Sass (Synactically Awesome Style Sheets)

Table of Contents

Introduction to SASS

A web browser can only understand CSS (Cascading Style Sheets) as the styling technique for any DOM (Document Object Model) element being rendered. CSS, as a language has its own feature set and it's fairly powerful, but at times does not provide enough functionality to create a clean and reusable chunk of rules.

For example you are not able to reuse a collection of rules in multiple selectors and the unavailability of variables in CSS leads to ambiguous pieces of data being scattered across the stylesheet. In addition, CSS stylesheets are getting larger, more complex, and harder to maintain.

To overcome some of these limitations, the concept of CSS preprocessors offers an advanced way of writing CSS in a more programmantic type syntax, which extends the basic functionalities of CSS. This code is then compiled and the output is normal CSS code that any browser can understand.

There are a multitude of CSS processors (and their respective compilers) available, but Sass is arguably the leader in this space (Less is another very popular preprocessor), but for the Guidewire ProducerEngage product, we need to use Sass and specifically the **node-sass** implementation (more on this later).

Sass lets you use features that don't exist in CSS yet like variables, nesting and looping, functions, mixins (a special type of function), inheritance and more. Sass will take your preprocessed file and save it as a normal CSS file that you can use in your website.

Brief History of Sass

- In 2006, Sass was designed and developed as a feature of the HAML markup language and written in Ruby
- In 2012, the first **libsass** library was released, which is a C/C++ port of Sass, designed to be dropped into other projects and wrapped. The intent being to provide Sass capabilities in other tools and languages. Being native C/C++, it is much faster than Ruby Sass.
- Around the time of the libsass release, wrappers are written for it in a number of languages and tools, including C, Crystal, Go, Java, JavaScript, Lua, .NET, Perl, PHP, Python, Ruby, Scala and **Node**. Node has proven very popular and the **node-sass** binding has been taken into the main Sass Github repository.
- Development of Ruby Sass as the primary implementation of Sass continued until April 2018.
- **Dart Sass** was introduced in April 2018 as the new primary implementation of Sass and Ruby Sass is deprecated.
- Currently in a sunset maintenance only period, end of life for Ruby Sass is scheduled for March 2019.
- Dart Sass will now receive new features before any other implementation.
- The libsass implementation (as used by **node-sass**) will continue as before, implementing new features as they are introduced to Dart Sass.

Sass Implementation and Version

The binding implementation of Sass used by ProducerEngage is **node-sass** (first released in 2012 as a node wrapper around libsass), so we will be installing that implementation.

The Grunt file in ProducerEngage indicates that the version number should be higher than 4.5.2 and I believe it actually uses version 4.6.1. However, that particular version has a major bug that prevents us from using the `--watch` or `-w` flag when compiling to CSS from VSCode or a command line, so so we will install version 4.9.2 as I know that one works correctly.

The vanilla JavaScript implementation of Sass that is used when installing via npm (node package manager) has some performance limitations when compared against the new Dart Sass, but it's still much faster than Ruby Sass.

Installing Sass with Node Package Manager

Ensure you have Node.js 8.9.4 installed globally, using command line. e.g.

```
C:\>node --version
v8.9.4
```

If you don't have it, you can get it from https://nodejs.org/en/download/

Ensure you have node package manager (npm) 5.6.0 installed globally, using command line. e.g.

```
C:\>npm --version
5.6.0
```

npm is distributed with Node.js, so if you installed Node, you should automatically get npm installed with it.

Install node-sass v4.9.2 globally with npm from the command line. e.g.

```
npm install -g node-sass@4.9.2
```

Test the installation from the command line. e.g.

```
C:\>node-sass -v
node-sass       4.9.2   (Wrapper)       [JavaScript]
libsass         3.5.4   (Sass Compiler) [C/C++]
```

That's it! There are no external dependencies and nothing else you need to install to use Sass

You can also get help on usage with command:

```
C:\>node-sass --help
```

Visual Studio Code Extensions for Sass

Since we are using Microsoft Visual Studio Code for development, we can improve the experience of working with Sass and the node-sass binding by installing the following VSCode extensions.

Live Server 5.1.1 Extension

See https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

This is needed as pre-requisite for Live Sass Compiler extension and used for live previews of Sass changes without having to re-load pages. This launches a development local Server with live reload feature for static and dynamic pages.

There are a number of ways to start and stop the live server, but if you have an HTML file, this extension adds a context menu allowing you to launch the page on the test server (default port is 5500 e.g. `http://127.0.0.1:5500/index.html`, but this can be changed).

To stop the server, the easiest way is by pressing the **Port:5500** button on the status bar at the bottom of the screen.

Live Sass compiler 3.0.0 Extension

See https://marketplace.visualstudio.com/items?itemName=ritwickdey.live-sass

This extension will watch for Sass changes (specific files or folders) and automatically compile CSS on the fly. Used in conjunction with Live Server we don't have to manually compile Sass to CSS using the command line and we don't need to refresh our browser page to see the effect of those changes (it automatically updates to reflect the new CSS).

This extension will work using defaults out of the box, but we can make a quick change to settings to improve it.

Go to File --> Preferences --> Settings and add the following:

```
// Set your exported CSS Styles, Formats & Save location.
"liveSassCompile.settings.formats": [
  {
    "format": "expanded",
    "extensionName": ".css",
    "savePath": "~/../css/"
  }
]
```

The `format` default is `expanded`, so leave that as is (you could change the format to `compact`, `compressed` or `nested`).

The `extensionName` default is `.css`, so leave that as is (you could change this to `.min.css`) if you wanted to minify the CSS (which you would do before deploying to production).

The `savePath` default needs to be changed. As written above, this instructs the extension to make the output folder for compiled CSS to be `css` and relative to the input folder. Effectively we can name our source folder anything we want (e.g. `sass-source`) and when Live Sass Compiler finds any changes to a sass file it will compile output to the `css` folder under the same parent as the `sass-source` folder.

You could also add the following to your user settings:

```
"liveSassCompile.settings.generateMap": false
```

The default is `true`, but you may want to disable sourceMap production during compilation by changing it to `false`. CSS sourcemaps allow the browser to map CSS generated by a pre-processor, such as Sass, back to the original source file, including exactly which Sass mixin, placeholder or variable is responsible for a given line of CSS. This is useful for debugging Sass with Chrome.

Sass 1.4.9 Extension

See https://marketplace.visualstudio.com/items?itemName=robinbentley.sass-indented

This is an optional extension and we may not need it, but Visual Studio Code only has out of the box support for the newer Sassy CSS `.scss` syntax. This extension was written to provide syntax highlighting, auto complete and snippets for Sass files written with the original indented Sass syntax - more on this below).

New Sassy CSS vs Original Sass Syntax

Before we go any further, we will cover the two versions of Sass syntax and the two file extensions used by Sass for these two different flavors of syntax. We may come across both types on the web, so we should know what they are and how they differ at a high level, but generally after this section, we will use .scss (Sassy CSS files) because that's what ProducerEngage uses.

First I will demonstrate a plain CSS file, then how Sass is written for both syntaxes (and file extensions).

- `.scss` is the newer file extension for what is known as Sassy CSS syntax
- `.sass` is the original file extension for original indented Ruby Sass syntax

CSS Code

Here is some simple CSS:

```
header {
    margin: 0;
    padding: 0;
    color: #fff;
}
```

Newer Sassy CSS Syntax - Extension Format .scss

To convert the above CSS into Sassy CSS `.scss` extension format, we will use a variable `$color` and give it a hexadecimal color value of `#fff` for white color. And then under the CSS style, instead of putting a hexadecimal color value of `#fff`, use the variable `$color` that was set in the beginning of the code.

```
$color:  #fff;
header {
    margin: 0;
    padding:0;
    color: $color;
}
```

Syntax highlighting and formatting for Sassy CSS is supported out of the box by Visual Studio Code.

Original Sass Indented Syntax - Extension Format .sass

Indented Syntax is the original way of writing Sass and is similar to HAML. It uses indentation to separate code blocks and newline characters to separate rules.

For our original syntax `.sass` extension, we will have the same variable and value just like the `.scss` extension format, but, this time, without semi-colons and brackets. This is the old format in writing Sass and although more concise, it's further removed from the original CSS than the newer Sassy CSS form.

```
$color: #fff
header
    margin: 0
    padding: 0
    color: $color
```

Syntax highlighting and formatting for indented Sass is not supported out of the box by Visual Studio Code, which is why we installed the Sass extension.

Example Usage

From this point on, I will assume Sassy CSS `.scss` files are being used, but the same applies to `.sass` files.

Manual Compile Command

With node-sass installed, you can use the command line interface to compile Sass files with the command below:

```
node-sass sass -o css
```

This manually compiles all Sass files in the `sass` folder and outputs the compiled CSS files to the `css` folder. You can open up the web page that uses this CSS in a browser and see the effect on the rendered HTML.

Automated Compile Command

You might not want to execute this command every time you make a change to your `.scss` file, as it can become cumbersome. In that case, you can keep a watch on the `.scss` files, and compile to CSS automatically, whenever there is a change to it.

```
node-sass -w sass -o css
```

When you make a change to the Sass files, you should see something like this in the console:

```
=> changed: C:\sass_tutorial\sass\main.scss
Rendering Complete, saving .css file...
Wrote CSS to C:\sass_tutorial\css\main.css
```

You can stop watching using `CTRL-C`

```
Terminate batch job (Y/N)? y
```

Automated Compile Extension

To make things even easier, we can leverage the Live Sass Compiler tool we installed earlier. From within VSCode you can click the **Watch Sass** button in the status bar at the bottom of the screen.

When you do this, the Live Sass Compiler extension will look for Sass files, compile them and put the CSS in the output folder specified by the savePath setting. It will also change the status to **Watching...**

You should see output in the console that looks like this:

```
Compiling Sass/Scss Files:
c:\sass_tutorial\sass\main.scss
--------------------
Generated :
c:\sass_tutorial\css\main.css
--------------------
Watching...
--------------------
```

Now every time you change a Sass file and save it, the CSS will be automatically re-generated. If you also have the Live Server started, you don't even have to refresh your page, the changes are immediately rendered.

When a change is detected, you should see something like:

```
Change Detected...
main.scss
--------------------
Generated :
c:\sass_tutorial\css\main.css
--------------------
Watching...
--------------------
```

To stop watching, just press the **Watching...** button in the status bar at the bottom of the screen.

Useful Reference URLs

Here are some useful URLs for reference:

1. Github node-sass repository is the official home of the node-sass implementation
2. Official Learn Sass Documentation is a high level basic introcution to Sass
3. Official Sass Reference Documentation contains every detail you will ever need to know about Sass
4. Tutorialspoint - Learn Sass is another decent tutorial on sass
5. Sass on Wikipedia contains history and background as well as some useful examples of Sass
6. 1st Web Designer - Learn Sass Tutorial is based on the Ruby implementation but works fine with node-sass

Import and Modularize

The first thing to do when starting to write CSS is to finalize the strategy in modularizing the CSS. For example you can decide if you are going to create separate stylesheets for each basic element of the UI, like typography, forms, iconography, buttons, layout, reset etc.

Though it is recommended that you use multiple files for the sake of maintainability, it is always the developer's call, depending on the ability to smartly group the rules in a single file and performance considerations.

The main drawback for the CSS `@import` is that each time you use it 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 we have two Sass files, `_reset.scss` and `base.scss` and we want to import `_reset.scss` into `base.scss`.

Sass to be imported - reset.scss

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

Sass doing the importing - base.scss

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

File `base.scss` imports `reset.scss`, but note that we do not need the file extension, it is inferred by Sass.

When we generate the CSS we will get a single CSS file:

```
html, body, ul, ol {
  margin: 0;
  padding: 0;
}
body {
  font: 100% Helvetica, sans-serif;
  background-color: #efefef;
}
```

Sass Partials

Sass files that contain little snippets of CSS that you can include in other Sass files are called partials. They allow you to modularize your CSS and 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. A partial requires that it be imported into another file that will be processed into CSS in order for it to be output.

In the following example, you will see a simple Sass architecture that illustrates this principal. Notice how `application.sass` is the only file that does not contain an underscore _ in the name as this will be the file that is output to CSS.

```
stylesheets/
|-- application.sass    // Sass manifest file
|
|-- _reset.sass         // Partials
|-- _variables.scss             |
|-- _functions.scss             |
|-- _mixins.scss                |
|-- _base.sass                  |
|-- _layout.sass                |
|-- _module.sass                |
|-- _state.sass                 |
|-- _theme.sass         // Partials
```

Variables

Variables are one feature of Sass that adds a lot of value to your code, in terms of maintainability. It helps in cases where you are required to make application wide changes in a single go.

Basic Syntax

Here's the key points you need to know when defining variables:

1) Variables consist of a key (name) and a value pair.

2) The name of the variable always starts with a `$` sign.

3) The variable name and its value are separated by a colon.

4) The semicolon ends the statement (in `.scss` files only, `.sass` files do not need them).

Here's an example of how you might declare a variable using Sass to define a primary color for your site:

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

Here `$primary-color` is the variable name and `#ccc`, is the color value that is stored in it.

Variable Types

All variables in Sass are assigned to a specific data type. They seven types are:

1) number (e.g. 42, 56px)

2) string (e.g. "Hello World!")

3) color (e.g. goldenrod, rgb(1, 45, 34), #BADA55)

4) bool (true or false)

5) list (e.g. (a, b, c), a, ,b, c)

6) map (e.g. (a: 1, b: 2))

7) null (null)

Typical Usage

It's important to identify exactly what needs to be converted to variables, for a maintainable Sass file. We can declare all the color values, images and font-family in variables, put all those variables in a single `.scss` file, and then include it in the main `.scss` file.

In the example below we define some variables, then use those variables in the CSS body selector:

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

When this 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 it.

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

This can be extremely powerful when working with brand colors and keeping them consistent throughout the site and will bring much more control over the theming of the UI, allowing us to quickly change the color values, iconography and other parameters which define the general look and feel of the UI.

Default Values

The `!default` flag is used as a mechanism to assign a variable some default value if it has not yet been assigned a value:

```
$padding: 10px;
$padding: 20px !default;
.foo {
  padding: $padding; // 10px
}
```

The `!default` flag is typically used in 3rd party libraries where sensible defaults are set, but which are designed to allow the consumer of the library to set their own custom values.

Bear in mind that variables with null value are treated as unassigned by default, which means that a variable assignment with `!default` will override a variable assignment to null.

```
$padding: null;
$padding: 20px !default;
.foo {
  padding: $padding; // 20px
}
```

Scope

Variables can be defined anywhere and just like other programming languages they have scope depending on where you place them. Sass handles scopes the way you would expect it to and utilizes shadowing. A variable defined inside a mixin or function is local to that code and a global and local variable could share the same name seamlessly (i.e. the local code has access to the local variable and everywhere else has access to the global variable and the two don't interfere with each other).

The `!global` flag allows you to override a global variable from a local scope. Here's an example:

```
$padding: 10px;
.module {
  $padding: 20px !global;
  padding: $padding;
}
.foo {
  padding: $padding;
}
```

The resulting CSS would now be:

```
.module {
  padding: 20px;
}
.foo {
  padding: 20px;
}
```

Interpolation

Like several other programming languages, Sass allows use of interpolation. So instead of having to use concatenation like this:

```
$name: 'Hurricane Florence';
.foo {
  content: 'Hello ' + $name + '!'; // Hello Hurricane Florence!
}
```

We can write this concise code instead:

```
$name: 'Hurricane Florence';
.foo {
  content: 'Hello #{$name}!';
}
```

Wrapping a variable identifier with #{} tells Sass to treat the content of the variable as plain CSS and one of the most common uses of interpolation is to use it within the `calc(..)` function.

```
.main {
  $sidebar-width: 300px;
  width: calc(100% - #{$sidebar-width}); // calc(100% - 300px)
}
```

Quirks

The one quirk you need to know about in variables is underscore `_` and dash `-` which are treated as the same character. So you could define a variable as `$primary-color` and retrieve it successfully using `$primary_color`. This was intentional on behalf of the designers as they perceived dash and underscore to be a stylistic preference, not a syntactical difference.

Control Directives and Expressions

This section covers how to control styling based on conditions or applying the same style many times with variations. They are mainly used in mixins (covered later) and provide considerable flexibility.

The `if` Condition

The built in `if` condition returns one result from two possible outcomes. The syntax is

```
if(expression, value1, value2)
```

So in this contrived Sass example:

```
h2 {
  color: if(1 + 1 == 2, green, red);
}
```

the CSS produced would be:

```
h2 {
  color: green;
}
```

The `@if` Directive

The `@if` directive accepts Sass expressions and uses the nested styles whenever the result of the expression is not false or null. The syntax is:

```
@if expression {  //CSS codes are written here }
```

So in this example:

```
p {
  @if 10 + 10 == 20   { border: 1px dotted;   }
  @if 7 < 2     { border: 2px solid;  }
  @if null    { border: 3px double; }
}
```

the CSS produced would be:

```
p {
  border: 1px dotted;
}
```

The `@else` Directive

The `@else` directive is used in conjunction with the `@if` directive to build more complex conditions. The syntax is

```
@if expression {
  // CSS codes
} @else if condition {
  // CSS codes
} @else {
  // CSS codes
}
```

So in this example

```
$carMake: audi;
p {
  @if $carMake == mercedes {
    color: red;
  } @else if $carMake == ford {
    color: blue;
  } @else if $carMake == audi {
    color: green;
  } @else {
    color: black;
  }
}
```

the CSS produced would be

```
p {
  color: green;
}
```

The `for` Directive

The `@for` directive allows you to generate styles in a loop with a counter variable used to set the output for each iteration. There are two flavors (`through` and `to`) discussed below..

The `through` keyword

The `through` keyword specifies the range including the values for `<start>` and `<end>`. The syntax is shown below where `<start>` and `<end>` are Sass expressions that return integers:

```
@for $var from <start> through <end>
```

If the `<start>` is greater than the `<end>` then the counter variable is decremented otherwise it is incremented.

So in this example

```
@for $i from 1 through 4 {
  .p#{$i} { padding-left : $i * 10px; }
}
```

we would start at 1 and increment through 2, 3 and 4 and produce the follwing output CSS:

```
.p1 {
  padding-left: 10px;
}
.p2 {
  padding-left: 20px;
}
.p3 {
  padding-left: 30px;
}
.p4 {
  padding-left: 40px;
}
```

The `to` keyword

The `to` keyword specifies the range including the values for `<start>` to the value **before** `<end>`. The syntax is

```
@for $var from <start> to <end>
```

If the `<start>` is greater than the `<end>` then the counter variable is decremented otherwise it is incremented.

So in this example

```
@for $i from 1 to 4 {
  .p#{$i} { padding-left : $i * 10px; }
}
```

we would start at 1 and increment through 2 and 3 and produce the follwing output CSS:

```
.p1 {
  padding-left: 10px;
}
.p2 {
  padding-left: 20px;
}
.p3 {
  padding-left: 30px;
}
```

The `@each` Directive

The `@each` directive allows us to deal with each item in a list or map. The syntax is

```
@each $var in <list or map>
```

So this example

```
@each $color in red, green, yellow, blue {
  .p_#{$color} {
     background-color: #{$color};
  }
}
```

The output CSS will be

```
.p_red {
  background-color: red;
}
.p_green {
  background-color: green;
}
.p_yellow {
  background-color: yellow;
}
.p_blue {
  background-color: blue;
}
```

Multiple Assignments

Multiple values can also be used with the `@each` directive using the following syntax

```
@each $var1, $var2, $var3 ... in <list>
```

In this example

```
@each $color, $border in (aqua, dotted), (red, solid), (green, double){
  .#{$color} {
    background-color : $color;
    border: $border;
  }
}
```

the following CSS will be produced

```
.aqua {
  background-color: aqua;
  border: dotted;
}
.red {
  background-color: red;
  border: solid;
}
.green {
  background-color: green;
  border: double;
}
```

Multiple Assignments with Maps

Similar to the prior example but this time using maps, which are considered as lists of pairs (key and value). The syntax for this variation of `@each` is

```
@each $var1, $var2 in <map>
```

In this example

```
@each $header, $color in (h1: red, h2: green, h3: blue) {
  #{$header} {
    color: $color;
  }
}
```

The output CSS will be

```
h1 {
  color: red;
}
h2 {
  color: green;
}
h3 {
  color: blue;
}
```

The `@while` Directive

the `@while` directive takes Sass expressions until the statement evaluates to false, it iteratively outputs nested styles. The key thing to note is that counter variable needs to be incremented/decremented on each iteration. The syntax is

```
while(condition) {
  // CSS code
}
```

The following example

```
$i: 50;
@while $i > 0 {
  .paddding-#{$i} { padding-left: 1px * $i; }
  $i: $i - 10;
}
```

would produce this CSS

```
.paddding-50 {
  padding-left: 50px;
}
.paddding-40 {
  padding-left: 40px;
}
.paddding-30 {
  padding-left: 30px;
}
.paddding-20 {
  padding-left: 20px;
}
.paddding-10 {
  padding-left: 10px;
}
```

Nesting

Sass allows you to define nested selectors that mimic the structure of HTML. This allows you to use shortcuts to create your CSS. For example, using nesting we could write:

```
div {
  p {
    color: black;
  }
}
```

This will compile to:

```
div p { color: black; }
```

We could have also given the div its own properties like this:

```
div {
  font-size: 14px;
  p {
    color: black;
  }
}
```

This compiles to two separate CSS rules:

```
div { font-size: 14px; }
div p { color: black; }
```

To nest styles, just enclose the selector (or selectors) inside the curly braces of another selector:

```
.parent {
  .child {
  }
}
```

Nesting can extend as many levels deep as you want, so you can nest elements inside of an element that is in turn nested inside another element:

```
.first-level {
  .second-level {
    .third-level {
      .fourth-level {
      }
    }
  }
}
```

It is generally considered bad practice to nest more than three levels deep as deeply nested structures become hard to read and maintain and may have unintended consequences. Having the ability to nest however, allows us to write CSS faster.

Ampersand Selector

You can use the `&` to build compound selectors. You do this by following the ampersand with a suffix. For example:

```
.col {
  &-span1 { width: 8.33%; }
  &-span2 { width: 16.66%; }
  &-span3 { width: 24.99%; }
}
```

Which will give us:

```
.col-span1 { width: 8.33%; }
.col-span2 { width: 16.66%; }
.col-span3 { width: 24.99%; }
```

The parent selector is placed where the ampersand is.

Nested Properties

Sass provides a shorthand way to write styles using CSS namespaces. Normally when setting properties in the same namespace, for example border, we have to write out individual properties in our style sheet. With Sass we can write the namespace once and nest its properties.

```
.example {
  border: {
    style: dashed;
    width: 30px;
    color: blue;
  }
}
```

This compiles to the following with the namespace is appended to the properties:

```
.example {
  border-style: dashed;
  border-width: 30px;
  border-color: blue;
}
```

Functions

Much like other programming languages, in Sass a function is a chunk of code that possibly accepts arguments and returns a value. It's used to extract a repeatable pattern into a reuseable piece of code.

The `@function` directive

In Sass, a function definition starts with `@function` directive, then the name of the function, then a pair of parentheses, possibly but not necessarily containing parameters. Here's a simple example:

```
// The `get-base-url()` function has no parameter
@function get-base-url() {
  @return '/assets/';
}
// Usage
.module {
  logo-image: url(get-base-url() + 'kemper-logo.png');
}
```

A function can be defined almost anywhere in a document, not just at root level and when it's defined in a rule set, the function is scoped to that rule set, the same way variables are local when defined within a specific block. The concept of global shadowing applies to functions as well in the same way as it works for variables. The value to be returned is denoted by `@return`.

Function Parameters

These are defined in the function signature as variables, separated with commas and are demonstrated in this fairly useless divide function.

```
@function divide($dividend, $divisor) {
  @return ($dividend / $divisor);
}
```

Parameters can have default values (these are called optional parameters). Following on from the example above, we will define the dividend as mandatory and the divisor as optional:

```
@function divide($dividend, $divisor : 2) {
  @return ($dividend / $divisor);
}
```

In this case, if the optional divisor is not provided, then the default value of 2 is used.

Note that optional parameters must always come after mandatory ones in the parameter list, so this would not work:

```
@function divide($dividend: 100, $divisor) {
  @return ($dividend / $divisor);
}
// Throws this error:
// Required argument $divisor must come before any optional arguments.
```

When calling a Sass function, you can either pass the arguments in the order defined, or pass named arguments (in whatever order you desire). Here's some examples of calling the function above that has the mandatory dividend and optional divisor:

```
$element-width: 1800;
.bar {
    width: divide($element-width, 3); // arguments in defined order, both passed to
function
}
.dar {
    width: divide($element-width); // relying on the default divisor, only passing
dividend to function
}
.far {
    width: divide($divisor: 4, $dividend: $element-width);  // using keyword arguments
to reverse the order
}
```

Functions can be defined to have an unlimited number of parameters if required. To do so, add an ellipsis arg-list `(...)` to the last part of the signature like this:

```
// The intention of `map-deep-get` is to help getting values deeply nested in maps
// The first parameter is the map to browse and any parameter after that are keys nested
within it
@function map-deep-get($map, $keys...) {
  @each $key in $keys {
    $map: map-get($map, $key);
  }
  @return $map;
}
```

An arg-list is a distinct datatype in Sass, although its behavior is identical to a list and you handle them in the same way as a list. You therefore have a choice to use either, but using an arg-list clearly indicates that we know there could be an unknown number of arguments passed.

Function Usage

Usually functions are used as CSS values and they can be used anywhere a variable can. So in selectors, media queries, properties, values, inside variables, functions and mixins. Like variables they might also need to be interpolated to work correctly if they are used in unconventional places. e.g.

```
// Example function declaration
@function my-function() {
  @return 'foo';
}
// Calling the function by itself does not work and throws an error:
// > `Invalid CSS after " my-function()": expected "{", was ";"`
.foo {
  my-function();
}
// Calling the function in place of a property works as long as it is properly
interpolated.
.foo {
  #{my-function()}: 'bar';
}
// Calling the function in place of a selector works as long as it is properly
interpolated.
.foo, #{my-function()} {
  content: 'bar';
}
```

```
// Calling the function inside a variable value works perfectly.
$foo: my-function();
// Calling the function in place of a media query value works perfectly.
@media (min-width: my-function()) { .. }
// Calling the function in place of a feature query value works perfectly.
@supports (content: my-function()) { .. }
```

Native Functions

Sass provides a lot of out of the box functions to make writing styles easier. For example, we can manipulate colors using functions like `lighten(...)` and `darken(...)`. For example:

```
.message-info {
  $color: blue;
  color: $color;
  background-color: lighten($color, 20%);
}
```

There are pages of functions which can be found in the listed in the Sass script functions documentation

Mixins

Mixins are reusable sets of styles which can be used throughout the stylesheet. Correct usage of mixins can reduce the usage of non-semantic stray classes like `border-red`, `no-padding` etc. The difference between a mixin and a function is that a mixin is a function that can output code rather than return a result like a regular function.

A function is a good way to abstract a repeated operation based on parameters, but a mixin is a good way to abstract repeated style patterns with the ability to adapt the output based on parameters.

The `@mixin` Directive

The `@mixin` directive is used to define a mixin function. Here is an example of a simple mixin with no parameters (although parameters could be passed just like a function):

```
@mixin notification {
   padding:10px;
   border-radius:5px;
   font-size:1em;
 }
.error{
   @include notification;
   background:red;
   color:white;
 }
```

Resulting CSS code:

```
.error{
   padding:10px;
   border-radius:5px;
   font-size:1em;
   background:red;
   color:white;
 }
```

The first section of code, declares a mixin named `notification`, which has a set of 3 rules defined inside. In the second section of code, the selector `.error` reuses the mixin, along-with its own rules. `@include` includes the mixin rules within a selector's rule sets.

This next example shows usage of parameters for height and width. If the height is omitted, it will default to be the same as the width:

```
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
.foo {
  @include size(100%, 42px);
}
.bar {
  @include size(100px);
}
```

When compiled, Sass will produce the following CSS:

```
.foo {
  width: 100%;
  height: 42px;
}
.bar {
  width: 100px;
  height: 100px;
}
```

Parameters in mixins behave exactly the same as in functions.

The `@content` Directive

The `@content` directive allows authors to pass a block of styles to their mixins. This is useful if you want to define abstractions relating to the construction of selectors and directives. for example:

```
@mixin on-event {
  &:hover,
  &:active,
  &:focus {
    @content;
  }
}
.foo {
  color: blue;
  @include on-event {
    color: red;
  }
}
```

In the code above we want foo to be blue, except when hovered, active or selected when we want it to be red. the CSS produced looks like this:

```
.foo {
  color: blue;
}
.foo:hover, .foo:active, .foo:focus {
  color: red;
}
```

Extension/Inheritance

The Problem That Extension Solves

One of the most useful but also potentially damaging features if misused is the Sass `@extend` directive. It's intent is to prevent repetition. To understand extension and inheritance in Sass, consider the following CSS which gives us a gray box for a message while adding color for info and warning variations:

```
.message {
  background-color: gray;
  border: 1px solid black;
  margin: 1em;
}
.info {
  background-color: blue;
  border: 1px solid black;
  margin: 1em;
}
.warning {
  background-color: red;
  border: 1px solid black;
  margin: 1em;
}
```

There's a lot of repetition and shows no relationships that exist between the message types. The repetition itself could be solved a number of ways, but most ways lack a way to show the the relationship between classes. The only way to show the relations in CSS is through selector grouping:

```
.message, .info, .warning {
  background-color: gray;
  border: 1px solid black;
  margin: 1em;
}
.info {
  background-color: blue;
}
.warning {
  background-color: red;
}
```

This code eliminates the repetition and shows some sense of grouping in the initial block, but if the subsequent `.info` and `.warning` blocks become separated from the initial block it is very difficult to know that they are related and if our relationships ever span multiple Sass partials we are out of luck.

The `@extend` directive provides a shortcut for this kind of selector grouping while making the relationships explicit. The following Sass will output the same CSS as before:

```
.message {
  background-color: gray;
  border: 1px solid black;
  margin: 1em;
}
.info {
  @extend .message;
  background-color: blue;
}
.warning {
  @extend .message;
  background-color: red;
}
```

The `@extend` feature in Sass aims to provide a way for a selector A to extend the styles from a selector B.

When doing so, the selector A will be added to selector B so they both share the same declarations.

Remember that `@extend` should be used only in those places where you want to share properties between the elements.

Problems with Extending

There are a number of issues with extension that have lead to some developers to reject `@extend` completely. The skepticism is justified and I am not going to expand too much here since it would take too long, but the main complaints are:

1) The `@extend` messes with the cascade, changing the specificity of styles in ways that are not obvious or easy to control. While mixins inject code where you call them, `@extend` injects code somewhere else entirely, adding selectors to the original extended code block, which might be in a different file or even hidden inside a third-party package. While that may not be a big issue for a utility such as clearfix, it can cause major issues when used recklessly. So If there is any chance that specificity will be an issue, `@extend` should be avoided and `@mixin` used instead.

2) Because a selector can appear multiple times in a stylesheet, extended selectors are replaced everywhere they appear. You might be altering code you haven’t considered. Many teams avoid this problem by only extending placeholder selectors and only defining placeholders in one location. That’s helps to make extensions clear and controllable.