Five Simple Steps

A Pocket Guide

Better CSS with Sass

by Cole Henley

Better CSS with Sass
by Cole Henley

Published in 2015 by Five Simple Steps
119 St Mary Street
Cardiff
CF10 1DY
United Kingdom

On the web: www.fivesimplesteps.com
and: cole007.net
Please send errors to errata@fivesimplesteps.com

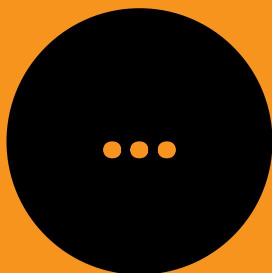
Publisher: Five Simple Steps
Copy Editor: Ary Lacerda
Technical Editor: Stu Robson
Production Manager: Amie Lockwood
Art Director: Craig Lockwood
Designer: Valentino Cellupica

Copyright © 2015 Cole Henley
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording or any information storage and retrieval system, without prior permission in writing from the publisher.

ISBN: 978-3-863730-81-9

A catalogue record of this book is available from the British Library.

Proudly made in Wales.



Introduction

In a relatively short period of time, Sass has become an essential part of the front-end *developer toolbox*. It takes a lot of the legwork out of writing good CSS and can help foster good practices on writing flexible, maintainable, robust and efficient code.

Personally, Sass is the first resource I go for when creating a new website. It has gone from a tool to speed up the way I write my style sheets to a means of writing increasingly complex and clever CSS. There are new blog posts and *code snippets posted daily* demonstrating the things that can be accomplished with Sass that were barely conceivable two or three years ago. However, with this speed in evolution, the barrier to entry has risen fast. Where do we start if we are new to Sass? How do we distinguish our mixins from our functions? How do we use variables? What is a partial? Although it is a relatively simple language to learn, Sass brings with it a wide range of terms that can be intimidating to somebody only familiar with HTML and CSS.

This pocket guide is intended to introduce you to Sass and will explain some of the confusing terms that may have held you back from learning this brilliant tool. It will provide an overview of how Sass can dramatically improve your workflow and help make your CSS work easier for you. Although this guide is primarily aimed at those new to Sass, hopefully there will be something for everybody within these pages.

1

Getting started with Sass

What is Sass

Sass is a pre-processor. What this means is that it takes code in one format and outputs it in another format; in this case, CSS. At its simplest, this can mean taking your normal CSS and minifying it to reduce the file size. However, as we will cover in this pocket guide, there are numerous benefits to using a pre-processor like Sass.

The first benefit is that it separates our working code from our production code. This is particularly good practice when it comes to working in teams and using *version control*. Secondly, we can use the functions and methods built into Sass to make how we write our CSS more efficient. Thirdly, we can use principles like nesting and partials to better organise our code. Finally, we can extend Sass with a range of third-party mixins, functions, libraries, and frameworks.

Sass vs Scss

Sass isn't the only CSS pre-processor. You may also have heard of *Less*. Originally, Less was by far more popular than Sass because Less was far easier to pick up, using a markup similar to CSS. However, over time Sass has become the pre-processor of choice amongst the discerning front-end developers, particularly with the introduction of its sibling syntax, Scss.

Sass itself uses a different form of notation from CSS. In an effort to speed up development, it employs its own shorthand, stripping out brackets and instead using tabs. For example, I might

define some colour rules for an alert box with the following, using tabs to denote the difference between the selector I'm defining (`.alert-box`) and the properties (`colour` and `border`) I'm affecting:

```
.alert-box
    color: #dd0000;
    border: 1px solid #dd0000;
```

If we're coming from CSS, this approach is quite difficult to pick up. Scss was introduced to ease the migration to a pre-processor by letting us extend the simple CSS we use every day. The above code example in Scss would simply be:

```
.alert-box {
    color: #dd0000;
    border: 1px solid #dd0000; }
```

Look familiar? Even without using a pre-processor, we can understand the above. Scss makes the transition to using a pre-processor much easier by working in a way we already know. Furthermore, it reduces the barrier to begin working with Sass because we can just drop our existing CSS into our Sass files and these will be processed just fine.

We distinguish between Sass and Scss by the file extension we use. Sass files are saved as .sass whilst Scss files are saved as .scss. Although Sass and Scss are ultimately two distinct syntaxes for delivering the same thing — using the same language, functions,

and principles — for ease of learning, I will focus on Scss. For the rest of this guide, I will use Sass to refer to the language and techniques involved in working with this pre-processor and .scss to refer to the files we are working with.

Installing Sass

You can run Sass in one of two ways: by installing an application on your computer or by using the command line. I will touch on some applications that can install Sass for you later but for this pocket guide, we will be installing and working with Sass through the command line. I've learned from bitter experience it is almost always better to work on the command line rather than relying on an application where often our understanding of what is happening is sacrificed for ease of use.

When we use a pre-processor like Sass, the processing can happen in one of two ways: by converting files as needed or by pointing Sass to a particular file or folder in order to watch for changes within your files. When a change is made, the processing kicks in and CSS is outputted in the desired format.

So what are we waiting for? Let's roll our sleeves up and get our hands dirty.

Mac: Ruby and Sass

Installing Sass on a Mac is relatively straightforward. It operates as a *gem* – that is, a package that runs using the Ruby programming language. Fortunately for Mac users, Ruby comes already installed, so all we need to do is install the Sass gem. We can do this by typing into the command line:

```
gem install sass
```

Depending on your permissions level, you may need to use the sudo command. This basically tells the command line that you are the root administrator or super user and that you really, really want to install Sass. You will likely be prompted to enter your administrator password. We should always exercise caution when using sudo but in this instance we should be fine:

```
sudo gem install sass
```

We can check to see if Sass is installed correctly by typing the following:

```
sass -v
```

The `-v` tells the command line that we are requesting the version of Sass that is available. As this is being written, the latest version of Sass is 3.4.15, so the above returns:

```
Sass 3.4.15 (Selective Steve)
```

Selective Steve is the release name of the latest (3.4) Sass version. If at any time we want to update our version of Sass, we type:

```
sudo gem update sass
```

This will replace our version of Sass with the latest available.

Windows

Windows users will first need to install Ruby, which can easily be achieved by using installer software. Check out the installer at *rubyinstaller.org* to get Ruby up and running. Once you have Ruby installed, follow the instructions for Mac Users above to then get Sass up and running.

Other tools

As well as the command line, there are a number of tools that will help compile Sass for you. *Mixture* is a great tool that can perform a range of actions to speed up your workflow, from compiling Sass to image compression and Javascript minification. For something simpler, *Scout* is a good app available for both Mac and Windows for handling Sass processing.

The above tools can be great if you are scared of the command line. However, learning from the command line gives you a greater chance to learn how things work under the bonnet and have a clearer understanding of what happens when things go wrong.

Running Sass on the command line

At its most basic, the way Sass works is by converting a file from one format to another. This is what we mean by a pre-processor. An example would be useful to illustrate this. Create a folder you want to work in and save a file called style.scss with some simple CSS, for example:

```
body {
    font-family: sans-serif; }
```

This is a relatively straightforward bit of CSS to change the default font of our website. Our browser won't understand .scss files so we must first convert this into CSS. To do this we use the command line to navigate to the folder our file is situated in:

```
cd /Applications/MAMP/htdocs/styles
```

All cd is doing is pointing the command line to a specific location in the file system we want to work in. We can then use the sass command to convert a .scss file in that folder into a .css file for use in our production code:

```
sass style.scss style.css
```

Et voila. If you look at your folder, it will now contain an extra file: style.css containing your outputted (production) CSS. Depending on the version of Sass you are using and your configuration, you may also have another file: sass.css.map but I will cover this in more detail later in this guide.

The above is quite straightforward but we don't want to be writing this out every time we want to convert our files to CSS, so we can use Sass to watch a file for any code changes. When a .scss file is updated, it will process the necessary changes automatically and produce the CSS we want to output.

This can be achieved by using the watch command:

```
sass --watch style.scss:style.css
```

There are two things to note here. Firstly, we are extending Sass with the watch command (note the two hyphen characters) and secondly, when using watch, we use a colon (`:`) instead of a space between the input and output files. This will now monitor the source file for any changes and process them as and when it needs to. This will continue until we terminate the command, which we can do with Ctrl + c.

If the .scss and .css files we are working with are in the same folder, we can even dispense with specifying the output file:

```
sass --watch style.scss
```

In the fourth chapter, I will talk about ways we can break our working Sass code into different files. In this context, we would want to watch out for changes to more than one file, so we want to apply the watch command to a folder instead. We can do this with the following:

```
sass --watch /styles
```

This says watch out for changes to files that are in the `/styles/scss` folder and output the results to the corresponding CSS files in the `/styles` folder. The above example are all using relative paths based on where we are in the file system. We can also use absolute paths to do the same thing:

```
sass --watch /styles/scss:/styles/
```

So we've got Sass installed, we know how to convert Sass (or .scss) files into CSS, and we can use our existing CSS, which will process just fine into Sass. But where's the fun in that? In the next chapter, I will start to look at some of the things we can do in Sass that we can't do in CSS.

```
sass --watch
    /Applications/MAMP/htdocs/styles/scss:/Applications/
    MAMP/htdocs/styles
```

So we've got Sass installed, we know how to convert Sass (or .scss) files into CSS, and we can use our existing CSS, which will process just fine into Sass. But where's the fun in that? In the next chapter, I will start to look at some of the things we can do in Sass that we can't do in CSS.

2

The building blocks of Sass

When we first look at Sass, we come across a number of terms that might be new to us, such as variables, scope, nesting, extends, mixins, and functions. These are all new to CSS but they ultimately serve the same purpose: to extend the functionality of CSS by borrowing from some of the principles of programming languages you may already be familiar with, such as PHP and JavaScript. This chapter will look at these and see how, once we have Sass compiling our Sass files into CSS, we can then make the most of what Sass has to offer.

Variables

How many times when writing CSS have you had to make repetitive changes across your files? For example, when a font-size changes or a client wants to modify a brand colour. In the past, we would have to do a 'find and replace' to change all these things within our style sheets. Variables mean we can make this information available globally by defining it once and then referring to this original definition elsewhere in our Sass files. So, for example, if we know we are going to be using red across our site, we might want to define this as:

```
$red: #dd0000;
```

Here we are defining a variable – `red` – using the dollar symbol (`$`) as a prefix to say we want to treat this as a variable. We then use a

colon (`:`), as we would with CSS, to define the value for our variable. Finally, we terminate our rule with a semi-colon (`;`). To reference that variable, taking our example from the previous chapter, we would use:

```
.alert-box {
    color: $red;
    border: 1px solid $red; }
```

This means that if, at some later stage, we want to update the tone of red, we can just update the variable's value and this change will be reflected in every instance that `$red` is used in our Sass.

```
$red: #bd250a;

// produces
.alert-box {
    color: #bd250a;
    border: 1px solid #bd250a; }
```

With this simple example, you can immediately see the value of variables. Any information that is repeated across your CSS file is best served by variables. The important thing to remember is that, like CSS, Sass is read in sequence, so you will need to define your variables before you start using them.

Colours, fonts used, and breakpoints defined in media queries all serve as great examples of when you might want to use a variable.

Nesting

The next important thing to know about Sass is that when we define our CSS, we can nest our arguments. In normal CSS, when we want to define the CSS of a child element, we would do the following:

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

In Sass, we can simplify this by nesting our queries to demonstrate their context. By nesting, we use curly brackets to wrap around selector rules. To define a rule for child elements, we just place these rules within the parent brackets:

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

Here the `li` selector is nested within the brackets of the `ul.nav` selector. In normal CSS, all our arguments are tied to the context of the specific selector chain we are using. In Sass, however, our arguments inherit context, so the above will output exactly the same as in our original declaration:

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

So with Sass, we can not only save the amount of selectors we have to type, but also make the CSS we write much more modular, or in other words, tied to the particular context we are styling. You can nest as deep as you want, but beware of the dangers of specificity as this will cause havoc once you end up with selectors *three or more levels deep*.

Sass accepts all the selectors we would normally use in CSS. However, with nesting, Sass introduces a new selector: the parent selector denoted by the ampersand character (`&`). The parent selector serves as a way of accessing a selector that has already been defined in our nesting. So, for example if we want a global list style which has padding but then want to define a class that has smaller padding, we can do the following:

```
ul {
    padding: 20px;
    &.slim {
        padding: 10px; }}
```

This will produce the following CSS:

```
ul {
    padding: 20px; }
ul.slim {
    padding: 10px; }
```

So & is repeating the immediate parent within our nesting sequence. And we can chain this infinitum, so the following:

```
ul {
    padding: 20px;
    &.slim {
        padding: 10px;
        &.slimmer {
            padding: 5px; }}}
```

Would ultimately produce:

```
ul.slim.slimmer {
    padding: 5px; }
```

You should exercise caution with this approach as by abusing nesting, we can end up with a nasty case of *classitis* and a tangled mess of selectors!

A really good example of where you might want to use the parent selector is with *pseudo-classes*. For example, with links, we

might want to define hover and focus states:

```
a {
    text-decoration: none;
    &:hover, &:focus {
        text-decoration: underline; }}
```

This will output as:

```
a {
    text-decoration: none; }
a:hover, a:focus {
    text-decoration: underline; }
```

Another use of parent selectors is if we want to increase the specificity of a particular nested selector to override a default setting. For example:

```
p {
    font-size: 1em;
    .article & {
        font-size: 1.2em; }}
```

This would apply a different font-size to paragraph elements contained within an .article, as this would create a more *specific rule*. Additionally, we can use the parent selector to set rules for a specific context. For example, if we wanted to style a certain element based

on its location within a site and we used different classes on the body element for each section:

```
.article h1 {
    background-color: #FFAFA5;
    .home & {
        background-color: #BBFCA2; }
        .about {
            background-color: #A0ADF1; }}

// produces
.article h1 {
    background-color: #FFAFA5; }
.home .article h1 {
    background-color: #BBFCA2; }
.article h1 .about {
    background-color: #A0ADF1; }
```

Since Sass version 3.3, we can also use parent selectors to create compound selectors. That is, we can define new classes by appending our selector to a parent class name. Previously, if we have two kinds of footers on a site that feature different font sizes, we might want to declare this in Sass as follows:

```
.footer {
    font-size: 16px;
    &.footer-small {
        font-size: 12px; }}
```

However, we can simplify this immensely by using the & to create a compound selector as follows:

```
.footer {
    font-size: 16px;
    &-small {
        font-size: 12px; }}
```

The output is similar but the code we are using to generate that output is much simpler, more concise, and ultimately less specific:

```
footer {
    font-size: 16px; }
.footer-small {
    font-size: 12px; }
```

This approach really comes to the fore when you start to look at *Object Oriented CSS* and approaches like *SMACSS* and *BEM*. I tend to use BEM in my projects, which is based on the principle of extending our class names to reflect the place of an element within a particular context. BEM stands for Block Element Modifier and involves defining our CSS classes by a block (the parent object), an element (a descendent object) and a modifier (variations to that object). Let us say that we have an article and want to provide styles for an image in that article but also an alternative styling if we want that image to be smaller. Using double underscore `__` as a separator for elements and double hyphen `--` as a separator for modifiers, we might write

the following:

```
.article {
    margin: 20px 0;
    &__image {
        width: 40%; }
    &--small {
        width: 20%; }}
```

The output for this is:

```
.article {
    margin: 20px 0; }
.article__image {
    width: 40%; }
.article__image--small {
    width: 20%; }
```

Thus, we can immediately see how we can use Sass to simplify the way we achieve increasing modularity within our CSS rules and class names. And in all honesty, such modular approaches would not have been possible without the simplicity that tools like Sass afford us.

Nesting CSS properties

As well as nesting our CSS selectors, we can also use namespaces to nest our CSS properties. Namespaces are a common term in programming to signify a group of variables or properties that share a particular functionality. Namespaces in CSS tend to be reserved for those properties that can be specified as shorthand – e.g. `font` – or broken down into specific sub-properties – e.g. `font-family`, `font-size`, etc. Other examples of namespaced properties in CSS include `background`, `list`, `margin`, `padding` and `border`. To nest our properties, we declare our root property and then nest any child properties within curly brackets, as we would for nested selectors:

```
.body {
    background: {
        color: #A1FF7F;
        image: url(tile.png);
        repeat: no-repeat;
        position: center; }}
```

This compiles to:

```
.body {
    background-color: #A1FF7F;
    background-image: url(tile.png);
    background-repeat: no-repeat;
    background-position: center; }
```

Extends

As we have seen, we can use the parent selector (`&`) to avoid repetition. However, this is very much tied to the scope of the particular parent. What happens if we want to reuse some Sass code later in our project? We have variables, but these are only really for reusing small values such as a pixel size or colour value; they cannot be used for reproducing large chunks of code. This is where extends come to the fore.

We can use extend – through the `@extend` command – to reuse already declared code snippets. For example, we might want to apply the same stylistic approach to both form elements and quote elements:

```
.form {
    border: 1px solid #219fe3;
    background: #7cc6ef; }
.quote {
    border: 1px solid #219fe3;
    background: #7cc6ef;
    color: #000; }
```

There is a lot of repetition here. We could instead use a common class between these elements to avoid repetition, but we can also use extends to repeat the same rules across classes. Taking the above example, we can instead write:

```
.quote {
    @extend .form;
    color: #000; }
```

Which will output the following:

```
.form, .quote {
    border: 1px solid #219fe3;
    background: #7cc6ef; }
.quote {
    color: #000; }
```

This makes our Sass much more reusable, but there can be dangers of our Sass code spilling out into other elements with this approach. To resolve this problem, we can use something called *placeholder selectors*.

Placeholder selectors

Extends can be incredibly useful for creating resuable blocks of CSS. However, in the above example, if we end up writing lots of styles specific to our `.form` element these will then be applied to our `.quote` elements which is not very helpful. This is what the placeholder selector is designed to resolve.

The point of a placeholder selector is that we are creating a set of rules that we never intend to be outputted into CSS except for

later reference as an extend within other elements. We define our placeholder selectors through the use of a percent symbol (`%`) to denote that this selector will only ever be used as a reusable piece of code:

```
%list-style {
    list-style: none;
    padding: 0; }
.list-group {
    @extend %list-style;
    margin: 2.5em 0 0.75em; }
```

CSS will not understand `%` so this part of the code is never outputted by Sass into our CSS:

```
.list-group {
    list-style: none;
    padding: 0; }
.list-group {
    margin: 2.5em 0 0.75em; }
```

This is much more effective and avoids the risks of over-extending our Sass.

So in this chapter, we've seen the ways in which Sass can be used to avoid repetition in our CSS. We can use variables to take control of values we want to reuse throughout our styles. We can nest our Sass to define context and through the parent selector, we

can keep our Sass files lean and efficient. Through extends, we can start to create reusable, modular styles and with the placeholder selector, we can begin to think about avoiding code bloat. In the next chapter, we will look at how some of the principles of programming can be applied to Sass to make our code more logical.

3

Logical Sass

We spent the last chapter talking about the ways Sass can make our code more reusable. A lot of these techniques, Sass borrows from the tried and tested principles of programming. In this chapter, we will look at how Sass takes this further by letting us apply logic and helping us make even more reusable CSS though mixins and functions.

Simple Maths

One of the nice features of Sass is that we can start to perform simple maths functions. Addition (`+`), multiplication (`*`), division (`/`) and subtraction (`-`) are all ways we can control and modify numbers within our Sass. We just need to make sure that we contain any mathematic operations within brackets, e.g.:

```
.heading {
    font-size: (24 / 2) + px; }
```

Will output as:

```
.heading {
    font-size: 12px; }
```

Note the use of the plus character after our calculation. Here, we are not using the `+` character to carry out an addition, but saying that we want to append the `px` unit to our calculation. This simple maths

may not seem to be very useful but the power of mathematics can be demonstrated when we start to look at mixins.

Mixins

We touched on extends and placeholder selectors as a way of creating reusable chunks of code. But what if we want to then modify that code? The code produced by extends and variables are fixed; we cannot change them once they have been defined, only override certain rules by redefining them. Mixins provide a way of creating reusable chunks of code but offer the ability to modify parts of that code as and when we need them. We define the mixin, any variables we want it to access (called arguments), and then our logic – the stuff we want our mixin to output – is contained within curly brackets:

```
$mixin mixin-name($variable) {
    output stuff here; }
```

A good, practical example where we might want to use a mixin is when defining em or rem font-sizes for a website. Say your designer has given you the pixel values for your fonts but working these out in relation to each other can quickly become very confusing. So a mixin can help work this out for us:

```
$base-font-size: 16;
@mixin font-size($size) {
    font-size: $size + px;
    font-size: ($size / $base-font-size) + rem; }
```

Here, we are defining `$size` as an argument and then outputting two lines of CSS: one calculating a rem font based on a global `font-size` (defined by our `$base-font-size` variable) and a `px` fallback for older browsers. Again, we are using the plus character (`+`) to concatenate – join together – the calculated values and the units we need. To access our mixin, we then use the `@include` method. This tells Sass we are treating what follows as a mixin:

```
.box {
    @include font-size(24); }
```

The resulting CSS for this is:

```
.box {
    font-size: 24px;
    font-size: 1.5rem; }
```

We can extend our mixin further to factor in `line-height`. Say we want to output a `line-height` as a proportion of our `font-size` (rather than using a specific unit). We can do the following, dividing our `$line-height` by our `$size` variable:

```
@mixin font-size($size, $line-height) {
    font-size: $size + px;
    font-size: ($size / $base-font-size) + rem;
    line-height: ($line-height / $size); }
```

So calling this mixin, we now give it two arguments:

```
.box {
    @include font-size(24, 32); }
```

Which outputs as:

```
.box {
    font-size: 24px;
    font-size: 1.5rem;
    line-height: 1.33333; }
```

If we want, we can set default arguments for our mixin. So, for example, if the `line-height` is relatively constant in our designs, we don't need somebody to enter it repeatedly. So if we know our `font-size` is 16px and our `line-height` is normally 32px, we could set a default `line-height` using the colon separator when defining our arguments:

```
@mixin font-size($size, $line-height: 32) {
…
.box {
    @include font-size (24); }
```

Say we want our text to sit along a baseline *rhythm*. We can start to use some simple maths in our arguments to define a consistent `line-height`. So if our base `font-size` is 16px and our base line height is 24px, we can specify a default line height of 24 divided by whatever value is entered for `$size`:

```
@mixin font-size($size, $line-height: (24/ $size)) {
```

So with very little effort, we can create really useful, reusable chunks of CSS. We don't necessarily even need to have custom variables in our mixins. One good example of this might be a mixin for when you are using floats intensively in your CSS layouts. We simply write a clearfix mixin to use every time we want to clear an element:

```
@mixin clearfix {
    &:before,
    &:after {
        display: table;
        content: '';
        line-height: 0; }
    &:after {
        clear: both; }}
```

Here, we are using the parent selector we covered in the last chapter and the `:before` and `:after` pseudo-classes in conjunction with the content property to insert content into the DOM and clear whatever precedes it. Note that without any arguments for our mixin, we can declare it and call it without needing to employ rounded brackets:

```
form {
    @include clearfix; }
```

This produces the following CSS:

```
form:before, form:after {
    display: table;
    content: "";
    line-height: 0; }
form:after {
    clear: both; }
```

Functions

A function is like a mixin but instead of returning code blocks, it can only be used to return values. As a nonsensical function, we could perform some simple maths to produce a value squared:

```
@function squared($number) {
    @return ($number * $number); }
```

The `@return` command is specifying the value or string we want to return from our function. To access this function, we would just do the following:

```
.text {
    padding: squared(10) + px; }
```

Which will return:

```
.text {
    width: 100px; }
```

One of the great things with Sass is that we can apply mathematics to variables featuring unit values. For example, consider the following double function:

```
@function double($number) {
    @return ($number + $number); }

.text {
    width: double(10px); }
```

Will output:

```
.text {
    width: 20px; }
```

As with mixins, we can declare as few or as many arguments as we want for our functions and also set default values:

```
@function line-height($font-size, $line-height: 1.5) {
    @return ($font-size * $line-height); }

.text {
    line-height: line-height (18px); }
```

Which returns:

```
.text {
    line-height: 27px; }
```

However, the real power of mixins and functions comes when we start to use conditionals within them to determine their output.

Conditionals

Conditionals are simple means for determining an output depending on an input: e.g. if X is Y then do Z. Conditionals are the building blocks of any programming language and are particularly useful when we write functions and mixins. Conditionals give us the chance to limit what we return to our CSS, avoid repeating ourselves, and ultimately make our Sass more extensible.

@if and @else conditions

A simple example of conditional logic might be a mixin to define what fonts we want to use in our site. Say we are using web fonts. We might want to specify particular fonts rather than rely on the browser to apply weight and style. In this instance, we use `@if` and `@else` to say that we'd like to test a set of conditions:

```
@mixin font-type($font: 'base') {
    @if ($font == bold) {
        font-family:'Avenir-Demi'; }
    @else if ($font == italic) {
        font-family:'Avenir-LightItal'; }
    @else {
        font-family:'Avenir-Light'; }}
```

In this example, we are wrapping our statement in brackets. In Sass, we don't need to do this. We could equally write:

```
@if $font == bold {
    font-family:'Avenir-Demi'; }
```

However, it is good practice when working with programming languages to enclose your conditional statements. We use the double equal (`==`) to test our variable against certain conditions; in this instance, the `font-weight` or style we want to apply. To access the mixin, we would then use the following:

```
.heading {
    @include font-type(bold); }
```

And the resulting output is:

```
.heading {
    font-family: 'Avenir-Demi'; }
```

We can also use conditions directly in our Sass. For example:

```
$weather: sunny;
p {
    @if ($weather == overcast) {
        color: grey; }
    @else if ($weather == sunny) {
        color: yellow; }
    @else {
        color: blue; } }
```

Which would return:

```
p {
    color: yellow; }
```

In the above examples, we are testing strings using the equals operator (`==`). We also have a range of other operators available to us:

The full range of operators are:

Operator	Tests against
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

@for loops and lists

Another conditional we can use is `@for`. The `@for` conditional loops through a sequence until a set of conditions are met. Most often, a for loop is used to go through a numeric sequence. For example, with a simple grid system, we might want to set the width for a series of columns on the fly. The following saves us from having to write out rules for 10 columns:

```
@for $i from 1 through 10 {
    .col-#{$i} { width: (10% * $i); }}
```

This outputs as:

```
.col1-1 {
    width: 10%; }
.col1-2 {
    width: 20%; }
…
.col1-10 {
    width: 100%; }
```

Here, we are setting an initial value for `$i` (1), then saying we want to increment it one at a time until we reach a defined limit (10). This is called a counter and we can access this when writing out our HTML classes. Note the use of the hash character (`#`) in our loop followed by curly brackets. This is to denote that we want to access the `$i` as a string in our selector. We can then use the integer `$i` and some simple multiplication within our rules to produce our widths: 10 × 1 = 10, 10 × 2 = 20, etc.

As well as looping through a numeric sequence, we can also loop through a range of string values using Sass lists. A list is a way of grouping variables together. Say we want to introduce different colour palettes for each section of our site. We can declare these in our Sass as variables:

```
$home: #F7E900;
$about: #FF5F09;
$news: #A0005E;
$links: #41004B;
```

We can then put the variables into a list as follows:

```
$pages: $home, $about, $news, $links;
```

Having defined our list we can access this in a `@for` loop like so:

```
@for $i from 1 through length($pages) {
    body.section-#{$i} {
        background: nth($pages, $i); }}
```

We use the built-in Sass function `length()` to find out how many items there are in our list and use this to loop through our list items, again using a counter, until we reach this value. We use the hash character to bring our integer into our selector and then access one of a range of built-in Sass functions – `nth()` – to retrieve the value in our list. This basically pulls out the defined value for the $pages item at each stage of our loop.

So to recap: we use `length()` to find out how many items are in our list, and `nth()` to find out the value at a particular position in our list. If you have worked with languages like PHP or JavaScript before, you will see that lists are similar to arrays.

The end result of our above Sass is a set of selectors based on our pre-defined values:

```
body.section-1 {
    background: #F7E900; }
body.section-2 {
    background: #FF5F09; }
…
```

In general, `@for` loops are most useful when using numbers. When we are working with strings and lists it is much more useful to use `@each`.

@each loops and Sass maps

Taking the `$pages` list and values defined in our last example, we can use `@each` to loop through our values for our CSS:

```
@each $item in $pages {
    body.section-#{ index($pages, $item)} {
        background: $item; }}
```

`@each` lets us loop through the individual values within our list. However, rather than using `$i` to increment through each instance, we use `$item` to loop through the values in our `$pages` list, which we access as `$item`. The `index()` method used here is the reverse of `nth()` used above. We are using this to get the position of the value that we are returning from our each loop in our list. However,

`.section-1` is not a very useful class. We can use Sass maps to make our CSS classes more meaningful. A Sass map is where we declare our list as a set of `key:value` pairs. For example:

```
$pages: (
    home: #F7E900,
    about: #FF5F09,
    news: #A0005E,
    links: #41004B);
```

We can then use the `@each` conditional to access these `key:value` pairs:

```
@each $key, $value in $pages {
    body.section-#{ $key} {
        background: $value; }}
```

Here, we are taking the `$key` to define our CSS selector and the `$value` to define our CSS value. This outputs the much more useful and semantic:

```
body.section-home {
    background: #F7E900; }
body.section-about {
    background: #FF5F09; }
…
```

So, with a few simple built in methods, we can loop through lists and variables to avoid having to write out lots of Sass.

4

Organising Sass

So we have covered the basics of Sass and looked at ways we can use mixins and functions to extend the Sass we are writing. With conditional logic, we can start to extend this further. This keeps our working code cleaner and concise whilst Sass takes care of the CSS. This is a key part of making your front-end code more organised.

We can take this idea of breaking our code into smaller, reusable components and apply this to our files.

@import and partials

You may be accustomed to using `@import` within CSS. This lets us break our CSS into different files and then request them as we need them in our CSS. For example, if we are using media queries:

```
@media screen and (min-width: 640px) {
   @import url(tablet.css) }
```

When we are using version control or working across teams, breaking our CSS down into smaller files is a great way to avoid conflicts and keep our CSS rules in thematic groups. However, the one main problem with this approach is that for each `@import`, our browser has to make another HTTP request and using lots of `@imports` can really slow down our *page load time*. With Sass, we can use `@imports` to break our Sass into different files but when it comes to being processed, we would still only output a single CSS file.

In Sass, `@import` is very similar to CSS, except we just need to mention the file name we want to include. We don't even need to include the file extension:

```
@media screen and (min-width: 640px) {
    @import 'views-tablet'; }
```

You can create as few or as many files as you want but if we want to make our code modular, then it is good practice to start breaking our CSS down into thematic files. For example, we might organise our Sass into broad thematic .scss files such as:

```
forms.scss
tables.scss
layout.scss
```

The one problem with this approach is that, if we are using Sass to listen to changes in a folder, we will end up with CSS equivalents for each of these files, even if we'd like to output them all into a single CSS file. The way to avoid this is to use the underscore character (`_`) at the start of each filename. This tells Sass we are treating this as a `partial` file, to be included in other Sass files, but don't want to output it as CSS. We don't need to use the underscore when referencing the file in our Sass:

```
// file _forms.scss
@import 'forms';
```

One benefit of `@imports` in Sass is that it doesn't matter where we include them. With CSS, `@import` rules have to be declared before we start defining any other CSS. With Sass, we can use them anywhere in our .scss files and they will still work. We just need to consider that Sass is processed in sequence, so if we are putting variables, functions, or mixins in an included file, we need to make sure we import it before referencing them.

File organisation

It is beyond the scope of this guide for a detailed discussion on how you should organise your Sass files. This is a matter of great debate and opinion with no correct way. However I agree with Harry Roberts when he says that *"it is a good idea to split discrete chunks of code into their own files."* Personally I tend to group my Sass files into folders based on theme and context.

```
/libs
/components
/views
```

In the `libs` folder I place core files to be used throughout the project, for example a reset file to define some consistent *CSS styles*, as well as my mixins and variables. If I'm using any third-party CSS such as needed for a lightbox gallery or a grid system these would also go in this folder, eg:

```
/libs/_reset.scss
/libs/_variables.scss
/libs/_mixins.scss
/libs/_gridset.scss
```

In my components folder I borrow from the principles of *Atomic Design* to organise my Sass rules into files relating to their specific context of use. For example form elements, navigation, accordians and base typography styles. The thing to remember when organising your Sass files is that you can have as few or as many as you need and can nest these as deep as you want:

```
/components/base/_nav.scss
/components/base/_buttons.scss
/components/base/_type.scss
/components/layout/_header.scss
/components/layout/_footer.scss
/components/theme/_palette.scss
…
```

Finally in my `views` folder I place Sass rules for responsive layouts tied to the specific media queries we will be using in our designs:

```
/views/core.scss
/views/tablet.scss
/views/desktop.scss
```

To retiterate - there is no right or wrong way to organise your Sass rules but the more you arrange these into separate files and folders the easier they'll be to work with.

Commenting your Sass

An essential part of organising your Sass is to use comments. There is no such thing as too many comments when you are writing your CSS. Comments help us to understand the rules we are defining and explain our decision making to others when we have to share our code, either within a team or when our Sass files are part of a deliverable for a client.

In Sass there are two kinds of comments: a double forward slash – `//` – for single line comments and the more familiar method from CSS – `/* */` – for multi-line comments:

```
// this is a single line comment
/*
    this is a
    multi-line comment */
```

The thing to note is that single-line comments will only ever appear in your Sass files but multi-line comments will normally be added to our CSS. Think of single line comments as private development comments and multi-line comments as public comments for describing your CSS output. However, we can control whether comments are included in our CSS through output styles.

Controlling your Sass output

Although we've only used the default Sass output style so far, there are in fact four main styles: nested, expanded, compact, and compressed. The default style is nested and will output in a format showing our nested selectors with indentation to show context and hierachy.

For example, our Sass might be as follows:

```
.heading {
    color: red;
    &-blue {
        color: blue; }}
```

With the default, nested style this outputs CSS as:

```
.heading {
    color: red; }
.heading-blue {
    color: blue; }
```

We can change the style of Sass we produce by using the `--style` command followed by the style we want to adopt. For example, to output expanded CSS we would enter the following in the command line:

```
sass --watch style.scss:style.css --style expanded
```

This would produce the following CSS:

```
.heading {
    color: red;
}
.heading-blue {
    color: blue;
}
```

Using `--style compact` would instead output the following CSS:

```
.heading { color: red; }
.heading-blue { color: blue; }
```

And finally, `--style compressed` outputs the following, minified CSS:

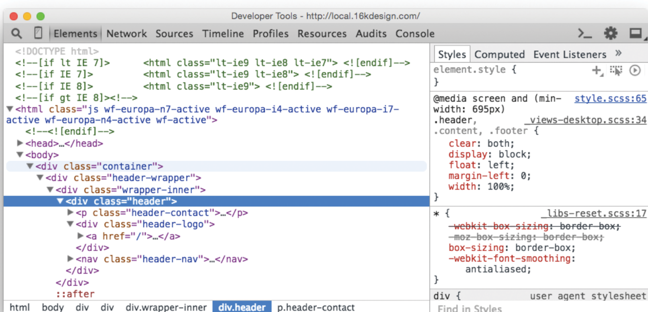
```
.heading{color:red}.heading-blue{color:blue}
```

When we are producing code for our production environment (i.e. our live site), it is good practice to output our CSS as compressed. This keeps the file size down and speeds up our page load time. If we wanted to include comments in our compressed styles (for example, to include copyright or attribution information), then we add an exclamation mark to the start of our comment:

```
/*!
    This comment will be shown, even when compressed */
```

Orientation with Source Maps

The one problem we can encounter with breaking our Sass into different files and controlling their output is that it can become very difficult to debug our CSS when things go wrong. Thankfully, in Sass 3.3, source maps were introduced. You remember earlier when we first compiled our Sass into CSS two files were created: `style.css` and `style.css.map`? Well, the `.css.map` file is there to help us know where our CSS rules are coming from. The Development Inspector in most modern browsers would normally show us where a CSS rule is coming from by telling us the line of the CSS file that is applying that rule. That is no good to us when we have broken our Sass down into different files. With Source Maps, when we compile our CSS and view the end result in a browser, the Inspector will show us the location of the Sass rules that are affecting that element:



Source maps are supported in the more modern browsers: Chrome, Firefox and Safari. In Chrome, you may need to enable this option by opening the `Dev Tool` settings and toggling the `Enable CSS Source Maps` option under Sources. Internet Explorer 11 in Windows 8.1 also offers source map support.

We can now see how partials are able to help us organise our Sass into distinct files and with source maps, we can see how and where our Sass affects our CSS. With comments, we can help others understand the CSS we write and by using the style command, we can change the format of the CSS we are producing. In the next chapter, I will talk you through the various ways you can level up what you have learned so far through extending the core features of Sass.

5

Level up your Sass

In this guide so far, the examples have been intentionally simple to help explain the terms and techniques of Sass. In this chapter, I want to look at some more complicated applications for Sass and see how we can start to use this in our real-life projects.

Dealing with errors

CSS is a pretty fault-tolerant language. If you use the wrong syntax or put a semi-colon out of place, it is pretty unlikely that your styles will break. However, as a pre-processor, Sass is pretty strict when you make mistakes. Chances are that in the course of following this guide, you have probably hit a few errors yourself. Errors in Sass will likely cause the CSS not to compile properly, if at all. Fortunately, Sass is pretty good at explaining the source of processing problems. For example, if I write the following:

```
.nav { background-color: red: }
```

Can you spot the error? Sass will throw the following back in the command line, telling me where I have made a mistake:

```
error style.scss (Line 1: Invalid CSS after " color: red":
    expected ";", was ":")
```

Furthermore, if we try to access the page requesting our CSS, we will be presented with a bit more context for the offending error:

Media query bubbling

The arrival of Sass has coincided with increasingly sophisticated CSS. The emergence of responsive web design in particular and its three tenets of fluid grids, media queries and *flexible images* have led to more complicated demands on our CSS. As we saw in the last chapter with media queries, it can be relatively straightforward to group our Sass into the context of how our CSS is going to be applied (e.g. organised into mobile, tablet and desktop views). At the same time, however, many front-end developers have started to adopt a

more modular method to creating their CSS. Concepts like *atomic design* and object *oriented CSS* have meant breaking your CSS down not necessarily into the context of delivery, but rather into more thematic contexts such as forms, tables, navigation, etc.

Whilst this modular approach is not necessarily incompatible with the broader ideas of responsive web design, the practicalities of writing and maintaining CSS that supports both approaches can rapidly become very complicated. However, Sass makes this much more straightforward through the concept of media query bubbling.

To provide an example: if we wanted to apply a media query to a specific element in CSS, we would write the following:

```
.aside {
    width: 100%; }
@media screen and (min-width: 800px) {
    .aside {
        float: right;
        width: 35%; }}
```

This is simply saying that by default, our `.aside` element is full width but once our viewport reaches 800px, we want the `.aside` element to be floated right with a width of 35%. The problem with the above is that we end up with a lot of repetition because we have to write our selector out twice.

With Sass, we can place our media queries within our parent selector and when outputted to CSS, the query will be ‘bubbled’ up. So taking the above example, we can write:

```
.aside {
    width: 100%;
    @media screen and (min-width: 800px) {
        float: right;
        width: 35%; }}
```

This produces exactly the same CSS we had before:

```
.aside {
    width: 100%; }
@media screen and (min-width: 800px) {
    .aside {
        float: right;
        width: 35%; }}
```

Media query bubbling then provides a way of keeping our media queries much more focussed. The one problem with this approach is that we end up writing out our media queries multiple times. We can simplify this if we use variables to determine our common breakpoints. For example, we might want to define a set of variables outlining our main breakpoints as follows:

```
$mobile: 640px;
$tablet: 800px;
$desktop: 1024px;
```

Or:

```
$break-small: 640px;
$break-medium: 800px;
$break-large: 1024px;
```

We can then use these variables within our media queries. That way, if we want to change our breakpoints at a later stage we only have to update the corresponding variable and this will cascade throughout our CSS.

```
.aside {
    float: none;
    width: 100%;
    @media screen and (min-width: $break-medium) {
        float: right;
        width: 35%; }}
```

Another way we can level-up our media queries is to write our own mixin to save us having to write out our media queries repeatedly:

```
@mixin responsify($breakpoint) {
    @media (min-width: $breakpoint) {
        @content; }}
```

This is the same as the mixins we created in Chapter 3 but introduces a new command: `@content`. The `@content` block is used to reproduce a set of style rules we define when requesting our mixin (note the curly brackets following our mixin `@include`):

```
.article {
    @include responsify($break-small) {
        float: left; }}
```

This takes the properties included in our .article declaration and wraps them in the CSS generated by our mixin. So the above example will be outputted as:

```
@media (min-width: 640px) {
    .article {
        float: left; }}
```

Clever hey? This keeps our Sass much leaner and gives us the opportunity to modify our media queries by updating the mixin. For example, if we wanted to specify a maximum width as well as a minimum width we can add more arguments to our mixin:

```
@mixin responsify($min, $media: screen, $max: false) {
    @if $max {
        @media #{ $media} and (min-width: $min) and (max-width:
        $max) {
            @content; }}
    @else {
        @media #{$media} and (min-width: $min) {
                @content; }}}
```

Here, we are passing three arguments into our mixin: minimum width (`$min`), media type (`$media`) and maximum width (`$max`). For media type and maximum width, we are setting default values in case these aren't entered as arguments: a default value of screen for `$media` and a default value of false for `$max`. This means that if an argument for `$max` is not entered, it will assume we only want to generate a media query for `min-width`.

In our mixin, we use an `@if` condition to see if an argument has been entered for `$max`. If so, we then want to generate a `max-width` in our media query. So despite updating our mixin, our original example would still work:

```
.article {
    @include responsify($break-small) {
        float: left; }}
```

However, by adding more arguments, we can generate more powerful and flexible media queries:

```
.article {
    @include responsify($break-small, screen, $break-medium -
    1) {
        float: left; }}
```

Which will output:

```
@media screen and (min-width: 640px) and (max-width: 799px) {
    .article {
        float: left; }}
```

Note we are reducing our `max-width` value by 1. When we use media queries for responsive design, we need to be careful that we don't create rules that overlap breakpoints. This ensures that our media query will only go up to the defined maximum breakpoint. If we then want to add a media query for `$break-large`, we know this will start at 800px and not overlap with the queries we have already defined.

Sass built-in functions

As well as writing our own functions and mixins, Sass comes with a range of its own functions that we can access. We have already touched on some built-in functions – for example, the `index()` and `nth()` functions – and whilst the full list of Sass functions are too numerous to cover in this guide, it's worth looking at a couple of examples.

Some of the most helpful Sass functions make working with colours easier. One I use all the time is `rgba()`. This is similar to the `rgba()` value we have in CSS but in this instance, we don't necessarily have to use RGB colours (such as 255,0,0). Instead, we can throw in a hexadecimal number and it will still work:

```
$red: #bd250a;
.alert {
    background-color: $red;
    background-color: rgba($red, 0.75); }
```

Here, we are defining the value for $red and then applying it twice. First, to provide a fallback for browsers that don't support `rgba()` – IE8, I am looking at you! – and another to provide a degree of transparency to the colour for more modern browsers. We could even put this in a simple `@mixin` if we were going to write this out repeatedly:

```
@mixin rgbaify($property, $colour, $opacity: 1) {
    #{$property }: $colour;
    #{$property }: rgba($colour, $opacity); }
```

You'll recall the use of the hash (`#`) character to write variables into our selectors from when we discussed lists and Sass maps. We would then access this mixin as follows:

```
.alert {
    @include rgbaify(background-color,$red, 0.75) }

// produces
.alert {
    background-color: #f00;
    background-color: rgba (255, 0, 0, 0.75); }
```

As well as the colour functions, there are a range of functions to help us work with strings, lists and maps. For example, we can use `round()`, `ceil()` and `floor()` to help us work with calculations or decimal numbers:

```
.icon {
    // round produces 26px (rounds to nearest integer)
    width: round(25.5px);
    // ceil produces 3px (rounds up to nearest integer)
    width: ceil(25px / 10);
    // floor produces 2px (rounds down to nearest integer)
    width: floor(25px / 10); }
```

The best way of learning about the full range of Sass functions is to dig in to the documentation, which is *full of useful examples*.

Sass Maps

The final thing I want to cover in this chapter is Sass maps. We looked at Sass maps when we looked at conditionals in the Chapter 3, but it is also worth looking at some of the more complicated things we can achieve with them.

Sass Maps come with a range of functions to help us work with lists of `key:value` pairs. However, the most useful function is probably `map_get()`. This function lets us fetch the value from our map based on its key.

Let's use the map we employed in Chapter 3 to define a colour palette for a site. In that example, we used the `@each` conditional to loop through our `key:value` pairs. However, we can use `map_get()` at any time to extract the value we need:

```
$pages: (
    home: #F7E900,
    about: #FF5F09,
    news: #A0005E,
    links: #41004B);

.news {
    background-color: map_get($pages, news); }

// produces
.news {
    background-color: #A0005E; }
```

We can then use the Sass built-in functions to build a palette based around these colours. For example, we might want a border that is darker than our `background-color` and a `font-color` that is lighter than our core colour, which we can achieve with Sass' `darken()` and `lighten()` functions by simply expressing the amount (as a percentile) by which we want to increase or decrease the brightness of this colour:

```
.news {
    background-color: map_get($pages,news);
    border-color: darken(map_get($pages,news), 25%);
    font-color: lighten(map_get($pages,news), 25%); }

// produces
.news {
    background-color: #A0005E;
    border-color: #200013;
    font-color: #ff21a3; }
```

Again, we could probably wrap this in a mixin to make our lives easier if we are iterating through several different colours. However, another way we can work with variability in our colour palettes is by creating a nested map. *This technique was first suggested by Tom Davies at Erskine Design* and lets us organise our colour palette into consistent, logical groups:

```
$palettes: (
    blue: (
        light: #9cb9c8,
        base: #669db3,
        dark: #f5814d),
    red: (
        light: #9cb9c8,
        base: #669db3,
        dark: #f5814d),
    orange: (
        light: #9cb9c8,
        base: #669db3,
        dark: #f5814d));
```

We can then access our colour values by using a function which performs a `map-get()` in a `map-get()` function. Confused yet? Let's have a look at this function:

```
@function palette($palette, $tone: base) {
    @return map-get(map-get($palettes, $palette), $tone); }
```

Here we are using `$palette` to get the colour we want to work with and $tone to define the tone of our colour. We set a default tone of 'base' so the user doesn't have to specify this second argument. In the function, we use `map-get()` twice: once to get the values from the list corresponding to our colour and again to get the child value from our list corresponding to the desired tone.

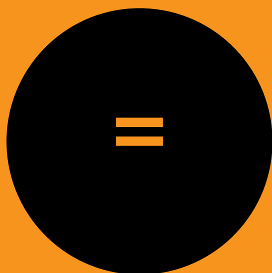
To access this function we would then do the following:

```
.box {
    color: palette(blue);
    &-border {
        border: 1px solid palette(blue, dark); }}

// produces
.box {
    color: #669db3; }
.box-border {
    border: 1px solid #f5814d; }
```

This is a really clever technique for keeping our colour variables tight and consistently defined. No more `$grey-light`, `$grey-lighter`, `$grey-lightest` in our Sass!

There are more `map-` functions but that is the subject of another book altogether!



Conclusion

So, we've covered quite a lot in these five chapters. We've talked about how to get up and running with Sass and some of the basics of Sass including variables, functions, and mixins. We've seen how we can introduce logic to write more sophisticated Sass and learned about how to organise our Sass files. However, we've just scratched the surface. New features are added to Sass regularly and with third-party libraries such as *Compass* and *Bourbon*, we can start to write more powerful Sass and extend the core libraries of functions and mixins available to us. However, that is the subject of another guide!

If you are itching to take what you have learned in this guide further, there are some great resources out there to discover more advanced Sass techniques. I can heartily recommend Hugo Giraudel's *Sass Guidelines* about setting up and maintaining Sass. The Sass Way is a great resource featuring articles covering beginner, intermediate and advanced *Sass techniques*. Stu Robson's excellent *Sass News* is a weekly newsletter with the latest in Sass news, articles, tutorials, and events. Finally, definitely keep an eye out for Roy Tomeij's forthcoming online course, *Advanced Sass*.

Hopefully, this guide will have given you enough confidence and knowledge to start writing better CSS with Sass and using it in your future projects.