INTRODUCTION TO SASS

Syntactically Awesome Style Sheets (SASS)

What is SASS?

- SASS is a popular preprocessor to CSS
- Sass allows the use of the concept of DRY (don't repeat yourself)
- LESS is another preprocessor competitor to SASS (you would only need to learn one CSS preprocessor language).

Syllabus:

- Introduction
- SASS vs SCSS vs CSS
- Setup (Mac)
- SASS Hello World
- Sass Concepts
- Comments
- SASS Variables & CSS Variables
- Nesting
- Partials & Imports
- Mix-ins (Functions, @Content, @If, @for, @each & @while)
- Extend/Inheritance (@extend)
- Operators

- Folder Structures (Best Practice)
- SASS Frameworks

Useful Links:

- http://sass-lang.com/
- https://smacss.com/
- http://caniuse.com/
- https://www.youtube.com/watch?v=S4mPsoZ7sG4&index=3&list=PLUoqTnNH-2XxOt7UsKlTqbfrA2ucGosCR (Brad Hussey)
- https://www.youtube.com/watch?v=P1G4_zxOxtk (Help People)
- https://www.youtube.com/watch?v=St5B7hnMLjg (The net ninja)
- https://www.youtube.com/watch?v=fbVD32w1oTo&list=RDQMITWf3p81b5Q&index=1 (LevelUpTuts)

Introduction & Overview:

What is a preprocessor? A scripting language that extends CSS and gets compiled into regular CSS syntax.

A ".sass or .scss" file is a preprocessed file that will later be converted to a ".css" file that the browser can understand.

Advantages of using SASS:

- Cleaner, reusable and extendable code
- Very easy syntax to learn
- Features like mixins, variables, extends, imports etc. (CSS like a programming language)
- More efficient workflow (especially for larger projects)

We will be looking at version 3 of SASS which introduces SCSS (Sassy CSS).

SASS VS SCSS VS CSS:

- Sass is a new syntax, it has no curly braces or semicolons and uses indentation & nesting for hierarchy.
- SCSS is like vanilla CSS i.e. it has no new syntax to learn (valid CSS == valid SCSS) however it still access to the features of SASS such as mixins, variables etc.
- CSS is the original language that requires you to type everything, it has no mixins, variables etc. and can become very messy code.

We will use SASS syntax as it is a different syntax but once we have learned it we are able to use either SASS or SCSS (SCSS is the same as SASS but uses curly braces { } and semicolons ; in its syntax just like CSS).

Setup (Mac):

Go to http://sass-lang.com/ to follow the guide on installing SASS onto your computer.

SASS requires ruby dependency but if you are using a Mac, Ruby comes pre-installed on your machine. You can installs SASS either using a Application (e.g. Koala) or through the command line. Ruby uses gems as its package manager.

Install SASS:

1. Open Terminal
2. Install Sass using the terminal command (*sudo for super user do*):

   ```
   sudo gem install sass
   ```

3. Double check SASS is installed on your machine by typing the command:

   ```
   sass -v
   ```

It should return Sass version number e.g: `sass 3.5.4`

You should now be ready to use SASS in your projects.

Please refer to the sass-lang.com/install page to view installation instruction for other operating systems (OS).

SASS Hello World:

In this example we will be creating our first sass file to understand how our sass file is compiled to spit out our css file for our Hello World html page. The project files can be found in the Sass Hello World Example folder.

We can write our SASS codes in any editor of your choice (we will be using CS Code/ Atom as our text editor).

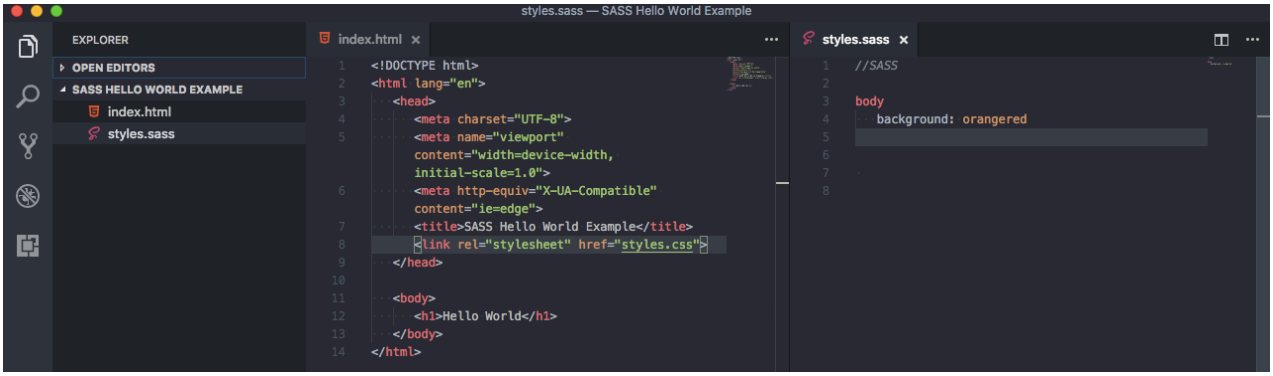
SASS require a compiler to compile the preprocessed .sass (or .scss) files into .css files. We can use either a GUI Application such as:

- Koala http://koala-app.com/ (Free)
- Codekit https://codekitapp.com/ (Paid/Free Trial (b*uy now popup every 15mins*))

or we can use the terminal as our sass compiler. In this example we will be using the terminal to compile our .sass file into .css code.

In the SASS Hello World Example folder we have two files:

- index.html (within <Body> tags we have Hello World in <h1> tags)



- styles.sass (our sass file to make the body-colour of our html page orangered)

We require Terminal (or GUI application) to compile our sass code into plain css code which the browser can read. To do this we need to open up Terminal and cd into the directory containing our project files (.sass file). VS Code has an integrated terminal (*view>integrated terminal*) and this by default cd to the folder opened within the VS Explorer menu.

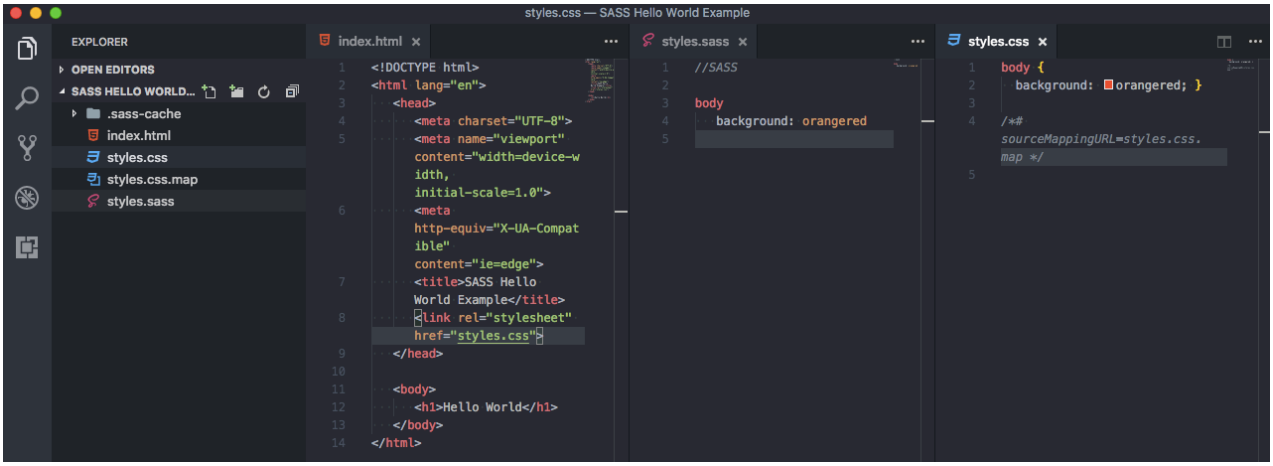
We now need to run the sass --watch command and add two arguments:

1. Which file/folder to watch and
2. Where to compile the css code.

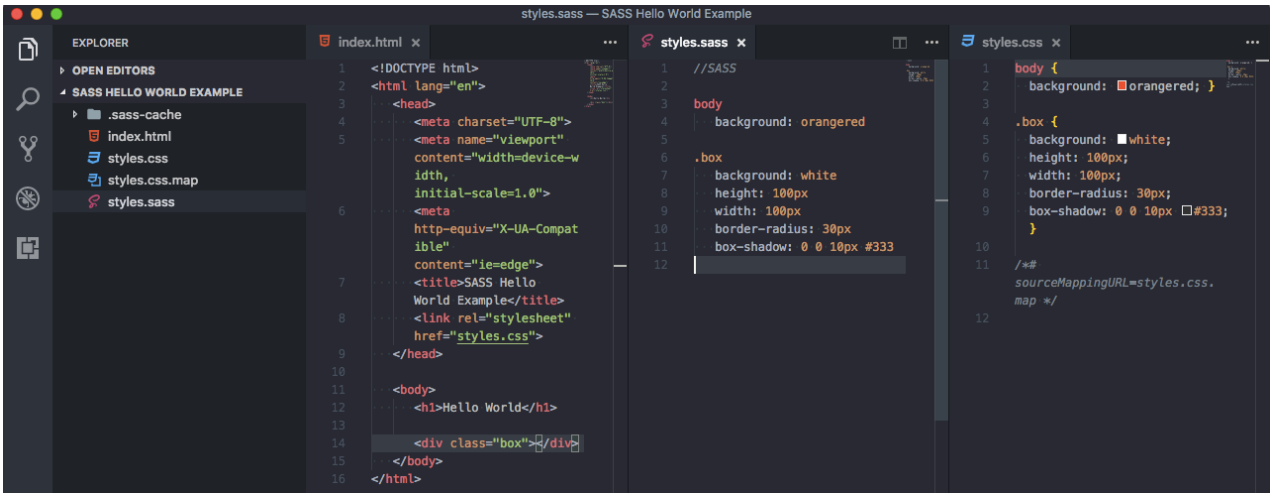
The sass command in the terminal would look something like:

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

This will create a styles.css file if one does not already exist and a styles.css.map (mapping file ignore but do not delete).



Leave the terminal running in the background and the sass watch command will continue to run in the background and look for any changes made to the .sass file and compile it to the .css file (i.e. every time you press save to the .sass file). In the example above we added a new box style to the .sass file and pressed save to update the .css file.



To end the sass watch command simply press Ctrl+C in the terminal.

SASS Concept:

Vanilla CSS on its own can be time consuming and while stylesheets are becoming larger and more complex, inherently they become harder to maintain. This is where a preprocessor such as SASS or LESS can help. SASS allows you to use features that are not in CSS yet like nesting, mixins, inheritance and other programming language concepts.

It must be noted that CSS is forever evolving and in the near distant future most of the features in SCSS will become part of vanilla CSS, the most recent addition to CSS being variables. Although we have variables within CSS, we can continue to use SASS for its many more useful features such as mixins and also manage/maintain our CSS codes for larger projects.

We will look at the many SASS features below and how we can utilise these SASS features/concepts in our projects. Finally, we will look at SASS best practice for folder/file structures for our sass and css files to manage larger and modular CSS projects.

SASS Comments:

There are two methods of adding comments to your sass code:

1. SASS comments

```
//SASS Comment - this is a private one line comment
```

2. CSS Comments — add forward slash and asterisk /* */

```
/*CSS Comment - this is a public multiline comment */
```

SASS comments are single line and so requires the double forward slash every time you add a return in your code, whereas CSS comments are multi-line. SASS comments are not compiled to the .css file and so are considered to be private comments that only the developer will see whereas CSS comments are made available to the public to see within the .css file.

SASS Variables:

SASS Variables (just like in many other programming languages) are a way to store information that you can reuse throughout your stylesheets. You can store any CSS property values such as colours, font-stack etc. To declare a variable we would the $ symbol before the name of the variable — for example:

```
$background-colour: #333
$font-colour: white
```

To use the variables declared above we would write the variable name next to the CSS property — for example:

```
body
    background: $background-colour
    color: $font-colour
```

View the SASS Comments & Variables project folder as an example of this concept.

Vanilla CSS Variables:

The syntax for creating a variable in CSS is completely different to SASS. Here is an example variable in CSS:

```
--background-colour: #333
```

To call this variable within our CSS we must type:

```
color: var(—background-colour);
```

Note: all valid CSS syntax are valid SCSS syntax - therefore we can use this syntax within our SCSS code but not within our SASS code (*SASS uses a different syntax*). SCSS also uses the $ symbol just like SASS to declare variables but have a semicolon ; at the end — for example:

```
$background-colour: #333;
$font-colour: white;
```

Nesting:

Nesting allows you to view a clear visual hierarchy of your HTML elements. CSS does not allow for nesting elements. However, using SASS we can nest elements as we would do in HTML. Here is an example of SASS syntax for nesting a websites navigation:

SASS

```
nav
  ul
    margin: 0
    padding: 0
    list style: none
  li
    display: inline-block
  a
    display: block
    padding: 6px 12px
    text-decorator: none
```

SCSS

```
nav{
  ul{
    margin: 0;
    padding: 0;
    list style: none;
  }
  li {
    display: inline-block;
  }
  a{
        display: block;
        padding: 6px 12px;
        text-decorator: none;
  }
}
```

[*Note: the difference between SASS and SCSS syntax is that SCSS is similar to CSS i.e it adds the curly braces { } and semicolons ; to its syntax whereas SASS does not.*]

This will produce the CSS code:

```
nav ul{
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li{
  display: block;
}
nav a{
display: block
padding: 6px 12px
text-decoration: none;
```

```
}
```

Partials & Imports:

Partials allows you to create partial SASS files that contain little snippets of CSS that you can include in your SASS file. Partials allows you to modularise your CSS which helps maintain your SASS/CSS, this is especially important for larger projects.

To create a partial file in SASS simply add a leading underscore for example:

```
_partial.sass or _partial.scss
```

The underscore lets SASS know that the file is a partial file and it should not be generated into a CSS file (SASS will ignore all partial files i.e files with underscores).

The SASS partials are used with the `@import` directive.

CSS has an import option that lets you split your CSS into smaller more maintainable portion. The drawback is that each time you use @import in your CSS it creates another HTTP request.

SASS import builds on top of the current CSS @import but SASS will take all the file(s) that you want to import and combine it with the file you're importing into so that you can serve a single CSS file to the web browser (i.e your do not require multiple HTTP request unlike CSS which will speed up your websites/web-apps).

To import a file we would use the @import followed by the file name. Note you do not need to add the file extension because SASS is smart and will figure this out for you.

Import example:

```
_partial.sass
base.sass
```

In the base.sass file we will add:

```
@import partial
```

Note: The ordering of @imports are important because the browser reads CSS from top to bottom and therefore dependencies should be loaded in order.

For example if we have a _partial.sass file that has dependencies on the _variables.sass file then our base should import in the following order:

```
@import “variables”
@import “partial”
```

When running the sass --watch command we should now make it watch for whole directories rather than individual files as this will cause errors especially where there are more than one SASS files and dependencies. To run the sass —watch command for a whole directory we would enter in the terminal:

```
sass --watch sass:css
```

View the SASS Partials & Import project folder as an example of this concept.

Mixins:

A Mixin allows you to create functions that you can reuse throughout your site. You can pass values/variables to make your mixin more flexible. A good example is for vendor prefixes for example border-radius.

To create a mixin we would add the = symbol followed by the mixing name. For example we will create a border-radius mixin which we can reuse within our SASS files.

```
=myFirstMixin()
      border-radius: 10px
```

To call this mixin within our SASS file we would use the + symbol followed by the mixin name. Note there are no arguments to pass into this function so we leave the opening And closing brackets empty.

```
+myFirstMixin()
```

To make our mixin dynamic we can add variables as our function arguments.

```
=myFirstMixin($radius, bg-color)
      border-radius: $radius
      background-color: $bf-color
```

We can now pass the arguments/variables whenever we call our mixin function within the opening and closing bracket ( ).

```
+myFirstMixin(10px, red)
```

View the SASS Mixins project folder as an example of this concept.

SCSS uses different syntax to create and call mixins. The above example is demonstrated below in the SCSS syntax format.

To create a mixin using SCSS syntax we would use the @mixin keyword followed by the function name and then opening and closing curly braces { }. All the functions styling should fall within the curly braces and each line should have a semi-colon ; at the end of each line — for example:

```
@mixin myFirstMixin($radius, bg-color) {
      border-radius: $radius;
      background-color: $bf-color;
}
```

To call this mixin using the SCSS syntax we would use the @include followed by the mixin name within the opening and closing braces { } — for example:

```
.box { @include myFirstMixin(10px, red); }
```

@Content Keyword:

The @content keyword is useful when creating mixins for media queries. It allows us to substitute our custom styles into a mixin wherever the @content keyword is present (i.e. the @content keyword is a placeholder item within your function).

An example of a media query:

```
=mixin mediaQuery($width)
      @media screen and (max-width: $width)
            @content
```

The @content allows us to pass in/substitute content whenever we call this function for a particular element that we would want to change based on the screen width (i.e. phone, tablet, pc). We no longer require multiple mixin for the media queries.

```
li
      float: left
      width: 100%/6
      +mediaQuery(600px)
            width: 100%
```

SASS Control Directives:

Sass control directives provide flow and logic for your mixin and functions. There are 4 control directives: @if, @for, @each & @while.

Example of @if syntax:

```
=mediaQuery($width...)
      @if length($width)==1
            @media screen and (max-width: nth($width,1))
                  @content
      @if length($width)==2
            @media screen and (max-width: nth($width,1)) and (min-
            width:nth($width,2))
                  @content
```

Note: by adding the … in an argument we are telling SASS that we do not know how many variables we are accepting within our arguments/parameters for our mixin; however, SASS should take all the variables and add it to a list.

The nth( ) function allows us to call a variable within the list by calling the index position number for the variable within the list. The first variable is index position 1, second is index 2, third is index 3 etc. (SASS does not use the zero indexing rule where 0 = first item).

To add a else clause we simply use the @else keyword after declaring the first @if statement.

Example of @for syntax:

There are two forms of the @for statement:

1. The first form is: @for $var from <start> through <end> which will start from the <start> and loop through each iteration and end at the <end>.

```
$class-slug: for !default
@for $i from 1 through 4
      .#{$class-slug}-#{$i}
            width: 60px + $i
```

This will produce the CSS result:

```
.for-1{
      width: 61px;
}
.for-2{
      width: 62px;
}
.for-3{
      width: 63px;
}
.for-4{
      width: 61px;
}
```

2. The second form is: @for $var from <start> to <end> which will start from the <start> and loop through each iteration to the <end> and stops looping. The loop will only stop once it reaches the <end>.

Example of @each syntax:

The @each syntax requires a list of variables. In SASS we can create a variable with list of items for example:

```
$authors: Adam, John, Dorothy
```

We can use the @each directive to loop through a list of items - form example:

```
=authorImages
      @each $author in $authors
            .photo-#{$author}
                  background: image-url("avatars/#{author}.png") no-repeat
.author-bio
      +author-images
```

This will produce the CSS result:

```
.author-bio .photo-adam{
        background: url('/images/avatars/adam.png') no-repeat;
}
.author-bio .photo-John{
        background: url('/images/avatars/John.png') no-repeat;
}
.author-bio .photo-Mary{
        background: url('/images/avatars/Mdam.png') no-repeat;
}
```

Example of @while syntax:

The @while syntax just like the above directives will repeatedly emit the nested block of styles until the statement evaluates to false.

```
=@eachLoop
        $types: 4
        $type-width: 20px
        @while $types > 0
                .while-#{ $types }
                        width: $type-width + $types
                $types: $types - 1
```

Each time this loops the types will reduce by 1. Eventually the @while statement will evaluate to false and this will end the loop style block. This will provide the following CSS results:

```
.while-4{
        width: 24px
}
.while-3{
        width: 23px
```

```
}
.while-2{
        width: 22px
}
.while-1{
        width: 21px
}
```

As you can see, SASS control directives are very powerful and can make your mixin more dynamic with the use of programming logic which was not possible with vanilla CSS. This can allow you to create a library of mixins/functions which you can use throughout the styling of your websites/web-apps.

Extend/Inheritance:

SASS extend feature allows you share sets of CSS properties from one selector to another. This helps you avoid having to write multiple class names for HTML elements. For example we are going to share the base message properties and extend to our error, warnings and successes messages.

SASS code:

```
.messages
  border: 1px solid #ccc
  padding: 10px
  color:  #333
.success
  @extend .message
  border-color: green
.error
  @extend .message
  border-color: red
.warning
  @extend .message
  border-color: yellow
```

This will create the CSS code:

```
.messages, .success, .error, .warning {
  border: 1px solid #ccc
  padding: 10px
  color:  #333
}
.success{
  border-color: green
}
.error{
  border-color: red
}
.warning{
  border-color: yellow
}
```

Operators:

SASS allows you to use math operators like +, -, *, / and % within your CSS. This is very helpful when you want to use maths within your mixins to do calculations for example dynamically sizing the width of components on your webpages.

Folder Structure (Best Practice):

Setting up a folder structure is important because not only does it allow you to maintain your SASS/CSS files but it also allows your website/web-app become modular and scalable as your project grows. Having a scalable workflow/folder structure at the beginning of any projects you embark will save you the hassle/headache of your website/web-app from getting out-of-control as your project grows.

SMACCS is a mindset/methodology you can incorporate within your workflow. You can view and download the book from https://smacss.com/

Using this methodology we can create a folder structure to keep our CSS files organised and prevent mixing our styles. You do not necessarily need to use all the folder structure found within SMACCS but you can use it as a reference. The purpose of SMACCS is to categorise your folder structure so that we have a pattern/rules for organising our style sheets.

You should create a folder structure that works best for you and your project. You should do some further reading of the SMACCS documentation to understand the methodology and then apply the methodology and create your own folder structure and file naming conventions that works best for you. This would help you keep organised and have a scalable CSS/SASS folder structure.

A Example Folder structure:

CSS

    app.css

SASS

    0-Plugins

        _plugins.sass

        _plugins-dir.sass

    1-Base

        _base.sass

        _base-dir.sass

    2-layouts

        _layouts.sass

        _layouts-dir.sass

    3-modules

        _modules.sass

        _modules-dir.sass

    _g-variable.sass

    _g-mixins.sass

    app.sass

This file structure allows you to separate your sass files into smaller manageable chunks. The directory “-dir.sass” files acts as a dependency file and will import all the individual sass files within the folder. The app.sass file will only reference the “-dir.sass” files. This allows your project to be scalable/modular as you add more sass files within the folder as you will only need to import the new sass file within the “-dir.sass” file only as the app.sass file will import everything within the directory file.

View the SASS Folder Structure project folder as an example of this concept.

SASS Frameworks:

There are a couple of SASS Frameworks that can be used within your SASS projects. These frameworks helps you use SASS to create websites/web-apps faster by using the mixin as plugins. Some popular SASS framework include:

- Bourbons (http://bourbon.io/)
- Neat (https://neat.bourbon.io/)
- Bitters (http://bitters.bourbon.io/)
- Refills (http://refills.bourbon.io/)
- Susy (http://oddbird.net/susy/)
- Compass (http://compass-style.org/)
- Empties

When you are more comfortable with CSS and SASS try using these frameworks and see how fast and easy it is to create projects using these frameworks within your projects.

Final Tip:

To prevent SASS from creating sourcemaps and cache files when the command is run in the terminal simply add to the end of the sass watch command --sourcemap=none and --no-cache — for example:

```
sass --watch SASS:CSS --sourcemap=none --no-cache
```

This is not recommended but the command is available should you wish not to use mapping files and have a cache file created.