**Sass**

## What is Sass?

* ****Sass**** stands for ****S****yntactically ****A****wesome ****S****tyle****s****heet
* Sass is an extension to CSS
* Sass is a CSS pre-processor
* Sass is completely compatible with all versions of CSS
* Sass reduces repetition of CSS and therefore saves time
* Sass was designed by Hampton Catlin and developed by Natalie Weizenbaum in 2006
* Sass is free to download and use

## Why Use Sass?

Stylesheets are getting larger, more complex, and harder to maintain. This is where a CSS pre-processor can help.

Sass lets you use features that do not exist in CSS, like variables, nested rules, mixins, imports, inheritance, built-in functions, and other stuff.

## A Simple Example why Sass is Useful

Let's say we have a website with three main colors:

#a2b9bc

#b2ad7f

#878f99

So, how many times do you need to type those HEX values? A LOT of times. And what about variations of the same colors?

## How Does Sass Work?

A browser does not understand Sass code. Therefore, you will need a Sass pre-processor to convert Sass code into standard CSS.

This process is called transpiling. So, you need to give a transpiler (some kind of program) some Sass code and then get some CSS code back.

## Sass File Type

Sass files has the ".scss" file extension.

## Sass Comments

Sass supports standard CSS comments /\* comment \*/, and in addition it supports inline comments // comment:

## Sass Variables

Variables are a way to store information that you can re-use later.

With Sass, you can store information in variables, like:

* strings
* numbers
* colors
* booleans
* lists
* nulls

Sass uses the $ symbol, followed by a name, to declare variables:

$*variablename*: value;

$myFont: Helvetica, sans-serif;  
$myColor: red;  
$myFontSize: 18px;  
$myWidth: 680px;  
  
body {  
  font-family: $myFont;  
  font-size: $myFontSize;  
  color: $myColor;  
}  
  
#container {  
  width: $myWidth;  
}

## Sass Variable Scope

Sass variables are only available at the level of nesting where they are defined.

Look at the following example:

$myColor: red;  
  
h1 {  
  $myColor: green;  
  color: $myColor;  
}  
  
p {  
  color: $myColor;  
}

CSS Output:

h1 {  
  color: green;  
}  
  
p {  
  color: red;  
}

Will the color of the text inside a <p> tag be red or green? It will be red!

The other definition, $myColor: green; is inside the <h1> rule, and will only be available there!

So, the CSS output will be:

## Using Sass !global

The default behavior for variable scope can be overridden by using the !global switch.

!global indicates that a variable is global, which means that it is accessible on all levels.

Look at the following example (same as above; but with !global added):

$myColor: red;  
  
h1 {  
  $myColor: green !global;  
  color: $myColor;  
}  
  
p {  
  color: $myColor;  
}

CSS Output:

h1 {  
  color: green;  
}  
  
p {  
  color: green;  
}

## Sass Nested Rules

Sass lets you nest CSS selectors in the same way as HTML.

Look at an example of some Sass code for a site's navigation:

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

## Sass Mixins

The @mixin directive lets you create CSS code that is to be reused throughout the website.

The @include directive is created to let you use (include) the mixin.

## Defining a Mixin

@mixin *name*{  
  property: value;  
  property: value;  
  ...  
}

@mixin important-text {  
  color: red;  
  font-size: 25px;  
  font-weight: bold;  
  border: 1px solid blue;  
}

## Using a Mixin

selector {  
  @include mixin-name;  
}

.danger {  
  @include important-text;  
  background-color: green;  
}

/\* Define mixin with two arguments \*/  
@mixin bordered($color, $width) {  
  border: $width solid $color;  
}  
  
.myArticle {  
  @include bordered(blue, 1px);  // Call mixin with two values  
}  
  
.myNotes {  
  @include bordered(red, 2px); // Call mixin with two values  
}

## Sass @extend Directive

The @extend directive lets you share a set of CSS properties from one selector to another.

The @extend directive is useful if you have almost identically styled elements that only differ in some small details.

The following Sass example first creates a basic style for buttons (this style will be used for most buttons). Then, we create one style for a "Report" button and one style for a "Submit" button. Both "Report" and "Submit" button inherit all the CSS properties from the .button-basic class, through the @extend directive. In addition, they have their own colors defined:

.button-basic  {  
  border: none;  
  padding: 15px 30px;  
  text-align: center;  
  font-size: 16px;  
  cursor: pointer;  
}  
  
.button-report  {  
  @extend .button-basic;  
  background-color: red;  
}  
  
.button-submit  {  
  @extend .button-basic;  
  background-color: green;  
  color: white;  
}

CSS Output:

.button-basic, .button-report, .button-submit {  
  border: none;  
  padding: 15px 30px;  
  text-align: center;  
  font-size: 16px;  
  cursor: pointer;  
}  
  
.button-report  {  
  background-color: red;  
}  
  
.button-submit  {  
  background-color: green;  
  color: white;  
}

## **Block Element Modifier**

* ****In a nutshell:****Use a standard naming convention for classes

BEM encourages developers to divide layouts into blocks and nested elements. Variations from the average appearance of a block or element should also be identified and applied using modifiers.

CSS declarations are applied using a single class name of format ****block-name**** for blocks and ****block-name\_\_element-name**** for elements, with two underscores in between. Modifier names are appended to classes, prefixed with an underscore or two hyphens for better clarity, for example ****block-name\_\_element-name\_modifer-name**** or ****block-name\_\_element-name--modifer-name****. An object is a block if it can exist without ancestors, otherwise it's an element.

Blocks can have nested blocks and elements, but elements cannot. Modifiers must be used alongside block and element classes, not instead of them.

BEM can be applied to a list, where ****list-block--inline**** and ****list-block\_\_item--active**** display lists horizontally and highlight items respectively:

<ul class="list-block list-block--inline">

<li class="list-block\_\_item">Item 1</li>

<li class="list-block\_\_item">Item 2</li>

</ul>

<ul class="list-block">

<li class="list-block\_\_item list-block\_\_item--active">Item 1</li>

<li class="list-block\_\_item">Item 2</li>

</ul>

BEM is a highly effective naming convention that creates predictably behaving CSS that is easy to manage, maintain and scale. BEM does have downsides, however, including the difficulty in inventing class names for deeply nested objects, the ridiculously long class names and bloated HTML that may sometimes result, and also the lack of consistency that is caused by the inability to share CSS between objects.

Atomic Css

Atomic CSS is a method of reducing the total amount of defined rules by creating a single rule for every declaration — enabling large style re-use.

Take a regular looking style rule:

.button {

border: none;

font-size: 14px;

background-color: purple;

border-radius: 3px;

}

<button className="button">Hello world</button>

Transformed to atomic CSS every declaration is now a rule:

.b-none {

border: none;

}

.fs-14 {

font-size: 14px;

}

.bgc-purple {

background-color: purple;

}

.br-3 {

border-radius: 3px;

}

<button className="b-none fs-14 bgc-purple br-3">Hello world</button>