# CSS Pre-processors - Sass

Once you really start getting into styling and CSS, you may start noticing that sometimes your code is repetitive. For example, say you decide to make links and headings a certain red (#ff0000). Now, if you want to change that red, you'll have to go and change every instance of #ff0000 in your CSS. If you had the colour in a variable, it is easy to change colours. Simply change your colour variable value.

If you find yourself wishing that CSS supported variables (e.g. to hold colour values) or calculations or even that there was a way to write CSS a little more efficiently, you're not alone. That's where CSS pre-processors—such as Sass—come into play.

Note that in CSS3, primitive variables—called custom properties—are allowed but older browsers (if you need to support them) may not support this feature.

SCSS (Sassy CSS) is an *extension* of CSS, so it looks very similar to CSS. In fact, all valid CSS is also valid SCSS. The only differences are the presence of variables, the ability to do calculations, nesting, and mixins (more on what this is later). We'll be looking at Sass (actually, SCSS) because this is more commonly found nowadays. (There is used to be another pre-processor—LESS—but this is not really used anymore. Bootstrap was originally built using LESS, but is now built using Sass.)

To get started quickly with Sass without the hassle of multiple installations, I suggest trying a GUI compiler to make your life easier. Over time, as you use more tools, you may prefer command-line (I usually use command-line), but command-line requires multiple installations, so to start or to try, I usually suggest a GUI compiler.

A GUI compiler you can try for Sass is Scout App (https://scout-app.io/).

## *Sass (http://sass-lang.com/)*

Sass uses ***.scss*** files (in older versions, you may find ***.sass*** files) so instead of ***.css*** files, you will now have ***.scss*** files. These ***.scss*** files will need to be compiled into ***.css*** files. When you style, you will use the ***.scss*** file(s) but in your HTML, you still point to the compiled ***.css*** files (browsers do not recognized ***.scss*** files).

### *Compiling via GUI*

Note that GUI compilers are at a disadvantage in that **you're usually limited to the versions of Sass supported**. For command-line, you can use the latest versions because you're not dependent on someone else integrating a specific version.

You can use a program such as Scout (https://scout-app.io/) to compile your Sass. Scout is a self-contained Ruby environment which runs Sass and Compass. Again, it usually preferable once you use a lot of Sass to use command-line, but to quickly get you started, try it out with a GUI. (I don't want you to spend too much time on installations and troubleshooting installs but rather just start using Sass.)

### *Compiling via command-line*

In order to compile Sass, you will need Dart Sass (https://sass-lang.com/dart-sass). You can download the standalone executable and add sass to your PATH (i.e. unzip the the Dart Sass ZIP and place the files somewhere, then modify your system PATH environment variable to include the path to the Dart Sass folder) or you can install using `npm` (you'll need Node.js for `npm`—get the LTS version).

You can then compile using the following command:

```
sass <relative-path-to-scss-file> <relative-path-to-compiled-css-file>
```

For example, if your ***styles.scss*** file is in a folder named ***scss*** in your project, and your compiled CSS should be in ***css/styles.css***, then navigate to your project in your command-line terminal and compile with:

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

To watch and auto-compile, navigate to your project folder in the terminal or cmd and type:

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

where ***scss*** is the folder holding my ***.scss*** Sass files and ***css*** is the folder where my CSS will be compiled to. `--watch` will watch for saved changes to your ***.scss*** files and auto-compile to CSS when it detects changes.

### *Variables*

Sass variables are prefixed with `$`. For example, to declare a variable named "red":

```
$red: #ff0000;
```

You can then use this variable later in your Sass styles:

```
#header {
  background-color: $red;
  color: #fff;
}
a {
  color: $red;
}
```

You can also create a variable to hold a font stack, like so:

```
$font: Arial, Helvetica, sans-serif;
```

Note that variable values are constant. They can only be defined once.

### *Nesting*

Nesting allows you to group styles more logically. This is nice because it allows you to group related styles, making them easier to find and understand. Take a look at the following example.

```
a {
  color: red;

  &:hover {
    color: green;
  }
}
```

This is equivalent to (CSS):

```
a {
  color: red;
}
a:hover {
  color: green;
}
```

The `&` is a special symbol denoting the parent selector. In the example, because `&` is nested *within* `a`, it denotes `a` (i.e. it's a self-selector).

You can now keep menu styles neat by:

```
#main-menu {
  /* some styles */

  li {
    /* some styles */
  }
  a {
    /* some styles */
  }
}
```

This is the equivalent of:

```
#main-menu {
  /* some styles */
}
#main-menu li {
  /* some styles */
}
#main-menu a {
  /* some styles */
}
```

Note that you can continue nesting, but it is not recommended that you go past 3 levels deep. If you recall, CSS parsing can get slower when you chain too many elements in a selector. Nesting is the equivalent of chaining (it is compiled as a chain).

### *Calculations*

If you want to calculate a number, you can use the following operators:

- +
- -
- *
- /
- % (modulus)

In the above list, modulus returns the remainder of division. For example, **4 % 2 = 0** and **15 % 7 = 1**.

### *Extending/inheritance*

Sass allows you to reuse style blocks by *extending* styles. This allows you to share styles amongst different selectors. Take a look at the following example:

```
.box {
  width: 100px;
  height: 100px;
  background-color: yellow;
}
.red {
  @extend .box;
  background-color: red;
}
```

The above example is sharing the box height and width style with the `.red` selector. The `.red` selector then overwrites the `background-color`.

### *Mixins*

Mixins are kind of like functions. You can have parameters to allow for some style customizations. This is handy for similar styles but with minor tweaks (e.g. different colour headings). You must first use the keyword `@mixin` to declare a mixin followed by the name of your mixin.
For example, to create a transition mixin to write out all the vendor-prefixed versions of the transition property followed by the standard transition:

```
@mixin transition($transition) {
  -webkit-transition: $transition;
  -moz-transition: $transition;
  -o-transition: $transition;
  transition: $transition;
}
```

In the above example, there is a parameter named $transition (denoted like a variable). Within the mixin, you're just using the parameter's value as the transition values. (Note that a mixin which does the same thing already exists in Compass.)

To use the mixin, you will need to **`@include`** it like so:

```
#square-spin {
  width: 100px;
  height: 100px;
  background-color: $red;
  @include transition(all 2s);

  &:hover {
    transform: rotate(1080deg);
  }
}
```

This will compile to:

```
#square-spin {
  width: 100px;
  height: 100px;
  background-color: #ff0000;
  -webkit-transition: all 2s;
  -moz-transition: all 2s;
  -o-transition: all 2s;
  transition: all 2s;
}
#square-spin:hover {
  transform: rotate(1080deg);
}
```

You can specify multiple parameters with default values as well. Having default values allows you to `@include` a mixin without specifying parameter values. You only need to change them if needed.

```
@mixin color-heading($bg:none, $color:#f00) {
  background-color: $bg;
  color: $color;
}
```

There's a lot more to Sass but this is just a start. If you're interested, take a look at the documentation. Of interest, are loops such as:

- @each (https://sass-lang.com/documentation/at-rules/control/each)
- @for (https://sass-lang.com/documentation/at-rules/control/for)
- @while (https://sass-lang.com/documentation/at-rules/control/while)

You can use loops to create auto-generated styles (this is how frameworks generate their sizing CSS classes for layout containers).

**Try on your own time (if you're interested in command-line tools)**

Take a look at Sass and ZURB Foundation. See if you can get the ZURB Foundation installed using npm (npm is included if you install Node.js) and a new ZURB Foundation Sass project created. (You'll need the command line.)

See http://foundation.zurb.com/sites/docs/sass.html for details.