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 website.

The most direct way to make this happen is in your terminal. Once Sass is installed, you can compile your Sass to CSS using the `sass` command. You'll need to tell Sass which file to build from, and where to output CSS to. For example, running `sass input.scss output.css` from your terminal would take a single Sass file, `input.scss`, and compile that file to `output.css`.

You can also watch individual files or directories with the `--watch` flag. The watch flag tells Sass to watch your source files for changes, and re-compile CSS each time you save your Sass. If you wanted to watch (instead of manually build) your `input.scss` file, you'd just add the watch flag to your command, like so:

```
sass --watch input.scss output.css
```

You can watch and output to directories by using folder paths as your input and output, and separating them with a colon. In this example:

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

Sass would watch all files in the `app/sass` folder for changes, and compile CSS to the

`public/stylesheets` folder.

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:

```
$primary-color: #333;
```

```
body {

  color: $primary-color;

}
```

When the Sass is processed, it takes the variables we define for the $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.

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

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

  li { display: inline-block; }

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

Css format:

```
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.

// _reset.scss

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

// base.scss

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

Css output

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

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

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 transform.

Scss syntax

```
@mixin transform($property) {
  -webkit-transform: $property;
  -ms-transform: $property;
  transform: $property;
}

.box { @include transform(rotate(30deg)); }
```

Css output:

```
.box {
  -webkit-transform: rotate(30deg);
  -ms-transform: rotate(30deg);
  transform: rotate(30deg);
}
```

To create a mixin you use the @mixin directive and give it a name. We've named our mixin transform. We're also using the variable $property inside the parentheses so we can pass in a transform 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.

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 using another feature which goes hand in hand with extend, placeholder classes. A placeholder class is a special type of class that only prints when it is extended, and can help keep your compiled CSS neat and clean.

Scss syntax

```
/* This CSS will print because %message-shared is extended. */
%message-shared {
  border: 1px solid #ccc;
  padding: 10px;
  color: #333;
}

// This CSS won't print because %equal-heights is never extended.
%equal-heights {
  display: flex;
  flex-wrap: wrap;
}

.message {
  @extend %message-shared;
}

.success {
```

```
  @extend %message-shared;
  border-color: green;
}

.error {
  @extend %message-shared;
  border-color: red;
}

.warning {
  @extend %message-shared;
  border-color: yellow;
}
```

Css syntax

```
.message, .success, .error, .warning {
  border: 1px solid #ccc;
  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.

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

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

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

Css format

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

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

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

Conditional Execution - @if

As you'd expect, the Sass @if directive and its companions @else if and @else, allow you to include Sass code in the CSS output only if certain conditions are met. The basic syntax is simple:

Scss syntax

```
$test: 3;
```

```
p {
        @if $test < 5 {

                color: blue;
        }

}
```

Css format

```
p { color : blue; }
```

Nested if :

Scss syntax

```
$test: 3;
p {
    @if $test < 5 {
        color: blue;
        @if $test == 3 {
            text-color: white;
        }
    }
}
```

Css format

```
p {
    color : blue;
    text-color: white;
}
```

@else if directive

```
$test: 3;
p {
   @if $test > 3 {
       text-color: red;
   }
   @else if $test < 3 {
```

```
        text-color: blue;
    }
   @else {
        text-color: white;
   }
}
```

Css format

```
p {
    text-color: white;
}
```

@for directive

Scss

```
@for $i from 1 through 5 {
    .list-#{$i} {
        width: 2px * $i;
    }
}
```

Css

```
.list-1 {
    margin-left: 2px;
}

.list-2 {
   margin-left: 4px;
}

.list-3 {
   margin-left: 6px;
}

.list-4 {
   margin-left: 8px;
}

.list-5 {
```

```
    margin-left: 10px;
}
```

@each

Finally, the @each directive will execute a set of items in either a list or a map.

```
@each $s in (normal, bold, italic) {
    .#{$s} {
        font-weight: $s;
    }
}
```

Css format

```
.normal {
    font-weight: normal;
}

.bold {
    font-weight: bold;
}

.italic {
    font-weight: italic;
}
```