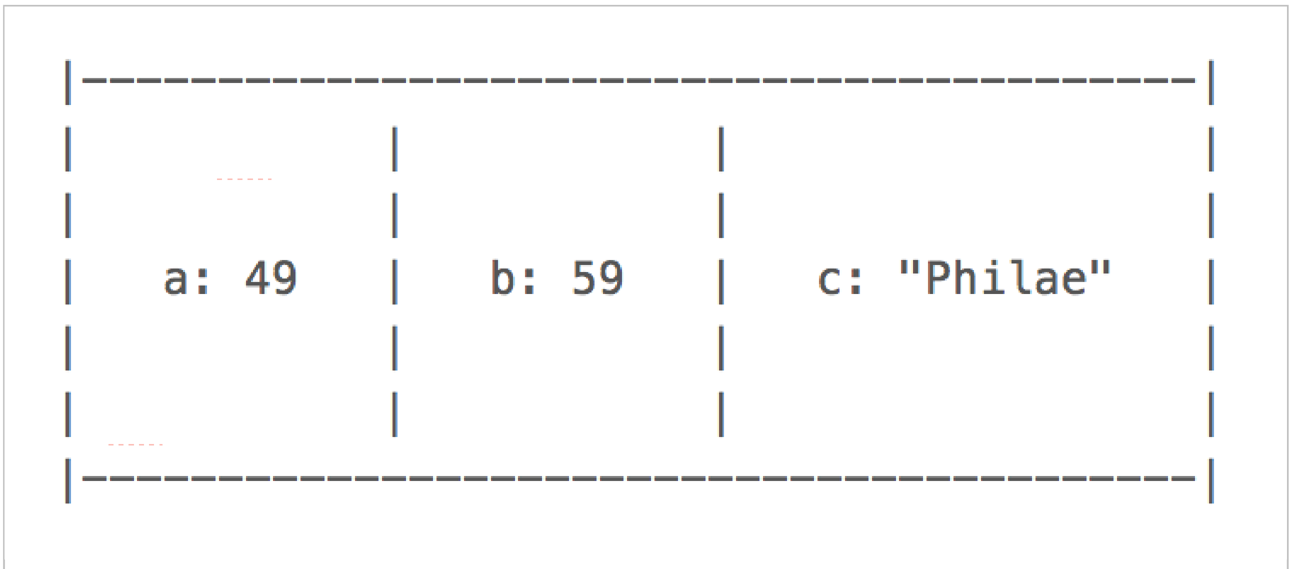
```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// log output with dot notation
console.log(`archive is ${object.archive}`);

// log output with bracket notation - returns undefined
console.log(`access is restricted to ${object[1]}`);

// log output with bracket notation
console.log(`purpose is ${object['purpose']}`);
```

Image - JS Object



```
|-----------------------------------------|
|           |           |                 |
|           |           |                 |
|   a: 49   |   b: 59   |   c: "Philae"   |
|           |           |                 |
|           |           |                 |
|-----------------------------------------|
```

JS Object

ES6 - template literals

```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// log output with template literals
console.log(`archive is ${object.archive}`);

// log output
console.log('archive is ' + object.archive);

// log output all object properties with template literals
console.log(`archive = ${object.archive}, access = ${object.access},  purp

// log output all object properties
console.log('archive = ' + object.archive + ', access = ' + object.access
```

Demos

- CSS - Grid
  - *grid basic - page zones and groups*
  - *grid basic - article style page*
  - *grid layout - articles with scroll*
- JSFiddle
  - *Basic logic - functions*
  - *Basic logic - scope*

Resources

- MDN - CSS3 Grid
- MDN - JS
- MDN - JS Data Types and Data Structures
- MDN - JS Grammar and Types
- MDN - JS Objects
- W3 Schools - CSS Grid View
- W3 Schools - JS