Comp 388/424 - Client-side Web Design

Spring Semester 2016 - Week 4

Dr Nick Hayward

Contents

Outline

Up to DEV week

- JS intro, basics, and fundamentals
- JS advanced, JQuery, APIs...
- basic JS tools, workflows, and methods
- testing, initial deployment
  - *pulling it all together*

CSS Basics - cascading rules - part 1

- CSS, or cascading style sheets, employs a set of **cascading** rules
- rules applied by each browser as a ruleset conflict arises
  - *eg: 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

So, hopefully you can already see how simple styling and rulesets can quickly become compounded and complicated. Different styles, specified in different places, can interact and affect each other in complex ways. This can become a powerful feature of CSS, but it can also create many issues with logic, maintenance, and design.

Therefore, we can often consider three primary sources of style information, relative to our documents, that form this cascade. They include,

- default styles applied by the browser for a given markup language
  - *eg: colours for links, size of headings, and so on...*
- styles specific to the current user of the document
  - *often affected by browser settings, device, mode...*
- styles linked to the document by the designer
  - *as we've seen, such styles can be linked in three ways including an external file, embedded in a definition at the beginning of the document, and as inline styles per element.*

The basic cascading nature of these options means that the following applies,

- browser's style will be default
- user's style will modify the browser's default style
- the styles of the document's designer will then modify the overall styles further

So, as we read documents in a browser, our styles might be applied in a cascading nature from multiple sources. The whole becomes the document that we see and use within our browser.

For example, a rendered document may include some styles from the browser's defaults for HTML. Then another part of the rendered style might come from customised browser settings or style definition files. For example, a user may customise their browser preferences or specify custom behaviours, which are then applied by the browser for rendering documents. Finally, we will also see styles applied from stylesheets linked to the document itself, or other embedded styles. It's not quite as simple as just looking at the linked CSS files.

So, to reiterate, for our styles in a cascade, a designer's stylesheets have priority, then user's stylesheets, and then the browser's own defaults.

- 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
     - *eg: 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*
- 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 also includes an interpretation of the concept of inheritance for its styles. So, descendants will inherit properties from their ancestors. For example, if we create a style on an element, all descendants of that element within the DOM will also inherit that style. This will apply unless the style is then overridden by another ruleset that specifically targets that element.

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

As `p` is a descendant of `body` in the DOM, it will inherit the background colour of the body. So, we can set our paragraphs' text colour to correctly show against the specified background colour.

This characteristic of CSS is an important feature, and it helps to reduce redundancy and repetition of styles. It is also another reason why it is useful to maintain an outline of the DOM structure for a given HTML document.

Again, however, there is a small caveat to this characteristic of CSS. Whilst most styles will happily follow this pattern, not all properties are inherited by default. For example, properties related to block-level elements are a notable issue with inheritance. **Margin, padding, and border rules are not inherited from ancestors.**

- 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 for our HTML document can be set for the body or within an element's specific ruleset. The first thing we need to do is specify our font-family,

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

The value for the font-family property specifies preferred and fall-back fonts for our document. If *Times New Roman* is not available, then the browser will try *Georgia* and *Serif*.

- Fonts can be set for the body or within an element's specific ruleset
- we need to do 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*

CSS Basics - fonts - part 2

We've already seen how we can change the colour of our text, but it's also useful to be able to modify the size of our fonts as well. For example,

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

With the above, we begin by setting the base font size to 100% of the font size for a user's web browser. This then allows us to scale our other fonts relative to this base size. So, we could use CSS absolute size values, such as `x-large`, which scales the size accordingly. Or, we could try relative sizes, such as `larger`, to help make our font sizes larger relative to the current context.

However, if we need better control of our font sizes, we can use `em`. These are meta-units, which represent a multiplier on the current font-size. They're derived from standard typography practices, which are based upon the standard width of an uppercase **M** in printing.

So, if the current font size has been set to `12px`, a font-size of `1.5em` will make the font actually an equivalent `18px`.

The obvious benefit of this approach, `em`, is that our text will scale according to the base font size. If we modify the size of the base, all font sizes set to `em` will also adjust accordingly.

Try different examples at

- W3 Schools
- 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*
  - *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 1 - CSS Fonts
- JSFiddle - CSS Fonts

CSS Basics - custom fonts

Using Fonts with CSS has often been a limiting experience, problematic at best, and reliant upon the installed fonts on a given user's local machine. There were workarounds, such as wrapping font files in JavaScript, and then serving from a remote server with the applicable HTML documents, but these were notoriously slow and buggy. They only tended to be employed, at best, as a final solution to a difficult problem.

However, with the advent of web fonts, the process of rendering with custom fonts is now considerably easier for designers and developers. We can deliver required fonts via the internet, using services such as Google's custom fonts.

- Google Fonts

From this site, we can pick and choose our custom fonts by selecting the *Quick-use* button. This loads a new page with options and instructions for using your chosen custom font. We then select our required character sets, add a `<link>` reference for the font to our HTML document, and then specify the fonts in our CSS. We need to add this new font as a font-family in our style sheet,

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

We can then style of our document's fonts as normal.

- 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
  - *pick and choose our custom fonts by selecting Quick-use*
  - *from the options, select*
  - *required character sets*
  - *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 2 - CSS Custom Fonts
- JSFiddle - CSS Custom Fonts

CSS Basics - reset options

To help us reduce browser defaults, we can use a CSS reset. Whilst often considered a rather controversial option, a reset allows us to start from scratch, and customise aspects of the rendering of our HTML documents in browsers.

They're often considered controversial for the following primary reasons.

- **accessibility** - we need to ensure that we cover options for accessibility, such as users navigating solely via a keyboard
- **performance** - another criticism is levelled at rendering performance. Resets are often reliant upon use of the universal selector, *, which can often lead to inefficient performance.
- **redundancy** - they can create a lot of redundancy within our styles, instead of simply relying upon some browser defaults

Therefore, use resets with care. One of the notable examples of resets is by Eric Meyer who originally discussed the logic behind this reset in a May 2007 blog post. He provides a stylesheet for resetting default styles, which can then be linked in our own documents. It can be a useful starting point for implementing such resets. Meyer also notes that resets often become part of frameworks and other libraries to help with overall consistency. At least you know you're starting from scratch, a blank canvas so to speak.

- help us reduce browser defaults, we can use a CSS reset
- often considered a rather controversial option

- reset allows us to start from scratch
- customise aspects of the rendering of our HTML documents in browsers
- 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 3 - CSS Reset Before

Demo - CSS Reset - After

Browser resets are implemented using the Eric Meyer stylesheet.

- Demo 4 - CSS Reset After

CSS - a return to inline styles

So, CSS, in some contexts at least, is being re-considered relative to dynamic styles generated and output from JavaScript. The styles are embedded as *inline* styles, but they are, in fact, generated and controlled from the application's JavaScript. Likewise, they can be updated and deleted as required by the application's logic.

With the rise of React, *inline* styles are oncemore gaining in popularity. For a long time, this style of CSS was avoided and remained the purview of novices, the lazy, or those who simply did not know any better.

However, for certain web applications they are now an option for maintaining and updating our styles. It should be noted, however, that their implementation is not the same as simply embedding styles in HTML. To re-iterate, they are dynamically generated, and can be removed and updated as part of our maintenance of the underlying DOM.

The redundant nature of the original inline embedding is no longer an issue with the use of JavaScript abstraction.

Some of the inherent benefits include the following,

- no cascade
  - *inline styles negate the global nature of CSS*
  - *cascade still exists, to some extent, but it is dramatically reduced*
- built using JavaScript
  - *helps maintain the same development environment*
  - *keeps logic, styles, rendering etc all in one place*

- *can be good or bad (depending upon your opinion)*

- styles are dynamic
  - *largely related to the concept of **state** in JavaScript*
  - *in effect, changes in dynamic conditions*
  - *influenced by user input, dynamic updates etc...*
  - *perceived beneficial to control style relative to state changes*

- *inline* styles are oncemore gaining in popularity
  - *helped by the rise of React*

- 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

There are also many detractors of this new implementation of inline styles. Whilst we can ignore the less pleasant comments, standard arguments include the following,

- 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, ie: 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
  - *psuedoclasses etc, media queries...*
- CSS can add, remove, modify classes
  - *dynamically update selectors using classes*

- 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, ie: 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
  - *psuedoclasses etc, media queries...*
- CSS can add, remove, modify classes
  - *dynamically update selectors using classes*

JS Intro

JavaScript is now a core, invaluable technology for client-side design and development. From vanilla JavaScript to the latest library, its growth as a development environment has exploded over the last few years. It is now being used as a powerful technology to help us rapidly prototype and develop web, mobile, and desktop applications.

We can use libraries such as JQuery, React, AngularJS, and Node.js to enable us to develop powerful, scaleable client and server applications. Using frameworks such as Apache Cordova, we can develop cross-platform applications for mobile devices. Finally, with GitHub's recent Electron project, for example, we can now leverage web technologies to build powerful desktop applications.

- 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*

JS Basics - operators

Operators are a particularly useful, and fundamental aspect of programming. It is no different with JavaScript.

They allow us to perform mathematical calculations, assign one thing to another, compare and contrast, and so on.

With the simple * operator, we can perform multiplication,

```
2 * 4
```

Likewise, we can add, subtract, and divide numbers as required within the logic of our applications. We can mix mathematical with simple assignment,

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

So, in this basic example, we are simply assigning the value 4 to a variable called a, and then calculate a value for the variable b using a + 2 resulting in a new total of 6 for b.

- 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

The following is a short summary of the more common operators currently used in JavaScript.

Assignment

- = eg: `a = 4`

Comparison

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

Compound assignment

- +=, -=, *=, /=
- compound operators combine a mathematical operation with assignment
- 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

In JavaScript, as with most forms of programming, we are able to express different representations of values often based upon a need or intention for that value. Such representations are known as **types** in common syntax.

JavaScript has **built-in** types, which allow us to represent **primitive** values. For example, if we need to perform a mathematical calculation we need and use **numbers**. For textual documents and output, we use **strings**, and to simply offer an option, **yes** or **no**, right or wrong, we can use a standard **boolean**, which allows us to represent a *true* or a *false* value.

Such values included in the source code are simply known as **literals**, and we can represent them as follows,

- string literals use double or single quotes eg: `"some text"` or `'some more text'`
- *numbers* and *booleans* are represented without being escaped, ie: they don't require encapsulating quotes... eg: `49`, `true;`

We can also consider and include arrays, objects, functions, and so on within our consideration of values and types for JavaScript. Each will be considered in detail later on.

- 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

In JS, we have the option and ability to convert, or more correctly **coerce** our values and types from one type to another. For example, if we have a number, which then needs to be printed, we can convert it, or coerce it, to a string. And, logically, we can perform the reverse for a string to a number.

JS provides different options for enforcing such coercion, including the following

```
var a = "49";
var b = Number(a);
```

In JS, variable a will be a string, and the resultant number coercion will create a new variable b as a number. The built-in JS function, `Number()`, is an explicit coercion, and allows us to convert any type to a number type.

There is also a less specific implicit coercion, which JS will often perform as part of a comparison. For example, if we compare

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

JS will do its best to force the correct answer by implicitly coercing the left string to a matching number, and then performing the comparison.

Whilst this is possible in JS, it's often considered bad practice and is something you should try to avoid where

possible. It's better to explicitly convert the type, and then perform the comparison.

However, there are rules that JS follows in trying to implement this implicit coercion. We'll look at these rules later on.

- 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

Often referenced as a **symbolic** container for values, and data, applications use such containers to keep track and update values during the various stages of an application. The easiest way to achieve this goal is to use a **variable** as a container for such values and data.

Variables allow values to vary over time, and JS is no different. However, one of the major differences lies in the way it declares its variables and assigns `type`. JS emphasises types for such values, and does not enforce them on the variable itself.

Known as **weak typing**, or **dynamic typing**, JS permits a variable to hold a value of any type. It can often be a benefit of the language, and a quick way to maintain flexibility in design and development.

- **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 emphasises 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

In JS, we declare a variable using the keyword `var`. and this declaration does not include any further necessary **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);
```

In the above example, we can see how `var a` maintains a running total of the value of a. Therefore, it is able to keep a record of these changes, and effectively the **state** of the value and its small part of the application.

In other words, **state** is keeping track of changes to any values in the application.

- 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

We can also use variables in JS to enable central, common references to our values and data. Better known in most languages simply as **constants**, such variables allow us to define and declare a variable with a value that is not intended to change throughout the application.

Such **constants** are often declared together, and form a store for values that can be abstracted for use throughout an app. If the value is later updated, this change ripples through the app to each reference to the variable.

As a convention, JS normally defines constants using uppercase letters,

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

We can also use multiple words in the naming convention for constants, and each word is separated using an underscore, _.

```
var TEMPLE_NAME = "Philae";
```

With the advent of ECMAScript 6, or ES6, there is now a new way to declare a constant, which uses the keyword `const` instead of `var`.

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

There are many different benefits to using constants,

foremost amongst them are the benefits of abstraction, and ensuring that the value is not accidentally changed. For example, if we tried to change the value of the above constant whilst the application was running, it would reject the change. In strict mode, this rejection would lead to the application failing with an error.

- use variables in JS to enable central, common references to our values and data
- better known in most languages simply as **constants**
- allow us to define and declare a variable with a value
  - *not intended to change throughout the application*
- **constants** are often declared together
- form a store for values abstracted for use throughout an app
- JS normally defines constants using uppercase letters,

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

- ECMAScript 6, ES6, uses the keyword `const` instead of `var`

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

- benefits of abstraction, ensuring value is not accidentally changed

JS Basics - comments

As with other languages, JS naturally includes the option to add comments within our code. There are currently two permitted implementations for comments,

single line comment

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

multi-line comment

```
/* this comment has more to say
hence the need for more lines... */
var b = "forty nine";
```

- 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
hence the need for more lines... */
var b = "forty nine";
```

CSS - test and try out

- JSFiddle - CSS Custom Fonts
- JSFiddle - CSS Fonts

Demos

- Demo 1 - CSS Fonts
- Demo 2 - CSS Custom Fonts
- Demo 3 - CSS Reset Before
- Demo 4 - CSS Reset After

References

CSS & Typography

- Eric Meyer - reset CSS
- MDN - CSS
- Cascading and inheritance
- Perishable Press - Barebones Web Templates
- Typography - EM units
- The Unicode Consortium
- Unicode Information
- Unicode examples
- W3 CSS
- W3 Schools - CSS
- W3 Schools - font-size

JavaScript & Libraries

- AngularJS
- Apache Cordova
- Electron
- JQuery
- MDN - JS
  - *MDN - JS Grammar and Types*
  - *MDN - JS Objects*
- Node.js
- React