Preprocessing

CSS on its own can be fun, but stylesheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Sass lets you use features that don't exist in CSS yet like variables, nesting, mixins, inheritance and other nifty goodies that make writing CSS fun again.

Once you start tinkering with Sass, it will take your preprocessed Sass file and save it as a normal CSS file that you can use in your web site.

The most direct way to make this happen is in your terminal. Once Sass is installed, you can run `sass input.scss output.css` from your terminal. You can watch either individual files or entire directories. In addition, you can watch folders or directories with the `--watch` flag. An example of running Sass while watching an entire directory is the following:

```
sass --watch app/sass:public/stylesheets
```

Variables

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse. Sass uses the `$` symbol to make something a variable. Here's an example:

SCSS SYNTAX Sass SCSS

```
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

When the Sass is processed, it takes the variables we define for the `$font-stack` and `$primary-color` and outputs normal CSS with our variable values placed in the CSS. This can be extremely powerful when working with brand colors and keeping them consistent throughout the site.

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

Nesting

When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't.

Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice.

With that in mind, here's an example of some typical styles for a site's navigation:

SCSS SYNTAX Sass SCSS

```scss
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

You'll notice that the `ul`, `li`, and `a` selectors are nested inside the `nav` selector. This is a great way to organize your CSS and make it more readable. When you generate the CSS you'll get something like this:

```css
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

Import

CSS has an import option that lets you split your CSS into smaller, more maintainable portions. The only drawback is that each time you use `@import` in CSS it creates another HTTP request. Sass builds on top of the current CSS `@import` but instead of requiring an HTTP request, Sass will take the file that you want to import and combine it with the file you're importing into so you can serve a single CSS file to the web browser.

Let's say you have a couple of Sass files, `_reset.scss` and `base.scss`. We want to import `_reset.scss` into `base.scss`.

SCSS SYNTAX Sass SCSS

```scss
// _reset.scss

html,
body,
ul,
ol {
   margin: 0;
  padding: 0;
}
```

```scss
// base.scss

@import 'reset';

body {
  font: 100% Helvetica, sans-serif;
  background-color: #efefef;
}
```

Notice we're using `@import 'reset';` in the `base.scss` file. When you import a file you don't need to include the file extension `.scss`. Sass is smart and will figure it out for you. When you generate the CSS you'll get:

```css
html, body, ul, ol {
  margin: 0;
  padding: 0;
}

body {
  font: 100% Helvetica, sans-serif;
  background-color: #efefef;
}
```

Partials

You can create partial Sass files that contain little snippets of CSS that you can include in other Sass files. This is a great way to modularize your CSS and help keep things easier to maintain. A partial is simply a Sass file named with a leading underscore. You might name it something like `_partial.scss`. The underscore lets Sass know that the file is only a partial file and that it should not be generated into a CSS file. Sass partials are used with the `@import` directive.

Mixins

Some things in CSS are a bit tedious to write, especially with CSS3 and the many vendor prefixes that exist. A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible. A good use of a mixin is for vendor prefixes. Here's an example for `border-radius`.

SCSS SYNTAX Sass SCSS

```scss
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}

.box { @include border-radius(10px); }
```

To create a mixin you use the `@mixin` directive and give it a name. We've named our mixin `border-radius`. We're also using the variable `$radius` inside the parentheses so we can pass in a radius of whatever we want. After you create your mixin, you can then use it as a CSS declaration starting with `@include` followed by the name of the mixin. When your CSS is generated it'll look like this:

```css
.box {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px;
}
```

Extend/Inheritance

This is one of the most useful features of Sass. Using `@extend` lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY. In our example we're going to create a simple series of messaging for errors, warnings and successes.

SCSS SYNTAX Sass SCSS

```scss
.message {
  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;
}
```

What the above code does is allow you to take the CSS properties in `.message` and apply them to `.success`, `.error`, & `.warning`. The magic happens with the generated CSS, and this helps you avoid having to write multiple class names on HTML elements. This is what it looks like:

```scss
.message, .success, .error, .warning {
  border: 1px solid #cccccc;
  padding: 10px;
  color: #333;
}

.success {
  border-color: green;
}

.error {
  border-color: red;
}

.warning {
  border-color: yellow;
}
```

Operators

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like `+`, `-`, `*`, `/`, and `%`. In our example we're going to do some simple math to calculate widths for an `aside` & `article`.

SCSS SYNTAX Sass SCSS

```
.container { width: 100%; }


article[role="main"] {
  float: left;
  width: 600px / 960px * 100%;
}

aside[role="complementary"] {
  float: right;
  width: 300px / 960px * 100%;
}
```

We've created a very simple fluid grid, based on 960px. Operations in Sass let us do something like take pixel values and convert them to percentages without much hassle. The generated CSS will look like:

```
.container {
  width: 100%;
}

article[role="main"] {
  float: left;
  width: 62.5%;
}

aside[role="complementary"] {
  float: right;
  width: 31.25%;
}
```