ISTA 230

CSS Frameworks & Sass

CSS

Browser Defaults

Consider the following HTML snippet:

HTML

```
<h1>Page Header</h1>
<p>This is my content.</p>
<p>This is more content.</p>
<p>...</p>
```

Even without any CSS applied, there are slight variations in the way these elements are displayed in the various web browsers we've discussed in this course.

Small differences in font size, padding, and margins can be easily spotted by overlaying the views from each browser.

These differences come from the default presentation styles included with these browsers. Depending on whether you're using Internet Explorer or Chrome, a heading might have slightly different margins, list items may or may not be indented, etc.

CSS

Common "Pre-Styled" Elements

Some common elements that have variation from one browser to the next are:

- Body (<body>)
- Headings (<h1>, <h2>, ...)
- Paragraphs (<p>)
- Lists (<ul>, <ol>)
- Links (<a>)
- More...

While often subtle, these inconsistencies can significantly impact the overall look and feel of a given website when viewed in different browsers, particularly when they have to do with padding and margins.

In the mid-2000s, a number of web developers began to address the presentation differences between browsers by including an initial CSS file that 'reset' all of the default styling to a baseline.

CSS

CSS Reset

The simplest CSS 'reset' files often looked like the following:

CSS - reset.css

```
body             { padding:0; margin:0; }
h1,h2,h3,h4,h5,h6 { padding:0; margin:0; }
p                { padding:0; margin:0; }
ul, ol, li       { padding:0; margin:0; }
```

Others felt that this wasn't inclusive enough and went so far as to use:

CSS - reset.css

```
* { padding:0; margin:0; }
```

While this did reset the padding and margins on everything, it also had unintended effects. Some browsers would removing the padding on form elements while others wouldn't apply this to form elements, making for a new set of inconsistencies.

CSS

CSS Reset

Rather than using the universal selector, web designer Eric Meyer developed a CSS reset which listed specific elements that shouldn't have any padding or margins. This provided the developer with a consistent baseline for most elements while not introducing additional inconsistencies.

CSS - reset.css

```
html,body,a,abbr,acronym,address,area,b,bdo,big,blockquote,button,
caption,cite,code,col,colgroup,dd,del,dfn,div,dl,dt,em,fieldset,
form,h1,h2,h3,h4,h5,h6,hr,i,img,ins,kbd,label,legend,li,map,object,
ol,p,param,pre,q,samp,small,span,strong,sub,sup,table,tbody,td,
textarea,tfoot,th,thead,tr,tt,ul,var
{
   margin:0;
   padding:0;
}
```

In addition to white space adjustments, Meyer's CSS reset file included additional typographical baseline settings as well as some default styling for lists and tables. The original CSS reset file can be found at http://meyerweb.com/eric/tools/css/reset/reset200802.css.

CSS

CSS Reset

To apply a 'reset, the CSS reset file is included as the first <link> element in the head, with your custom CSS file(s) following thereafter.

HTML

```
<head>
   <title>My Page</title>
   <link href="reset.css" type="text/stylesheet" rel="stylesheet" />
   <link href="style.css" type="text/stylesheet" rel="stylesheet" />
</head>
```

CSS

CSS Reset

- Pro: Consistency across all browsers.
- Pro: More control for designers.
- Con: Potentially more work required in developing CSS.
  For example, some CSS resets actually removed all bullets from list items.

CSS - reset.css

```
ol, ul
{
   list-style: none
}
```

If the designer wanted to use an unordered list, the list items would no longer have a default bullet. It would be up to the designer to remember and specify exactly how they wanted these elements to display.

- Con: Potential accessibility issues.
  For example, some CSS resets would remove the default underlining on links.

CSS - reset.css

```
a, a:link, a:visited, a:focus, a:hover, a:active
{
   color: #069;
   text-decoration: none;
}
```

The designer would need to remember to add in non-color-based visual cues for their links since the default was no longer applied.

CSS

CSS Frameworks

In addition to providing a common baseline for all browsers, CSS resets often contain some default styling of their own. As more styles are added, these morph from a pure 'reset' to more of a CSS framework. Frameworks consist of a set of pre-defined CSS rules, typically built using class selctors.

One of the most popular CSS frameworks is Bootstrap, a framework developed by a designer working at Twitter. For a full list of features available in the Bootstrap framework, visit http://getbootstrap.com/components/.

CSS

Bootstrap - Grid-Based Layouts

In addition to a number of useful design styles and elements, Bootstrap also provides us with the option of using a grid-based layout. This grid system is:

- Fluid-based
- Relatively easy to use
- Easy-to-use responsive design option

CSS

Bootstrap - Grid-Based Layouts

The Bootstrap grid system relies on a 12-column grid. In order to use this grid, the following approach must be used:

- All elements must be placed within a **.container** or a **.container-fluid**
- Use **.row** to create horizontal groupings of columns.

CSS

Bootstrap - Grid-Based Layouts

Lastly, each section of your site will require a class that specifies how many columns on the grid it should span. In order to do this, Bootstrap provides us with four device-based prefixes:

- **.col-xs-#** (< 768px)
- **.col-sm-#** (>= 768px)
- **.col-md-#** (>= 992px)
- **.col-lg-#** (>= 1200px)

These prefixes can be applied with an integer number from 1-12 which specifies how many columns it should span. For example:

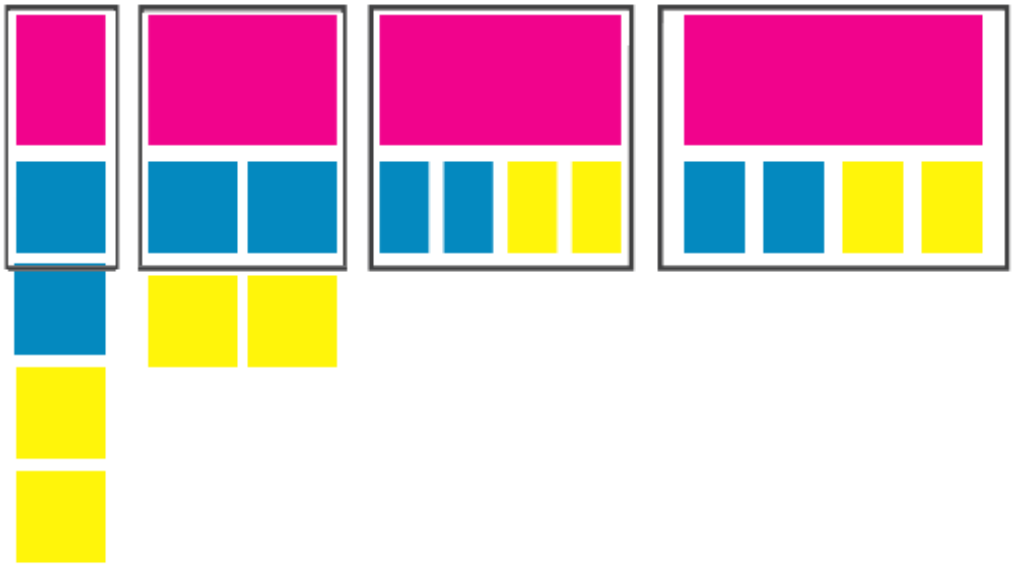
```
<nav class='col-xs-6'>
```

would indicate that the navigation element should span six columns on screens smaller than 768px.

CSS

Responsive Design - Common Patterns

As an example, consider the following design pattern from our lecture on responsive design:



Featured Items

At smaller resolutions, we'd want a single column. At slightly larger resolutions, we'd want an element that spans the width of the page, followed by two rows of two columns. On even larger screens, we'd want a row of four columns at the bottom of the page.

CSS

Bootstrap - Grid-Based Layouts

Using Bootstrap's grid system, this becomes a relatively trivial task.

All rows should be placed inside of a element with the class attribute 'container' (or 'container-fluid'). Each row should then be contained within a element with the class attribute 'row'.

Lastly, each section of the page should be placed inside of a element with class values matching the number of columns we want the element to span at a given resolution.

HTML

```
<div class="container">
  <div class="row">
     <div class="col-xs-12">...</div>
  </div>

  <div class="row">
    <div class="col-xs-12 col-sm-6 col-md-3">...</div>
    <div class="col-xs-12 col-sm-6 col-md-3">...</div>
    <div class="col-xs-12 col-sm-6 col-md-3">...</div>
    <div class="col-xs-12 col-sm-6 col-md-3">...</div>
  </div>
</div>
```

In the above example, we use elements for our layout. In a real-world case, we would likely use other block elements as well (<header>, <nav>, etc.).

CSS

Bootstrap - Responsive Design

Bootstrap also provides us with some useful classes for toggling whether certain elements should be visible at a given resolution.

- Classes provided to toggle visibility
    - .visible-xs or .hidden-xs
    - .visible-sm or .hidden-sm
    - .visible-md or .hidden-md
    - .visible-lg or .hidden-lg

CSS

CSS Frameworks - Minified

Often times, frameworks such as Bootstrap will optimize the size of their CSS file by removing all unnecessary white space. Frameworks will typically provide this optimized version of their framework along side the unoptimized version. Typically, the optimized (or 'minified') version of the file will follow the naming convention of filename.**min**.css.

CSS

CSS Frameworks

While CSS frameworks do offer a number of advantages, they are not without their downsides:

- Pro: Speeds up development!
- Pro: Cross-browser functionality
- Pro: Easy to make clean, responsive layouts
- Con: Applies styles whether you want them or not
- Con: Adds extra code (e.g. Bootstrap CSS file is 7000+ lines unminified)...

If you need to build a website fast and have limited resources, a framework can provide you with a **HUGE** head start. That said, CSS framework files are often large and can require a fair amount of additional bandwidth to download.

It is up to you as the designer to determine whether a framework is a good fit for your project.

CSS

CSS Preprocessors

CSS was designed to be accessible to the masses. The designers of CSS have intentionally not included some more advanced functionality that might prove to be a barrier to novice designers.

To help make up for these limitations, a number of CSS preprocessors have emerged to provide some additional functionality for web designers. The goal of these frameworks is to apply the DRY ("Don't Repeat Yourself") principle to CSS.

Preprocessors are programs that you install on your local computer or web server. While very similar to CSS, they have their own syntax and capabilities which take some getting used to. Rather than exploring all of the preprocessor options out there, we'll focus on Sass, the most popular CSS preprocessor at this time.

CSS

Sass - CSS Preprocessors

'SASS' stands for Syntactically Awesome Style Sheets. This preprocessor takes in a Sassy CSS file (with extension '.scss') and converts it into a valid CSS 3 file (with extension '.css').

While this process is usually done via command line, you have a much simpler option available to you. When you upload your .scss file to your ISTA 230 account, a small button that says 'Transform' will appear next to it. Clicking on this button will run the .scss file through the Sass preprocessor and create a .css file with the same filename.

For example, clicking 'Transform' for a file named 'styles.scss' will result in a file named 'styles.css' being created, based on the conent of the 'styles.scss' file.

CSS

Sass

So, what's the point of a CSS preprocessor? Imagine that you have a large CSS file with a primary color used throughout. If you were required to change that primary color, you would have to find each instance of the color in your CSS and modify it.

CSS - styles.css

```
body { background-color: #02BD1A; }
...
h1,h2,h3,h4,h5,h6 { color: #02BD1A; }
...
a { color: #02BD1A; }
```

While this is a seemingly trivial task, it could be made more complicated by the fact that you might have used uppercase letters in your hex codes in some places and lowercase in others. You might have also used hex codes in one property and RGB in another.

While this is a seemingly trivial task, it could be made more complicated by the fact that you might have used uppercase letters in your hex codes in some places and lowercase in others. You might have also used hex codes in one property and RGB in another.

Sass provides us with a better option through the use of variables:

CSS - styles.scss

```
$mainColor: #02BD1A;

body { background-color: $mainColor; }
...
h1,h2,h3,h4,h5,h6 { color: $mainColor; }
...
a { color: $mainColor; }
```

Sass variables start with a '$', followed by one or more alphanumeric characters, dashes, or underscores. These variables provide us with a way to use a single identifier to represent a value or values within our CSS while only specifying the value of the variable once.

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
body { background-color: #02BD1A; }
...
h1,h2,h3,h4,h5,h6 { color: #02BD1A; }
...
a { color: #02BD1A; }
```

CSS

Sass - Variables

Sass variables can store multiple values as well, making them useful for things like font families:

CSS - styles.scss

```
$font-serif: 'Trajan Pro', Georgia, 'Times New Roman', serif;
$font-sans: 'Liberation Sans', Helvetica, Arial, sans-serif;

h1,h2 { font-family: $font-serif; }

p { font-family: $font-sans; }
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1,h2 { font-family: 'Trajan Pro', Georgia, 'Times New Roman', serif; }

p { font-family: 'Liberation Sans', Helvetica, Arial, sans-serif; }
```

CSS

Sass - Comments

In addition to the CSS comment syntax that we've become accustomed to using, we can also include Sass-specific comments using '//'. These comments, while visible in the Sass file, are not included in the final CSS output, providing us with a way to include comments that we don't want anyone else to see.

CSS - styles.scss

```
/*
 * My Custom CSS Framework
 *
 * @license
 * Dual licensed under the MIT and GPL licenses.
 */

// Confession: I code better when
// listening to Taylor Swift...
body
{
   ...
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
/*
 * My Custom CSS Framework
 *
 * @license
 * Dual licensed under the MIT and GPL licenses.
 */

body
{
   ...
```

CSS

Sass - Functions

Sass also provides a number of functions for use within your Sass files. For example, the 'darken' function takes two values, a color and a percentage, and returns a color with the lightness decreased by that percentage.

CSS - styles.scss

```
$main-color: #333;
$light-color: #aaa;
$accent-color: #00a7e0;

h2.secondary
{
   color: $light-color;
   background-color: darken($accent-color, 30%);
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h2.secondary
{
   color: #aaa;
   background-color: #0079A1;
}
```

A full list of Sass functions can be found by visiting http://sass-lang.com/documentation/Sass/Script/Functions.html.

CSS

Sass - Nesting Properties

Sass also allows us to nest properties, making our CSS more readable. For example, there are a number of CSS properties that all begin with the prefix "font-". Each of these can be nested together within your Sass file, as seen in the example below:

CSS - styles.scss

```
header
{
   font
   {
      size: 140%
      style: italic;
      weight: bold;
      variant: small-caps
   }
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
header
{
   font-size: 140%
   font-style: italic;
   font-weight: bold;
   font-variant: small-caps
}
```

CSS

Sass - Nesting Rules

In addition to nesting properties, we can also nest related selectors and rules within Sass. For example, we can specify CSS rules for our <nav> element **AND** for <a> within that <nav> element using the following syntax:

CSS - styles.scss

```
nav
{
   padding: 0;
   margin: 0;

   a
   {
      color: #0f0;
   }
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
nav
{
   padding: 0;
   margin: 0;
}

nav a
{
   color: #0f0;
}
```

CSS

Sass - Nesting Rules

When nesting, it's often necessary to refer to the parent selector. You can do so using the '&' character.

CSS - styles.scss

```
a
{
   color: #a00;
   text-decoration: none;

   &:hover
   {
      color: #f00;
      text-decoration: underline;
   }
}
```

While this may seem trivial, having related selectors nested within one another creates a more logical connection than they would be using the standard CSS syntax.

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
a
{
   color: #a00;
   text-decoration: none;
}

a:hover
{
   color: #f00;
   text-decoration: underline;
}
```

CSS

Sass - Nesting

Being able to refer to the parent selector also allows us to create styles that override the default styles, as seen below:

CSS - styles.scss

```
h1
{
   font-size: 125%;

   // For our full-page layout
   body.special &
   {
      font-size: 400%;
   }
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1
{
   font-size: 125%;
}

body.special h1
{
   font-size: 400%;
}
```

CSS

Sass - Mixins

Sass also provides us with the concept of 'mixins'. Mixins allow the designer to create a set of styles that can be reused throughout their CSS without having to rewrite the same rules over and over.

Mixins are created using the '@mixin' directive followed by the name of the mixin, as seen below. To use a mixin within your CSS, you'll use the '@include' directive, followed by the name of the mixin you want to include.

CSS - styles.scss

```
@mixin large-text
{
   color: #000;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}

h1
{
   @include large-text;
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1
{
   color: #000;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}
```

While the above example may seem trivial, we could easily reuse all of the properties multiple times within our CSS property while still keeping our CSS dry.

CSS

Sass - Mixins

One of the great benefits of mixins is that they can take arguments. These arguments are specified using the Sass variable syntax in parentheses after the mixin name.

In the example below, we pass a single variable named '$color' to our mixin. We then use that variable within our mixin as we would any other Sass variable.

CSS - styles.scss

```
@mixin large-text($color)
{
   color: $color;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}

h1
{
   @include large-text(#036);
}
```

In the example above, we pass the value '#036' to our mixin. When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1
{
   color: #036;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}
```

CSS

Sass - Mixins

We can also pass multiple variables to our Sass mixins, giving us a lot of flexibility in how we use them in our CSS.

CSS - styles.scss

```
@mixin large-text($color, $background)
{
   color: $color;
   background-color: $background;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}

h1
{
   @include large-text(#fff, #036);
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1
{
   color: #fff;
   background-color: #036;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}
```

CSS

Sass - Mixins

We can also set default value for our mixin variables. This allows us to define a standard value that will be used if the user doesn't specify a value for every variable.

In the example below, the mixin has two variables, '$color' and '$background'. '$background' is set to have a default value of '#fff'. When the mixin is called, we've provided only a value for '$color'.

CSS - styles.scss

```
@mixin large-text($color, $background: #fff)
{
   color: $color;
   background-color: $background;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}

h1
{
   @include large-text(#036);
}
```

When the above Sass file is transformed, it will produce the following CSS file. Note that the output contains both the value we passed for color as well as the default value for background.

CSS - styles.css

```
h1
{
   color: #036;
   background-color: #fff;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}
```

CSS

Sass - Mixins

We can specify default values for all variables for a given mixin. While this allows us to define a standard value for every variable, it requires that we specify the variable name (or names) we want to change if we want to pass along a non-default value for some (but not all) of the mixin variables.

CSS - styles.scss

```
@mixin large-text($color: #000, $background: #fff)
{
   color: $color;
   background-color: $background;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}

h1
{
   @include large-text($background: #a00);
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
h1
{
   color: #000;
   background-color: #a00;
   font-size: 150%;
   font-weight: bold;
   text-transform: uppercase;
}
```

CSS

Sass - Mixins

Mixins are particularly useful when it comes to including CSS 3 properites that require a browser prefix. For example, we can create a mixin for linear gradients that looks something like the following:

CSS - styles.scss

```
@mixin linear-gradient($from, $to)
{
   background-color: $to;
   background-image: -webkit-linear-gradient($from, $to);
   background-image:    -moz-linear-gradient($from, $to);
   background-image:     -ms-linear-gradient($from, $to);
   background-image:      -o-linear-gradient($from, $to);
   background-image:         linear-gradient($from, $to);
}

body
{
   @include linear-gradient(#000, #999);
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
body
{
   background-color: #000;
   background-image: -webkit-linear-gradient(#000, #999);
   background-image:    -moz-linear-gradient(#000, #999);
   background-image:     -ms-linear-gradient(#000, #999);
   background-image:      -o-linear-gradient(#000, #999);
   background-image:         linear-gradient(#000, #999);
}
```

While this doesn't prevent our final CSS file being cluttered with browser prefixes, it does provide us with a more readable alternative in our Sass file.

CSS

Sass - Import

With mixins being as powerful as they are, it is not surprising that developers often define a number of mixins that they regularly use on all of their projects. To keep these mixins reusable, it is not uncommon for developers to create a separate file specifically for their mixins.

This file can then be included using the '@import' directive, as seen below:

CSS - mixins.scss

```
@mixin linear-gradient($from, $to) { ... }
@mixin rounded-corners($radius) { ... }
@mixin transition($property, $duration) { ... }
...
```

CSS - styles.scss

```
@import mixins.scss

body
{
   @include linear-gradient(#000, #999);
...
```

CSS

Sass - Import

Some developers take this one step further, creating individual files for their reset CSS, their Sass variables, and their Sass mixins. That said, it is ultimately up to you as the designer to determine how you'd like to arrange your Sass files.

CSS - styles.scss

```
@import reset.scss
@import variables.scss
@import mixins.scss

body
{
   @include linear-gradient($main-color, lighten($main-color, 10%));
...
```

CSS

Sass - Extend

Sass also provides us with the '@extend' directive, which allows us to include the styles from another CSS selector.

In the example below, we want to make sure that the '.alert-success' selector has the same styles as the '.alert-default' selector. Using the '@extend' directive, we can easily associate these two:

CSS - styles.scss

```
.alert-default
{
   border-radius: 5px;
   border: 1px solid #333;
}

.alert-success
{
   @extend .alert-default;
   border-color: #0a0;
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
.alert-default, .alert-success
{
   border-radius: 5px;
   border: 1px solid #333;
}

.alert-success
{
   border-color: #0a0;
}
```

CSS

Sass - Placeholders

With the '@extend' directive, we can also include 'placeholders'. Placeholders are essentially a selector that's not output when your Sass file is compiled but can be used with the @extend directive.

Placeholders are written exactly as you would any CSS id selector but with a '%' instead of a '#'. In the example below, we want to create a placeholder for centering block elements on the page. We create a placeholder named '%centeredBlock' which can then be used throughout our Sass file.

CSS - styles.scss

```
%centeredBlock
{
   display: block;
   margin-left: auto;
   margin-right: auto;
}

.mainContent
{
   @extend centeredBlock;
}

.frontImage
{
   @extend centeredBlock;
}
```

When the above Sass file is transformed, it will produce the following CSS file:

CSS - styles.css

```
.mainContent, .frontImage
{
   display: block;
   margin-left: auto;
   margin-right: auto;
}
```

CSS

Sass - Mixin vs. Extend

Because the functionality of @include and @extend are similar, students often ask which they should use for any given case. While there is no firm answer, it helps to remember the key differences bewteen the two:

- @include
    - adds new **CSS rule(s)**
    - allows for variables to be passed
    - duplicates code
- @extend
    - adds new **CSS selector(s)**
    - does **NOT** allow for variables to be passed
    - doesn't duplicate code

The best way to determine which you should use is to ask yourself the following question:

- Do I need to be able to pass variables?
    - If yes, use @include
    - If no, use @extend

CSS

CSS Preprocessors

- Pro: Adds lots of useful functionality
- Pro: Can make your CSS dry
- Pro: Time saver!
- Pro: Better organization, easier maintenance
- Con: Extra step in your workflow

Given the number of benefits (and relatively small number of negatives), I haven't found a good argument for not using Sass on web design projects. While it does take some getting used to, I highly recommend you start incorporating it into your design process!