ADVANCED CSS WDD 331

Getting started with Sass

Verify Setup

During the first week of the semester you were asked to install several things (Software setup). You will have a really hard time completing this activity if you did not get that done. The first thing that we need to do is verify that your Sass is working correctly.

Open up a command prompt on your computer (Mac: open the Terminal app, PC: Start->Run->CMD) and type the following to make sure that SASS is installed and running correctly: `sass -v`. You should see something like: `Sass 3.4.21 (Selective Steve)`. Your version might be different than this...that is fine. You should just not get an error. If you see an error go revisit the Software setup page

Note!

Sass actually has two versions of it's syntax. The original syntax is called just Sass and uses a `.sass` extention on your files. The other syntax which has become more popular and which we will be using is the SCSS syntax with a `.scss` file extention. The SCSS syntax is still considered Sass...and the compiler we just installed will work for either. The file extension will tell it what it needs to know.

SCSS syntax

Variables

The first past of Sass we will discuss is actually one of my favorite parts. Even if we gained nothing else from Sass than variables it would still be awesome. A variable is really nothing more than a way to store a *value* with an *identifier* (variable name) that we can re-use over and over in our code.

Here is a pretty common scenario. You are nearing the end of a large project and have over a thousand lines of CSS to show for it. In a meeting with the client they suddenly declare that the color they were so in love with and that you used *everywhere* as a result, they don't like so much anymore. Of course it can't be as simple as replacing the color everywhere that it is used with something else...they just want *less* of it. :) With variables this becomes really simple, without, it could be hours of work.

To create or use a variable in SCSS we need to use a special character: the '$'. This lets the Sass compiler know that what follows is going to be a variable name. Here is an example:

```scss
// Variable assignment
$my-variable: 42px;

// Variable usage
.foo {
     width: $my-variable;
}
```

01 Create HTML file

For this exercise we will use SCSS to style a simple contact form. Begin by creating a file called `scss-demo.html` in your editor and copy the following code into it.

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>Scss Demo</title>
    <meta name="viewport" content="width=device-width, initial-
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>

<body>
    <h1>Contact Form</h1>
    <form method="post" action="#" id="contactform">
        <fieldset>
            <legend>Contact me</legend>
            <div>
                <label for="name">Name:</label>
                <input type="text" name="name" id="name" size="
            </div>
            <div>
                <label for="email">Email:</label>
                <input type="email" name="email" id="email" si
            </div>
            <div>
                <label for="subject">Subject:</label>
                <input type="text" name="subject" id="subject"
            </div>
            <div>
                <label for="message"> Message:</label>
                <textarea name="message" id="message" cols="50
            </div>
            <input type="submit" value="Submit">
        </fieldset>
    </form>
</body>
```

```
        </html>
```

02 Declaring SCSS variables

Create a file called `styles.scss` in your editor. Add a comment at the top to indicate that variables will follow.

> Remember that for SCSS you can either use the traditional CSS commenting syntax: `/* comment */` or the SCSS syntax: `// comment` . The SCSS comments however will **not** be included in the final .css that is generated.

Lets start by defining some variables to hold the colors that we would like to use to style our form. Create 4 variables that will contain the body background color, form background color, headline color, and input box color. Below is an example of what mine looks like:

```
            //colors
            $body-background-color: darkred;
            $form-background-color: white;
            $input-color: white;
            $headlines-color : white;
```

> Be aware that Sass treats dashes '-' and underscores '_' the same in variable names. So `$body-background-color` and `$body_background_color` would refer to the same variable.

Variables can be used for values, properties, or even to build selectors. In some cases however we need to have the variable evaluated and replaced with it's value before it's used (Like if you wanted to use a variable to build a selector for example). This is called interpolation. Here is an article explaining more: All you ever need to know about SASS interpolation.

03 Using SCSS variables

Lets now use the body background color variable to change the background of the page. We use a variable just like we defined it. Everywhere that we would have use "darkred" we can now use `$body-background-color`. An example of this might look like:

```
body {
    background-color: $body-background-color;
}
```

You probably noticed that when we added the rule to change the body background color with our variable nothing happened. We are writing SCSS (in the `styles.scss` file) which the browser does not understand. Notice as well that the html we started with above is looking for a `styles.css` file. Our SCSS has to be converted back to plain CSS for it to actually work. This is why we had to install the SASS program onto our computers.

CSS custom properties

Variables used to be only possible with a preprocessor, but CSS now has the feature built in natively with custom properties. Now that we can do this with CSS is there a reason to do it in SCSS? Take a minute and read about the Difference between types of CSS variables, then come back and finish the activity.

04 Compiling SCSS

Go back to your command prompt and change the working directory to the directory that contains your files for this exercise.

If you are unsure how to change directories on you computer check out one of these resources: Command Line on the Mac or Command Line on the PC

Once there we need to give Sass the command to convert our .scss file to a .css file. Type the following:

```
sass styles.scss styles.css
```

If you are on Windows and you get an error when you try the sass command you may need to run the Ruby version of CMD. Do Start->Run->Ruby and choose the Ruby version of cmd and it should work.

The first part (sass) is the command. It takes 2 parameters. The first is the file to convert, the second is the name we want for the converted file. If you do not see any errors then open (or refresh) your page in the browser and you should now see your background color taking effect. You will also see a styles.css file that was created with the converted SCSS. Notice that everything in there now is plain CSS. Your variable is gone. It is important that you never edit the styles.css directly. It will get entirely overwritten each time you convert your .scss file.

If you did see errors then goto the line indicated and fix them. Then run the sass command again.

There are some great functions to manipulate color in Sass. Try using `lighten()`, `darken()`, or `mix()` for example. In fact there is a webcast on sitepoint that talks about creating a color scheme with Sass color functions.

05 Nesting

If we wanted to make changes to the `<input>`s in our form we might write something like the following in CSS:

```
#contactform {
    background-color: $form-background-color;
}
#contactform input, #contactform textarea {
    background-color: $input-color;
}
```

This works fine, but over time as we add more and more rules to our css, related rules tend to get separated, and this can be hard to maintain. This is where nesting comes in. Nesting is a feature of preprocessors that allows us to write rule sets inside of other rule sets. Lets look at an example. Using nesting the CSS above could be re-written as follows:

```
#contactform {
    background-color: $form-background-color;
    input, textarea {
        background-color: $input-color;
        flex: 2;
    }
}
```

The major advantage of nesting is really in helping to organize your code. With that in mind a caution: **Don't go crazy with nesting!** Nesting too many levels deep can actually make code harder to follow and read.

Nesting can be done with either selectors or contexts like `@media` or `@supports`. This in my opinion is where nesting really shines. Lets say for example that we want to have our form fill 100% of the width of the screen at small sizes, but only take up

90% on medium with a max-width of 700px. Perfect job for a media query. Lets look and see what that would look like with nesting.

```scss
#contactform {
    background-color: $form-background-color;
    input, textarea {
        background-color: $input-color;
        flex: 2;
    }
    @media (min-width: $break-small) {
       & {
          width: 90%;
          max-width: 700px;
          margin-left: auto;
          margin-right: auto;
        }
    }
}
```

If the above code is not making sense make sure to compile it to css. Review the css generated and compare it to the SCSS to help you understand what is going on.

Notice the '&' above. This is the ampersand selector (or parent selector reference). This always refers to the parent selector as the name implies. In our case `& = #contactform`.

Having all of the css for an element in one place can be very helpful. Add the media query code above to your scss file. Make sure to define variables for your breakpoints as well. Also add a large screen breakpoint as well that will make width: 70% and max-width: 900px

06 Mixins

There tends to be a lot of duplication in css. We tend to solve the same problems over and over from project to project, and even within the same project! Then you add in prefixing and developers end up writing a lot of the same css over and over. Enter Sass Mixins. These allow us to define collections of reusable css snippets. The best way to wrap your mind around this would be with an example.

Lets say we wanted to use flexbox on a page. To create a flex parent or container we might add the following css

```
.flex-container {
  display: flex;
  flex-flow: row wrap;
}
```

But...since older versions of most browsers don't recognize display: flex, in order to get the highest percentage of compatibility we should add all the prefixed flex declarations. that would make our code look more like this:

```
.flex-container {
 display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
    -webkit-flex-flow: row wrap;
    -ms-flex-flow: row wrap;
    flex-flow: row wrap;
}
```

That adds quite a bit of additional markup each time we need a flex container. Lets create a mixin to save us some typing.

```
@mixin flex-container() {
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
    -webkit-flex-flow: row wrap;
    -ms-flex-flow: row wrap;
    flex-flow: row wrap;
}
```

Now whenever we need to create flex-container we can simply do something like this:

```
fieldset > div {
    @include flex-container;
}
```

and it will generate the following css:

```
fieldset > div {
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
    -webkit-flex-flow: row wrap;
    -ms-flex-flow: row wrap;
    flex-flow: row wrap;
}
```

This is a nice time saver, but what if we need a column layout instead of row? This mixin would not work for that as it is. We need to modify it to take a *parameter* or two. Parameters can let us decide things such as whether we want row or column, wrap or nowrap when we use the mixin. Modified it would look like this:

```
@mixin flex-container($axis, $wrap: nowrap) {
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
    -webkit-flex-flow: $axis $wrap;
    -ms-flex-flow: $axis $wrap;
    flex-flow: $axis $wrap;
}
```

This mixin now accepts 1 or 2 parameters. Notice that we have provided a value on the second: nowrap. If we do not send 2 parameters then it will default to nowrap for the second. To use this we would do the following (assuming we want column nowrap as our flex-flow):

```
fieldset > div {
    @include flex-container(column);
}
```

The parameter gets entered in ( ) after the mixin name. If we wanted row wrap as our options instead we would do this:

```
fieldset > div {
    @include flex-container(row, wrap);
}
```

Lets take a look at another example as well.

```
.center {
  width: 90%;
  max-width: 960px;
  margin-left: auto;
  margin-right: auto;
}
```

This code would center a block element on a page. We might use this, or something very similar many times on a typical page. Lets see how it would look as a mixin.

```
@mixin center {
  width: 90%;
  max-width: 960px;
  margin-left: auto;
  margin-right: auto;
}
#content {
    @include center;
}
```

This is helpful, but can still only be used when we want a box that will fill most of the screen. Lets use parameters to make it more reusable and useful in a wider variety of cases.

Modify the center mixin to take 2 parameters that will be used to specify width and max-width. Max-width should be an optional parameter (It should have a default value of 960px). Then replace the code that we added earlier to center the form on medium and large screens with the mixin.

So are you tired of typing out

```
sass styles.scss styles.css
```

Everytime you make a change to your scss? One shortcut that you may have discovered is you can hit the up arrow on your keyboard to go back and see commands you have previously run. This can be very helpful. Another option would be to learn another SASS command. We can tell it to watch for changes in a file, and when it sees any to automatically convert the file to css. The command to do this would look like the following:

```
sass --watch styles.scss:styles.css
```

When you are done working you should tell it to stop watching as well. You do this by hitting `ctrl+c` on your keyboard while in the command/terminal window.

07 Mixins and @media (@content)

Media queries are powerful tools for building responsive websites, but they can really cause problems with the organization and readability of your css. We tend to make large blocks of changes for each query far from the original declaration of the rule. Often it can be helpful to have the default rules and all of the breakpoints next to each other. Mixins can help with this. Take the following code for example:

```
//set breakpoint sizes
$break-small: 20em;
$break-medium: 31em;
$break-large: 60em;

@mixin breakpoint($break) {
@if $break == large {
```

```
@media (max-width: $break-large) { @content; }
}
@else if $break == medium {
@media (max-width: $break-medium) { @content; }
}
@else if $break == small {
@media (max-width: $break-small) { @content; }
} @else {
@media ($break) { @content; }
}
}
```

you should be able to recognize the start of the mixin, but there a couple of new things there as well. First notice the `@if $break == large`. This is a conditional statement. Basically it reads like this: if the value in the variable $break is equal to "large" then do the following.

The other new concept is the `@content` statement. @content is a powerful feature of Sass, it allows us to pass css into our mixins! This in important with this example in particular since a breakpoint mixin would not be very helpful if all the breakpoints did the same thing.

> If you are having a hard time visualizing what that SCSS code above will produce remember that once you have compiled your SCSS you can look at the generated CSS. Comparing the written SCSS with the generated CSS can be a powerful tool to help you wrap your mind around some of these concepts.

The best way to understand this is with and example of how the breakpoint mixin would be used.

```
body {
background: white;
@include breakpoint(large) { background: red; }
@include breakpoint(medium) { background: cyan; }
@include breakpoint(small) { background: green; }
@include breakpoint(“max-width: 480px”) { background: yellow; }
}
```

the output of this once compiled to css would be the following:

```
body {
    background: white;
}
@media (max-width: 60em) {
     body {
        background: red;
    }
}
@media (max-width: 31em) {
    body {
        background: cyan;
    }
}
@media (max-width: 20em) {
    body {
        background: green;
    }
}
@media (max-width: 480px) {
    body {
        background: yellow;
    }
}
```

Notice that the css that was inside the { } after the @include for the mixin got inserted where the @content placeholder was. Whenever you use @content you will be expected to have those braces with some css. If you do not have the @content placeholder in your mixin you will not use braces when you use (@include) it.

Our last task with this activity is to add a breakpoint mixin to your scss. Then use it to replace the nested @media commands we added earlier. Lets also use it to do the following: One small screens we want the input boxes to take 100% width, and the labels should be above the input. One medium and large screens the label should be to the left side of the input. Here is a screenshot of the small and large versions of the form.