Intermediate CSS & HTML

Jen Zajac

Layout Fundamentals

- Static elements sit in the 'normal' flow of the page
- Vertical margins get smooshed together
- Floats interact with each other until they get cleared

Block Formatting Contexts

- Those layout fundamentals are true within a block formatting context
- The <html> element creates a block formatting context
- But there are other ways of creating them too, via the application of certain properties

- position
  - absolute
  - fixed
- display
  - inline-block
  - table-cell
  - table
  - caption
  - flex
  - inline-flex
- overflow
  - hidden
  - scroll
  - auto

Block Formatting Contexts

- WITHIN EACH block formatting context
  - Floats interact with each other
  - Vertical margins collapse
- We can take advantage of this fact to clear elements without having to have a specific clearing element or clearfix class

Stacking Contexts

- If we have two elements on the page that overlap due to being positioned, the element further down the DOM is on top
- We can change the stacking order of elements by using the z-index property and giving a position:relative with no co-ordinates (or another non-static position)

Stacking Contexts

- So to stack something on top we'd put on an even higher z-index?
- Not quite that simple... the nesting matters.

Stacking Contexts

- A stacking context is created by applying one of these properties to the element:
  - z-index
  - opacity (when less than 1)
  - transform
- Each stacking context behaves like a group or folder which has its own position – the children of the element are 'nested' within it

Sidenote:

- If you slap on a 'z-index:10000' or something to that effect to make something 'just work', I will smite you!

Better use of selectors

- Use descendant selectors for better performance and to avoid getting yourself in a mess with things like “ul li ul li { }”
- Descendant selectors will only apply rules to immediate children of an element, not grandchildren

```
ul > li { /* style only first level li's*/ }
```

:before and :after content

- There are a couple of nice tricks you can do with this, e.g.:

```
a[href]:after {
    content: " (" attr(href) ")";
}

.next:after {
    content:" \25BA"; /* encoding for ► */
}
```

Semantic markup

- <button> is better than #button or .button
- If you need to disable it, use disabled=”true” and do your styling based using attribute selector

```
[disabled=true] { /* your styles */ }
```

- If you have to use something else at least give it ARIA role="button"

Semantic markup

- Use <nav> and <footer> and <article> etc. but only base your styling directly on those element selectors if you can be sure you'll only use them for one purpose... i.e. don't!
- Give them a more meaningful class name e.g. <nav class=”main-nav”>

And now for something completely different...

http://flukeout.github.io/

Maintainable CSS

- Many different theories, OOCSS, BEM etc.
- We'll focus on SMACSS, “Scalable and Modular Architecture for CSS”
- Not a rigid system, just a bunch of best practice suggestions that can be customised

SMACSS

- Five categories of CSS
  - Base
  - Layout
  - Module
  - State
  - Theme

SMACSS – Base styles

- Basic styling that applies to particular tags throughout whole site
- E.g. your reset stylesheet, body font declarations, font sizes for h1, h2 etc.
- HTML5boilerplate has a good set of starter base rules (but consider if you really need it)

SMACSS – Layout styles

- How the basic blocks on the page are organised e.g. header, footer, main content, sidebar etc.
- Will often contain media queries to handle responsive layout
- Differentiate from other styles by using ID selectors or prefixing classes with 'layout-'

SMACSS – Module styles

- Reusable components like media objects, a carousel, a nav bar
- Split modules out into their own stylesheets, include via @import
- Module elements should use descendant or child selectors of the module e.g.:

```
.megamenu .item .title { }
```

SMACSS – Subclassing modules

- Modules are re-usable – but what if you need a variant version?
- Give that element two classes, the original module class plus the subclass

```
<button class="action-button action-button-big">

.action-button.action-button-big { }
```

SMACSS – State styles

- State overrides other styling
- Often applied by JavaScript
- A module or layout element 'is-' something

```
<div class=”dropdown-menu is-open”>

.dropdown-menu.is-open { }
```

- Be careful about specificity

SMACSS – Theme styles

- Not the same thing as e.g. a Drupal theme
- Rather, it is something like a user-switchable color scheme or a location specific theme
- Generally only applies to things like colours, background images
- Use a separate stylesheet for each theme which overrides the normal styles

SCSS

- SCSS is a CSS preprocessor, a.k.a. SASS
- Disadvantages: compilation step!
  - Slows down testing of changes
  - How do we deploy the compiled file
  - How do we store the files in version control?

SCSS – Advantages

- Variables!
- Basic scripting and reusable components
- Better structure, nicer comments (// instead of /* */)
- Lots of tools we can leverage e.g. Compass

SCSS – Let's do some

- Open scss-example/index.html in a browser
- Open scss-example/scss/main.scss in an editor
- In a terminal, do 'cd scss-example' and then do 'gulp watch'

Nesting

```
header {
    padding:10px;

    h2 { font-style: italic; }

    &:hover {
        background-color:lightsteelblue;
    }
}
```

Variables

```
$brand-color: lightsteelblue;

header {
    padding:10px;

    h2 { font-style: italic; }

    &:hover {
        background-color:$brand-color;
    }
}

article {
    background-color:$brand-color;
}
```

Media Queries

```
$breakpoint: "only screen and (max-width : 320px)";

h2 {
    font-style: italic;

    @media #{$breakpoint} {
        font-size:100%;
    }
}
```

Cleverer @import

- Any file names that begin with an underscore will be imported into place during the compilation
- The 'partial' .scss file won't get converted to a .css file

@extend

```
.error {
  border: 1px #f00;
  background-color: #fdd;
}

.serious-error {
  @extend .error;
  border-width: 3px;
}
```

@mixin

```
@mixin large-text {
  font: {
    size: 120%;
    weight: bold;
  }
  color: #ff0000;
}

.highlighted-para {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

@mixin with arguments

```
@mixin fancy-border($color, $width: 1in) {
  border: {
    color: $color;
    width: $width;
    style: dashed;
  }
}

article { @include fancy-border(blue); }
```

@extend vs. @mixin

- @extend usually produces better compiled CSS, so use that when you can
- But if you need to pass in arguments you'll have to use @mixin

What is this gulp thing anyhow?

- Very flexible JavaScript based pipeline for precompilation
- LOTS of pre-prepared plugins to do things like help with vendor prefixes
- Other alternatives include things like Compass

Questions?