

Comp 324/424 - Client-side Web Design

Spring Semester 2018 - Week 4

Dr Nick Hayward

CSS Basics - intro to syntax

- simple, initial concepts for CSS syntax
- follows a defined syntax pattern, e.g.
- selector
 - e.g. *body* or *p*
- declaration
 - *property and value pairing*

```
body {  
  color: black;  
  font-family: "Times New Roman", Georgia, Serif;  
}
```

- `body` is the selector, `color` is the property, and `black` is the value.

CSS Basics - rulesets

- a CSS file is a group of rules for styling our HTML documents
- rules form **rulesets**, which can be applied to elements within the DOM
- rulesets consist of the following,
 - *a selector - p*
 - *an opening brace - {*
 - *a set of rules - color: blue*
 - *a closing brace - }*
- for example,

```
body {  
  width: 900px;  
  color: #444;  
  font-family: "Times New Roman", Georgia, Serif;  
}
```

- [HTML Colour Picker](#)

CSS Basics - comments

- add comments to help describe the selector and its properties,

```
/* 'color' can be set to a named value, HEX value (e.g. #444) &c. */  
p {  
  color: blue;  
  font-size: 14px;  
}
```

- comments can be added before the selector or within the braces

Image - CSS Syntax

Selector

```
|-----|  
|  p  |  
|-----|
```

Declaration

```
|-----|  
| { font-size: 14px; } |  
|-----|  
      ^         ^  
      |         |  
property      value
```

CSS Syntax

CSS Basics - display

- display HTML elements in one of two ways
 - *inline* - e.g. `<a>` or ``
 - *displays content on the same line*

```
<div class="content">
  <p>
    <a href="...">Philae</a> is a <span>Ptolemaic</span> era temple in Egypt.
  </p>
</div>
```

- more common to display elements as *block-level* instead of *inline* elements
- element's content rendered on a new line outside flow of content
- a few sample block elements include,
 - `<article>`, `<div>`, `<figure>`, `<main>`, `<nav>`, `<p>`, `<section>`...
- *block-level* is not technically defined for new elements in HTML5

CSS Basics - inline elements

Current inline elements include, for example:

- b | big | i | small
- abbr | acronym | cite | dfn | em | strong | var
- a | br | img | map | script | span | sub | sup
- button | input | label | select | textarea
- ...

Source - [MDN - Inline Elements](#)

n.b. not all inline elements supported in HTML5

CSS Basics - block-level elements

Current block-level elements include:

- address | article | aside | blockquote | canvas | div
- fieldset | figure | figcaption | footer | form
- h1 | h2 | h3 | h4 | h5 | h6
- header | hgroup | hr | main | nav
- ol | output | p | pre | section | table | tfoot | ul | video
- ...

Source - MDN - Block-level Elements

n.b. *block-level* is not technically defined for new elements in HTML5

CSS Basics - HTML5 content categories - part I

- **block-level** is not technically defined for new elements in HTML5
- now have a slightly more complex model called **content categories**
- includes three primary types of content categories

These include,

- **main content categories** - describe common content rules shared by many elements
- **form-related content categories** - describe content rules common to form-related elements
- **specific content categories** - describe rare categories shared by only a small number of elements, often in a specific context

CSS Basics - HTML5 content categories - part 2

- **Metadata content** - modify presentation or behaviour of document, setup links, convey additional info..
 - `<base>`, `<command>`, `<link>`, `<meta>`, `<noscript>`, `<script>`, `<style>`, `<title>`
- **Flow content** - typically contain text or embedded content
 - `<a>`, `<article>`, `<canvas>`, `<figure>`, `<footer>`, `<header>`, `<main>`...
- **Sectioning content** - create a section in current outline to define scope of `<header>` elements, `<footer>` elements, and *heading* content
 - `<article>`, `<aside>`, `<nav>`, `<section>`
- **Heading content** - defines title of a section, both explicit and implicit sectioning
 - `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`, `<hgroup>`

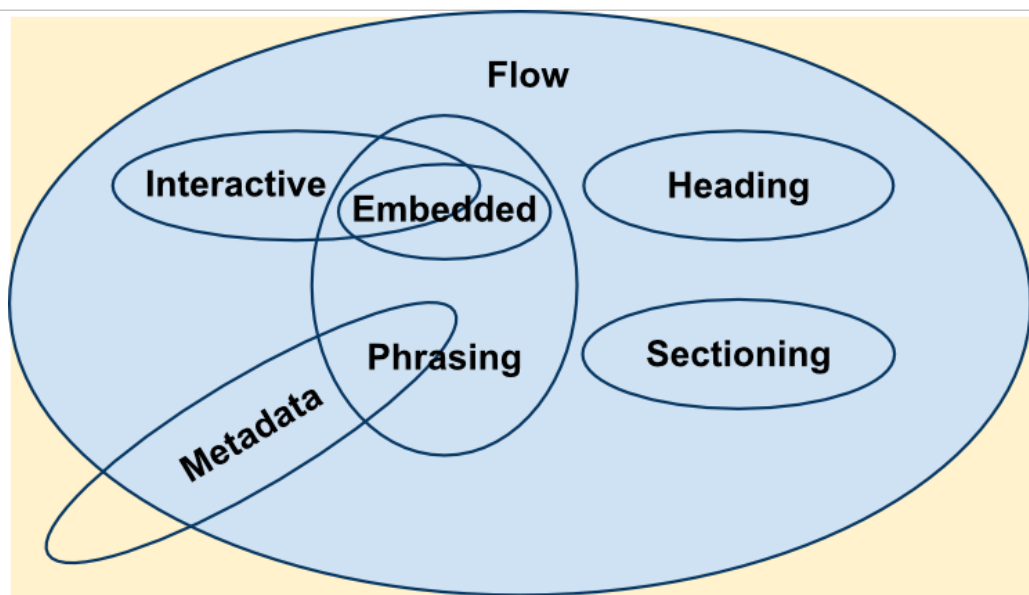
Source - MDN Content Categories

CSS Basics - HTML5 content categories - part 3

- **Phrasing content** - defines the text and the mark-up it contains
 - `<audio>`, `<canvas>`, `<code>`, ``, `<label>`, `<script>`, `<video>`...
 - other elements can belong to this category if certain conditions are met. e.g. `<a>`
- **Embedded content** - imports or inserts resource or content from another mark-up language or namespace
 - `<audio>`, `<canvas>`, `<embed>`, `<iframe>`, ``, `<math>`, `<object>`, `<svg>`, `<video>`
- **Interactive content** - includes elements that are specifically designed for user interaction
 - `<a>`, `<button>`, `<details>`, `<embed>`, `<iframe>`, `<keygen>`, `<label>`, `<select>`, `<textarea>`
 - additional elements, available under specific conditions, include
 - `<audio>`, ``, `<input>`, `<menu>`, `<object>`, `<video>`
- **Form-associated content** - elements contained by a form parent element
 - `<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`...
 - there are also several sub-categories, including *listed*, *labelable*, *submittable*, *resettable*

Source - MDN Content Categories

Image - HTML5 Content Categories



HTML5 Content Categories

Source - MDN - Content Categories

CSS Basics - box model - part I

- consideration of the CSS box model
- a document's attempt to represent each element as a rectangular box
- boxes and properties determined by browser rendering engine
- browser calculates size, properties, and position of these required boxes
- properties can include, for example,
 - *colour, background features, borders, width, height...*
- box model designed to describe an element's required space and content
- each box has a series of edges,
 - **margin** edge
 - **border** edge
 - **padding** edge
 - **content** edge

CSS Basics - box model - part 2

Content

- box's **content area** describes element's actual content
- properties can include color, background, img...
 - *apply inside the **content** edge*
- dimensions include **content width** and **content-height**
- content size properties (assuming that the box-sizing property remains default) include,
 - *width, min-width, max-width, height, min-height, max-height*

Demo - CSS Box Model

- Demo - CSS Box Model

CSS Basics - box model - part 3

Padding

- box's **padding area** includes the extent of the padding to the surrounding border
- background, colour etc properties for a content area extend into the padding
 - *we often consider the padding as extending the content*
- padding itself is located in the box's **padding edge**
- dimensions are the width and height of the **padding-box**.
- control space between padding and content edge using the following properties,
 - *padding-top, padding-right, padding-bottom, padding-left*
 - *padding (sizes calculated clock-wise)*

Demo - CSS Box Model - Padding

- [JSFiddle - CSS Box Model](#)

CSS Basics - box model - part 4

Border

- **border area** extends **padding area** to area containing the borders
- it becomes the area inside the **border edge**
- define its dimensions as the width and height of the **border-box**
- calculated area depends upon the width of the border we set in the CSS
- set size of our border using the following properties in CSS,
 - *border-width*
 - *border*

Demo - CSS Box Model - Border

- JSFiddle - CSS Box Model

CSS Basics - box model - part 5

Margin

- **margin area** can extend this border area with an empty area
 - *useful to create a defined separation of one element from its neighbours*
- dimensions of area defined as width and height of the **margin-box**
- control size of our margin area using the following properties,
 - *margin-top, margin-right, margin-bottom, margin-left*
 - *margin* (sizes calculated clock-wise)

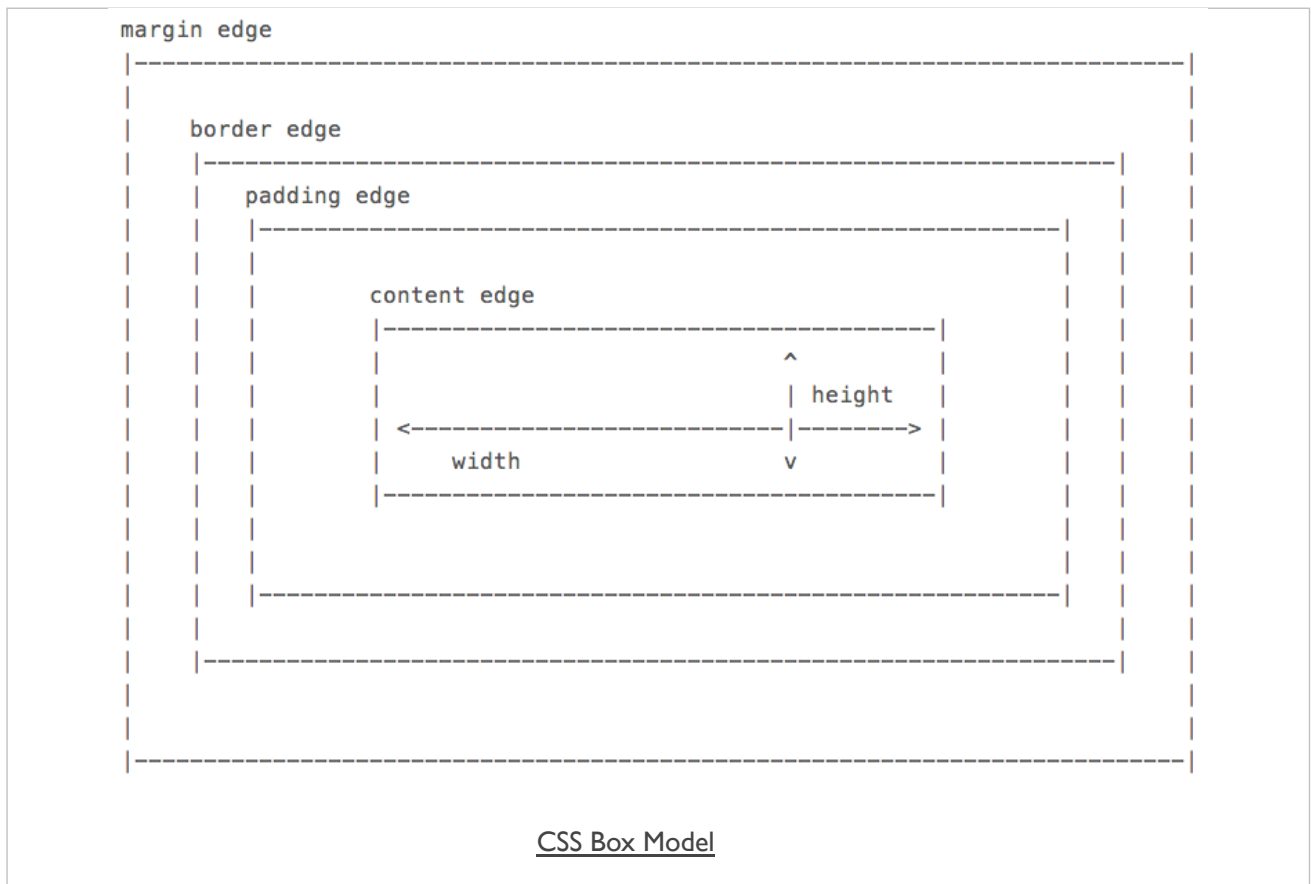
Demo - CSS Box Model - Margin

- JSFiddle - CSS Box Model

Demo - CSS Box Model

- Demo - CSS Box Model

Image - CSS Box Model



Source - MDN - CSS Box Model

CSS Basics - selectors

- **selectors** are a crucial part of working with CSS, JS...
- basic selectors such as

```
p {  
  color: #444;  
}
```

- above ruleset adds basic styling to our paragraphs
 - *sets the text colour to HEX value 444*
- simple and easy to apply
 - *applies the same properties and values to all paragraphs*
- specificity requires classes, pseudoclasses...

CSS Basics - classes

- add a **class** attribute to an element, such as a `<p>`
 - *can help us differentiate elements*
- also add a **class** to any DOM element
 - e.g. *add different classes to multiple `<p>` elements*

```
<p class="p1">paragraph one...</p>
<p class="p2">paragraph two...</p>
```

- we can now select our paragraphs by class name within the DOM
- then apply a **ruleset** for each class
- style this class for a specific element

```
p.p1 {
  color: #444;
}
```

- style all elements with the class p1, and not just `<p>` elements

```
.p1 {
  color: #444;
}
```

CSS Basics - pseudoclasses

- add a class to links or anchors, styling all links with the same ruleset
- we might also want to add specific styles for different link states
- styling links with a different colour
 - e.g. *whether a link has already been used or not*

```
a {  
  color: blue;  
}  
  
a:visited {  
  color: red;  
}
```

- visited is a CSS **pseudoclass** applied to the <a> element
- browser implicitly adds this pseudoclass for us, we add style

```
a:hover {  
  color: black;  
  text-decoration: underline;  
}
```

- pseudoclass for link element, <a>, hover

CSS Basics - complex selector - part I

- our DOM will often become more complicated and detailed
- depth and complexity will require more complicated selectors as well
- lists and their list items are a good example

```
<ul>
  <li>unordered first</li>
  <li>unordered second</li>
  <li>unordered third</li>
</ul>
<ol>
  <li>ordered first</li>
  <li>ordered second</li>
  <li>ordered third</li>
</ol>
```

- two lists, one unordered and the other ordered
- style each list, and the list items using rulesets

```
ul {
  border: 1px solid green;
}
ol {
  border: 1px solid blue;
}
```

Demo - Complex Selectors - Part I

- Demo - Complex Selectors Part I

CSS Basics - complex selector - part 2

- add a ruleset for the list items, ``
- applying the same style properties to both types of lists
- more specific to apply a ruleset to each list item for the different lists

```
ul li {  
  color: blue;  
}  
ol li {  
  color: red;  
}
```

- also be useful to set the background for specific list items in each list

```
li:first-child {  
  background: #bbb;  
}
```

- pseudoclass of `nth-child` to specify a style for the second, fourth etc child in the list

```
li:nth-child(2) {  
  background: #ddd;  
}
```

Demo - Complex Selectors - Part 2

- Demo - Complex Selectors Part 2

CSS Basics - complex selector - part 3

- style odd and even list items to create a useful alternating pattern

```
li:nth-child(odd) {  
  background: #bbb;  
}  
li:nth-child(even) {  
  background: #ddd;  
}
```

- select only certain list items, or rows in a table etc
 - e.g. *every fourth list item, starting at the first one*

```
li:nth-child(4n+1) {  
  background: green;  
}
```

- for **even** and **odd** children we're using the above with convenient shorthand
- other examples include
 - *last-child*
 - *nth-last-child()*
 - *many others...*

Demo - CSS Complex Selectors - Part 3

- Demo - Complex Selectors Part 3

CSS Basics - cascading rules - part I

- CSS, or cascading style sheets, employs a set of **cascading** rules
- rules applied by each browser as a ruleset conflict arises
 - e.g. issue of **specificity**

```
p {  
  color: blue;  
}  
p.p1 {  
  color: red;  
}
```

- the more specific rule, the class, will take precedence
- issue of possible duplication in rulesets

```
h3 {  
  color: black;  
}  
  
h3 {  
  color: blue;  
}
```

- **cascading** rules state the later ruleset will be the one applied
 - *blue heading instead of black...*

CSS Basics - cascading rules - part 2

- simple styling and rulesets can quickly become compounded and complicated
- different styles, in different places, can interact in complex ways
- a powerful feature of CSS
 - *can also create issues with logic, maintenance, and design*
- three primary sources of style information that form this cascade
 1. default styles applied by the browser for a given markup language
 - *e.g. colours for links, size of headings...*
 2. styles specific to the current user of the document
 - *often affected by browser settings, device, mode...*
 3. styles linked to the document by the designer
 - *external file, embedded, and as inline styles per element*

CSS Basics - cascading rules - part 3

- basic cascading nature creates the following pattern
 - *browser's style will be default*
 - *user's style will modify the browser's default style*
 - *styles of the document's designer modify the styles further*

CSS Basics - inheritance

- CSS includes inheritance for its styles
- descendants will inherit properties from their ancestors
- style an element
 - *descendants of that element within the DOM inherit that style*

```
body {  
  background: blue;  
}  
p {  
  color: white;  
}
```

- p is a descendant of body in the DOM
 - *inherits background colour of the body*
- this characteristic of CSS is an important feature
 - *helps to reduce redundancy and repetition of styles*
- useful to maintain outline of document's DOM structure
- most styles follow this pattern but not all
- margin, padding, and border rules for block-level elements **not inherited**

CSS Basics - fonts - part I

- fonts can be set for the body or within an element's specific ruleset
- we need to specify our font-family,

```
body {  
font-family: "Times New Roman", Georgia, Serif;  
}
```

- value for the font-family property specifies preferred and fall-back fonts
 - *Times New Roman*, then the browser will try *Georgia* and *Serif*
 - " " - quotation marks for names with spaces...

n.b. " " added due to CSS validator requesting this standard - it's believed to be a legacy error with the validator...

CSS Basics - fonts - part 2

- useful to be able to modify the size of our fonts as well

```
body {  
  font-size: 100%;  
}  
h3 {  
  font-size: x-large;  
}  
p {  
  font-size: larger;  
}  
p.p1 {  
  font-size: 1.1em;  
}
```

- set base font size to 100% of font size for a user's web browser
- scale our other fonts relative to this base size
 - CSS absolute size values, such as *x-large*
 - font sizes relative to the current context, such as *larger*
 - *em* are meta-units, which represent a multiplier on the current font-size
 - relative to current element for required font size
 - 1.5em of 12px is effective 18px
- *em* font-size scales according to the base font size
 - modify base font-size, *em* sizes adjust
- try different examples at
 - [W3 Schools - font-size](#)

Demo - CSS Fonts

- [Demo - CSS Fonts](#)
- [JSFiddle - CSS Fonts](#)

CSS Basics - fonts - part 3

- rem unit for font sizes
- size calculated against root of document

```
body {  
  font-size: 100%;  
}  
p {  
  font-size: 1.5rem;  
}
```

- element font-size will be root size * rem size
 - e.g. *body font-size is currently 16px*
 - *rem will be 16 * 1.5*

CSS Basics - custom fonts

- using fonts and CSS has traditionally been a limiting experience
- reliant upon the installed fonts on a user's local machine
- JavaScript embedding was an old, slow option for custom fonts
- web fonts are a lot easier
- Google Fonts
 - *from the font options, select*
 - *required fonts*
 - *add a <link> reference for the font to our HTML document*
 - *then specify the fonts in our CSS*

```
font-family: 'Roboto';
```

Demo - CSS Custom Fonts

- [Demo - CSS Custom Fonts](#)
- [JSFiddle - CSS Custom Fonts](#)

CSS Basics - reset options

- to help us reduce browser defaults, we can use a CSS reset
- reset allows us to start from scratch
- customise aspects of the rendering of our HTML documents in browsers
- often considered a rather controversial option
- considered controversial for the following primary reasons
 - *accessibility*
 - *performance*
 - *redundancy*
- use resets with care
- notable example of resets is Eric Meyer
 - *discussed reset option in May 2007 blog post*
- resets often part of CSS frameworks...

Demo - CSS Reset - Before

Browser default styles are used for

- `<h1>`, `<h3>`, and `<p>`
- Demo - CSS Reset Before

Demo - CSS Reset - After

Browser resets are implemented using the Eric Meyer stylesheet.

- Demo - CSS Reset After

CSS - a return to inline styles

- *inline* styles are once more gaining in popularity
 - *helped by the rise of React &c.*
- for certain web applications they are now an option
 - *allow us to dynamically maintain and update our styles*
- their implementation is not the same as simply embedding styles in HTML
 - *dynamically generated*
 - *can be removed and updated*
 - *can form part of our maintenance of the underlying DOM*
- inherent benefits include
 - *no cascade*
 - *built using JavaScript*
 - *styles are dynamic*

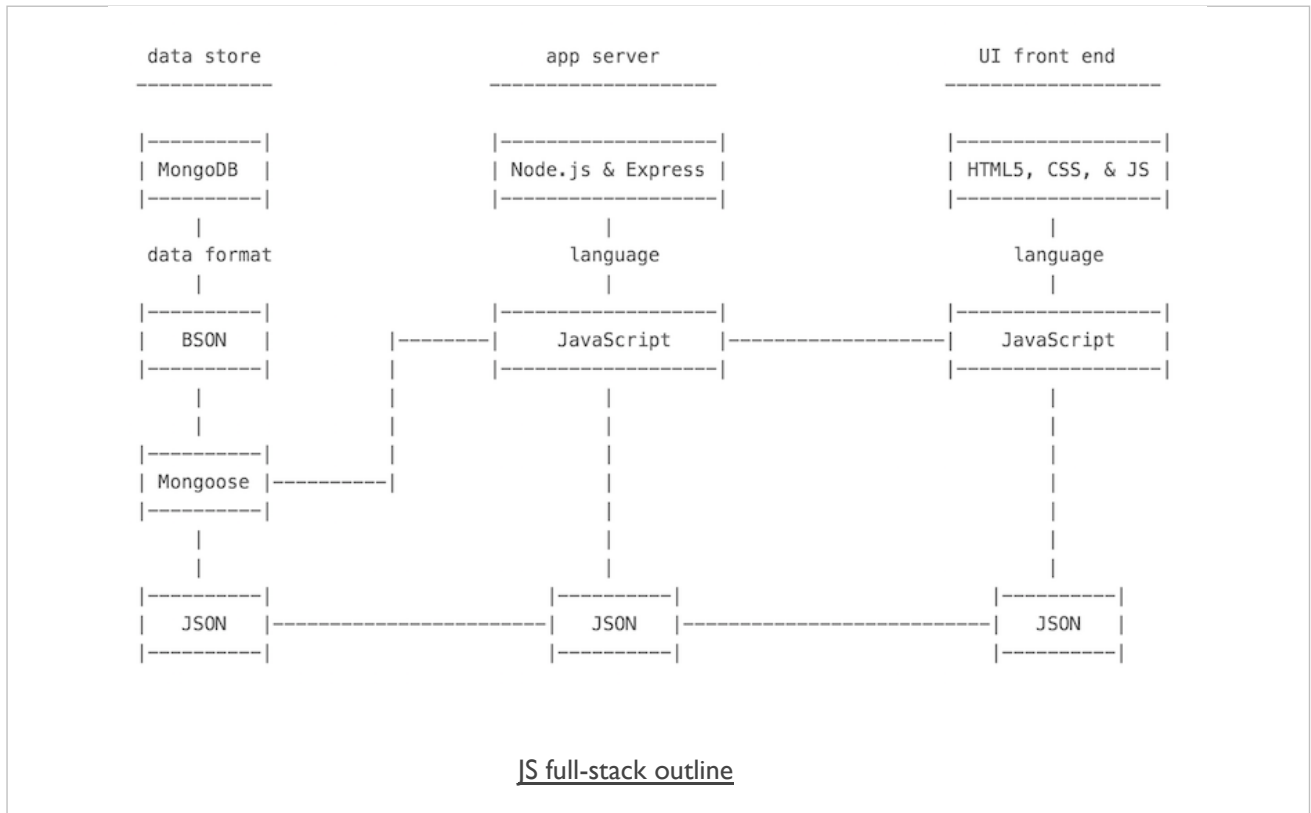
CSS - against inline styles

- CSS is designed for styling
 - *this is the extreme end of the scale - in effect, styling is only done with CSS*
- abstraction is a key part of CSS
 - *by separating out concerns, i.e. CSS for styling, our sites are easier to maintain*
- *inline* styles are too specific
 - *again, abstraction is the key here*
- some styling and states are easier to represent using CSS
 - *pseudoclasses etc, media queries...*
- CSS can add, remove, modify classes
 - *dynamically update selectors using classes*

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



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 I

- **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";
```

Demos

week 4

- CSS
- Demo - CSS Box Model
- Demo - Complex Selectors Part 1
- Demo - Complex Selectors Part 2
- Demo - Complex Selectors Part 3
- Demo - CSS Fonts
- Demo - CSS Custom Fonts
- Demo - CSS Reset Before
- Demo - CSS Reset After
- JSFiddle tests - CSS
- JSFiddle - CSS Box Model Padding
- JSFiddle - CSS Fonts
- JSFiddle - CSS Custom Fonts

References

week 4

CSS

- CSS Tricks - nth child recipes
- JSFiddle - CSS Basics
- MDN - CSS
- CSS box model
- Perishable Press - Barebones Web Templates
- W3 CSS
- W3 Schools - CSS
- W3 Schools - HTML Colour Picker
- W3 Web Style Sheets - Even & Odd