Brain Upgrade — Learn Anything In Under 2 Hours

CSS Preprocessor

Sass

Level 0 - For beginners

Jérémy Mouzin

Brain Upgrade Book Concept

You're busy. I get it.

You want to learn Sass from scratch fast.

You don't want to spend hours deciphering boring documentation or searching answers on StackOverflow.

Brain Upgrade collections are explicitly designed for this. In this book you'll get:

- Straight to the point concise and clear core concepts
- Well designed examples that actually teach you something
- Everything you need to be up and running for Sass production code in under 2 hours
- Well explained answers of the top 10 most asked questions about Sass on StackOverflow

No introduction, no history, no fluff, no bullshit, no conclusion. Just a Fast and Happy™ learning!

Contact For Help

If you have any problem while applying the knowledge in this book **please contact me**. I'm here to help you learn, don't be shy! I don't want you to be stuck, I know how much frustrating it is.

I would have loved to have such a free service while learning Sass so don't hesitate to contact me, you'll make me happy :o).

live:jeremy.mouzin
jeremy.mouzin@gmail.com

I'm available on Skype to help you live from Monday to Friday (9:00 AM — 6:00 PM UTC+01:00).

I'm waiting for your call.

Why Learning Sass?

When you write a lot of CSS your stylesheets quickly become huge, repetitive, and difficult to maintain. CSS is a great language, but it lacks lots of features to help you write better stylesheets, fast.

You may also want to use again some particular resources like color palettes or a specific design across several projects.

Sass lets you ease the creation and the maintenance of your stylesheets, allow you to reuse previous work in seconds and will make you save a ton of time in the process.

Bonus point: it's a highly demanded skill by employers, so if you're looking for a front-end dev job nowadays, don't hesitate to learn it! Keep reading!

Let's install this marvelous tool! You can skip this step and directly use SassMeister online tool instead. It will produce CSS from your Sass syntax right from your browser!

Installation

This may be the most difficult part of this book, so please don’t hesitate to call me to get help.

Open a terminal to get a command line and install Sass on your computer.

If you’re on Linux, type:

command line

```
sudo gem install sass —no-user-install
```

If you’re on MacOS, type:

command line

```
sudo gem install sass
```

If you’re on Windows:

1. Install Ruby installer
2. Run `cmd` to open a terminal window
3. Type:

command line

```
gem install sass
```

Sass should now be installed on your computer. Double-check by typing:

command line

```
sass -v
```

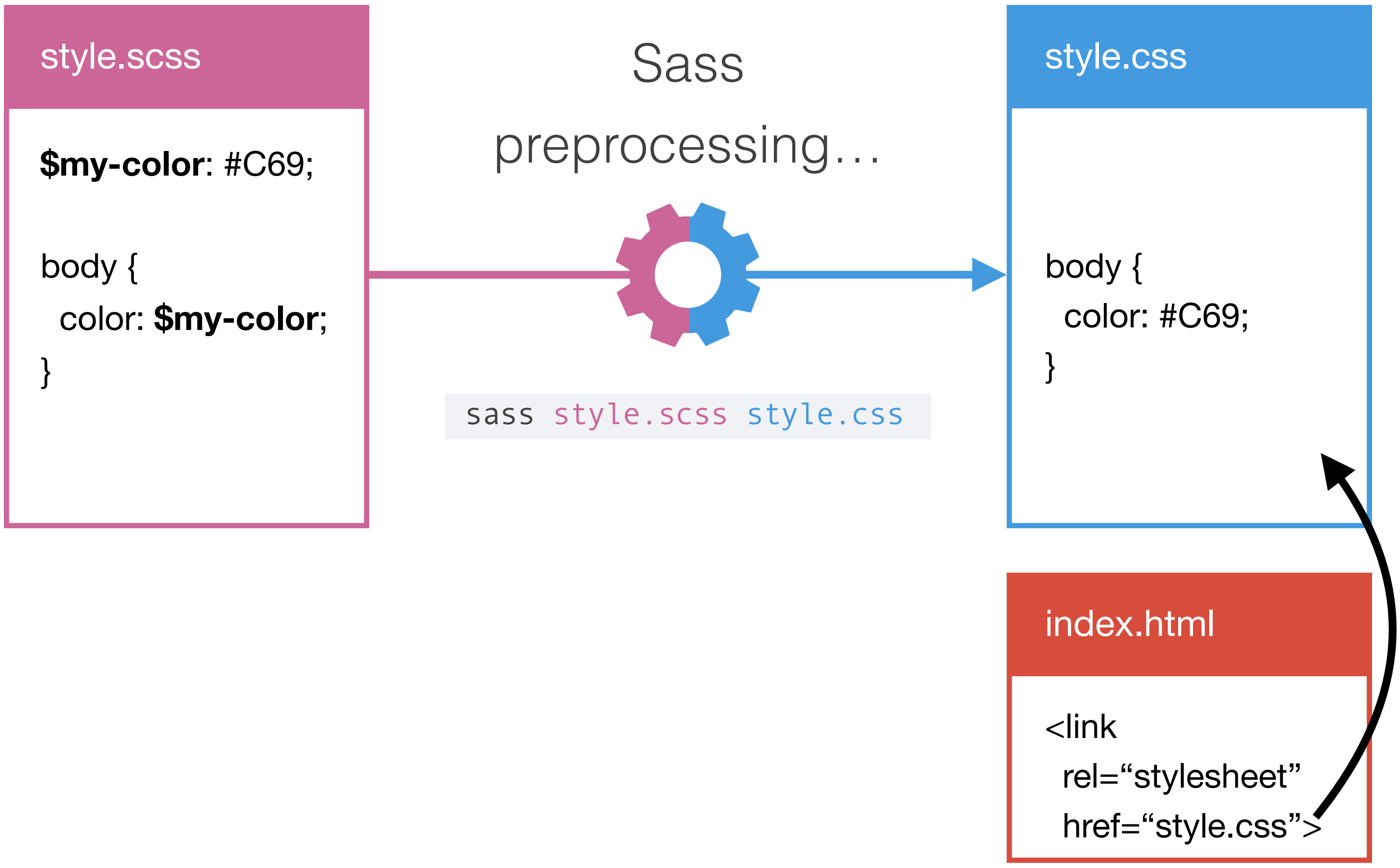
It should return **Ruby Sass 3.5.6** or a newer version.

Congratulations! You’ve successfully installed Sass on your computer, you’re ready to go!

How Sass Works

Sass transforms a source file that uses Sass syntax into a valid CSS stylesheet, ready for production.

Sass source files use **.scss** extension. SCSS means **S**assy **CSS.** The output is a CSS stylesheet with **.css** extension.



```scss
$my-color: #C69;

body {
  color: $my-color;
}
```

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

```css
body {
  color: #C69;
}
```

```html
<link
  rel="stylesheet"
  href="style.css">
```

Linking to a Sass source **.scss** file from your HTML won't work. You must use the generated **.css** file.

Generate Your First Stylesheet

Create a file **style.scss** with the content below:

style.scss

```
h1 {
  color: #CC6699;
}
```

This file doesn't use any Sass features yet. But you can already use it as a source file.

From the directory where you created **style.scss**, type:

command line

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

This creates the **style.css** file that you will use in your HTML header.

Note that this command also generates one new file and one new hidden directory:

```
.sass-cache/     // Hidden directory (starts by a dot)
style.css.map    // Map file
```

Plus, you'll see a strange line at the end of the generated CSS stylesheet:

style.css (end of file)

```
/*# sourceMappingURL=style.css.map */
```

Don't worry about these. They're just Sass internal plumbing to speed up the preprocessing computation.

They are useful only for the generation of the CSS stylesheet from your source file. Don't upload them to your website, they're not necessary.

Once you're done developing you can delete them safely.

Speed Up Your Development Process

It's cumbersome to always have to run **sass** command line tool to produce the corresponding CSS stylesheet.

To ease the development process, you should use the option **--watch** like this:

command line

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

Note the colon ':' between the source and the destination file. Don't forget it or it won't work.

This text will appear in your terminal after a few seconds:

```
>>> Sass is watching for changes. Press Ctrl-C to stop.
      write style.css
      write style.css.map
```

Sass tells you that it's now watching your source file and has just written **style.css** and **style.css.map**.

Don't close your terminal window or hit Ctrl+C otherwise **sass** won't watch your files anymore.

If you need a new terminal, just open a new terminal window.

The magical **--watch** option will detect automatically the changes in your Sass source file and process them to produce the new corresponding CSS stylesheet.

Each time you change your **style.scss** source file and save it, the **style.css** file will be rewritten.

This way you always see the results in your **style.css** file while working only on **style.scss**.

Watch Your Files And Make Your First Change

Monitor your source file **style.scss** by typing:

command line

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

// Wait for these lines to appear
>>> Sass is watching for changes. Press Ctrl-C to stop.
      write style.css
      write style.css.map
```

Now modify **style.scss**, change the color value from #CC6699 to #FF9800:

style.scss

```
h1 {
  color: #FF9800;
}
```

As soon as you save your modifications, you'll see the command line prompt display a new message:

command line

```
>>> Change detected to: style.scss
      write style.css
      write style.css.map
```

Check your **style.css** content, it should be updated:

style.css

```
h1 {
  color: #FF9800; }
```

Congratulations! You've just generated automatically your CSS from your Sass file!

You can see that the last closing curly bracket is not on its own line like it was in the original Sass source file. That's normal, it’s just the default *nested* style that is applied to your output by the **sass** command line tool.

If it bothers you, you can set the style to *expanded* by using the **--style** option like this:

command line

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

Modify the original source file, save it and you’ll see the curly bracket on its own line again.

The different options available for the style flag are here. In the rest of this book we’ll use the default *nested* style.

Now that you know how to work with Sass, let’s spice things up by learning some very nice features!

Variables

You can store any CSS value you can think of in variables to reuse them in your stylesheet wherever you want.

A variable must be first declared before using it.

Example

style.scss

```scss
// Declare a variable named hop-brush with value #C69
$hop-bush: #C69;

// Use the variable to set body font color
body {
  color: $hop-bush;
}
```

You can use variables for all CSS types of values: colors, font sizes, distances, font families etc.

Example

style.scss

```scss
// Use any valid CSS color format
$hop-bush: #CC6699;
$hop-bush: rgb(204,102,153);
$hop-bush: hsl(330, 50%, 60%);

// You can use numbers in variables names
$padding: 16px;
$padding-2x: 32px;
$padding-3x: 48px;

// Due to historical reasons, hyphens and
// underscores are interchangeable
$chameleon-width: 20%;
$chameleon_height: 30%;
#chameleon {
  // Variables names are valid
  width: $chameleon_width;
  height: $chameleon-height;
}

// You can really use any valid CSS value format
$border: 1px solid green;
$margins: 10em 12px 8rem 0;
```

```
// Strings are valid no matter if you use no quote,
// simple or double quotes
$default-font: Source Sans Pro, "Helvetica", 'sans-
serif';
```

Even if hyphens and underscores are interchangeable in variables names, you should stick with only one or the other. Stay consistent.

Variables Scopes

Sass allows you to declare variables inside a block.

But first, some definitions.

style.scss

```scss
body {
  $my-color: #042;    // Declare $my-color variable
  color: $my-color;
}
```

Block: file area contained between an opened curly bracket and a closed curly bracket.

Variable scope: file area where you can access and use a variable.

By default, the scope of a variable starts from its declaration to the end of the current block.

Therefore the scope of the variable **$my-color** is the whole **body** selector rule block represented by the vertical blue line to the left.

For variables declared outside of any block, they're accessible from the declaration until the end of the file. They're also accessible **within any block** in that space.

Let's see an example of several variables declared outside and inside blocks.

Example

style.scss

```scss
$global-color: #F16;

html {
  background-color: $my-color;     // Not accessible
}

$my-color: #C69;

body {
  $local-color: #F1C;
  color: $my-color;                // Correct
  background-color: $local-color; // Correct
}

div {
  background-color: $my-color;     // Correct
  color: $global-color;            // Correct
  border: 1px solid $local-color  // Not accessible
}
```

Make A Variable Scope Global

You can make a variable accessible from outside its declaration block by using the **!global** flag like this:

style.scss

```scss
body {
  $max-width: 800px;
  $max-width-global: 800px !global;

  max-width: $max-width;          // Correct
  max-width: $max-width-global; // Correct
}

#main {
  max-width: $max-width;          // Not accessible
  max-width: $max-width-global; // Correct
}
```

The **!global** flag is useful when you use several different Sass source files to produce one CSS stylesheet.

Nevertheless, you should use it sparingly because it makes the source files less readable in term of which variable you can use or not at your current position.

You know the !important flag in CSS? Well, the !global one is quite similar, use it as a last resort.

Congratulations! You know everything there is to know about variables, you'll love using them!

Comments

They're the same as in CSS.

One line comments start with //.
Multiline comments are surrounded by /* */.

One line comments are **not** copied to the CSS stylesheet.

Multiline comments **are** copied to the CSS stylesheet.

Example

style.scss

```
// This comment won't be found in the final stylesheet
// This one neither :(

/* This one will: it uses multiline delimiters! :) */

/* You can store some useful information using
comments, like the version of your stylesheet */
```

This will produce:

style.css

```css
/* This one will: it uses multiline delimiters! :) */

/* You can store some useful information using
comments, like the version of your stylesheet */
```

Interpolation #{ }

Sass variables can be used in CSS selectors and property names. But you can't just use the **$var** syntax. Indeed you **cannot** do this:

you-cannot-do-this.scss

```scss
$class-name: my-great-class;
$attribute: border;

p.$class-name {              // Incorrect
  $attribute-color: blue;   // Incorrect
}
```

Sass will try to use the **$attribute-color** variable (instead of **$attribute**) but it doesn't exist. To delimit your variable you must use interpolation like this **#{**$var-name**}**.

When you use interpolation on a variable you simply replace it with its value. Think of it as a simple search & replace feature for variables.

Example

style.scss

```
$class-name: my-great-class;
$attribute: border;

p.#{$class-name} {              // Correct
  #{$attribute}-color: blue;   // Correct
}
```

will produce:

style.css

```
p.my-great-class {
  border-color: blue; }
```

You can even use interpolation within comments to store the version of your website or web app:

style.scss

```scss
$version: 1.4;

/* Version: #{$version} */
```

will produce:

style.css

```css
/* Version: 1.4 */
```

Project Structure Architecture

Let’s take a break from Sass syntax and think about how and where to store your files.

Small Project

The simplest structure is everything in the same directory.

```
project
├── index.html
├── style.css
└── style.scss
```

That’s simple and work for most little projects. It’s the perfect structure to learn Sass.

But when you start to work on bigger projects, it will become quickly limited.

Medium Project

If you have lots of files and several different stylesheets you should use a directory to store them.

```
medium-project
├── index.html
├── *.html
└── stylesheets
    ├── style.css
    ├── style.scss
    ├── *.css
    └── *.scss
```

If you don’t want to mess up your source files and the CSS stylesheets that are used for production, you should separate the sources from the outputs.

It’s a good practice to upload only your output files on your server and keep your sources in your source control system only.

To achieve this, you can use this structure:

```
better-medium-project
├── index.html
├── *.html
├── src
|   └── style.scss
|   └── *.scss
└── stylesheets
    └── style.css
    └── *.css
```

You would **not** upload the **src** directory to your web server, but just the HTML files and **stylesheets** directory.

Big Project

For big projects with lots of files, reusable Sass stylesheets and design systems in place, you should check out this excellent article from The Sass Way or the 7:1 pattern from Sass Guidelines.

Final Note About Architecture

There are thousands of ways to structure your files for a project. You're free to use the structure you want, but keep in mind these good practices.

BEST PRACTICES

1. Never upload source file to your web server, it will bloat it over time.

2. Separate your source files from your output files. Use a separate directory for each type. Common directories names for source files are **src**, **sass** or **scss**. Common directories for output files are **dist**, **css**, **remote**, **stylesheets**. Pick one combination and stick with it.

3. Never use spaces in directory names or filenames. Instead, use a hyphen or underscore like this `my-variables.scss` or `my_variables.scss`. I prefer using hyphens because they're easier to type and read.

Nested Rules

In CSS, it’s common to repeat selector rules like this:

boring-repetition.css

```css
h2 {
  font-size: 1.4em;
  margin: 1.8em 0 0.5em;
}

h2 .small {
  font-size: 0.7em;
  font-weight: normal;
}

h2 .medium {
  font-size: 1em;
  font-weight: normal;
}
```

To avoid these repetitions, Sass introduces **nested rules**.

To use them, simply put your children declarations (`.small` and `.medium`) within the parent rule (`h2`) like this:

boring-repetition.scss

```scss
h2 {
  font-size: 1.4em;
  margin: 1.8em 0 0.5em

  // This will produce the h2 .small rule
  .small {
    font-size: 0.7em;
    font-weight: normal;
  }

  // This will produce the h2 .medium rule
  .medium {
    font-size: 1em;
    font-weight: normal;
  }
}
```

Nested rules can work on several levels of nesting. Let's see this in an example.

several-levels-of-nesting.scss

```scss
.parent {
  color: #000;

  // One level deep
  .child1, .child1a {
    color: #111;

    // Two levels deep
    .child2 {
      color: #222;
    }
  }

  // Back to level one
  .child3 {
    color: #333;
  }
}
```

will produce:

several-levels-of-nesting.css

```css
.parent {
  color: #000; }

  // One level deep
  .parent .child1, .parent .child1a {
    color: #111; }

    // Two levels deep
    .parent .child1 .child2, .parent .child1a .child2 {
      color: #222; }

  // Back to level one
  .parent .child3 {
    color: #333; }
```

By default the **nested style** applied to the CSS output lets you see quickly the nested level thanks to indentation.

One indentation for the first level, two for the second, etc.

Referencing Parent Selector With The & Special Character

Let's say you want to specify a rule for the `:hover` state of a `.parent` class. You need the selector `.parent:hover` so you write something like this:

this-will-not-work-sorry.scss

```
.parent {
  :hover {
    text-decoration: none;
    border-bottom: 1px solid blue;
  }
}
```

You guessed it, it doesn't work!

Indeed it will produce something totally coherent for nested rules normal processing flow but not very useful for what you want to achieve:

this-will-not-work-sorry.css

```
// See the space problem here?
.parent :hover {
  text-decoration: none;
  border-bottom: 1px solid blue; }
```

Sass naturally puts a space between the `.parent` and the `:hover` child selector. So how can you produce the `.parent:hover` selector?

By using the & special character. Here is the solution:

this-works.scss

```
.parent {
  // Using the special & char will make the trick
  &:hover {
    text-decoration: none;
    border-bottom: 1px solid blue;
  }
}
```

will produce:

this-works.css

```
// Now you’ve got .parent:hover without any space!
.parent:hover {
  text-decoration: none;
  border-bottom: 1px solid blue; }
```

Using the & character makes Sass behave like a dumb search and replace feature.

Indeed, in the last example, the special & character has just been replaced by the parent selector value within the child value.

Let’s decompose the process so you can understand the difference between the normal flow and the use of &.

Let’s first set a generic approach to how everything works in general:

generic-approach (pseudo Sass code)

```
<parent> {
  <child> {
    color: #111;
  }
}
```

Let’s use `<parent> = .parent` and `<child> = .child`. The normal flow to produce the output of this file is:

1. `<parent> <child>`
2. `.parent .child`

Nothing fancy here, we replace each part with its value. It’s a normal flow.

Let’s use `<parent> = .parent` and `<child> = &:hover`. Sass output for this is just “`.parent:hover`” as we’ve seen earlier.

But if you respect the normal flow, step-by-step, it should produce something like this:

1. `<parent> <child>`
2. `.parent &:hover`
3. `.parent .parent:hover`

Do you spot the difference on the output (step 3) here? There is another `.parent` at the beginning!

So why Sass doesn't produce this selector?

It's because using the & special character breaks the normal flow and use instead a very simple search and replace function on the `<child>` value only.

Therefore if Sass sees a & character in the child value, the step 1 "`<parent> <child>`" is replaced only by "`<child>`". This value becomes "`&:hover`", which becomes "`.parent:hover`".

That's an important concept to understand because thanks to the & special character you'll be able to create selectors really easily! Let's see some advanced examples.

advanced-example.scss

```scss
.parent {
  // You can use special & char wherever you need it!
  // At the beginning, in the middle or at the end
  &:hover { color: #000; }
  .child1 & .child1a { color: #111; }
  .child2 & { color: #222; }

  // Or at a deeper nested level
  .child3 {
    .child3-1 & { color: #333; }
  }

  // And even near other words!
  &-small { font-size: .5em; }
}
```

Read carefully the produced code below. You'll see that the & character makes Sass behave like a search and replace simple feature:

advanced-example.css

```css
.parent:hover {color:#000}
.child1 .parent .child1a {color:#111}
.child2 .parent {color:#222}
.child3-1 .parent .child3 { color:#333}
.parent-small { font-size: .5em: }
```

Using the & char will let you generate classes names very easily based on the parent value. For example, you can use that flexibility to build responsive containers:

responsive-containers.scss

```scss
.container {
  max-width: 800px;

  &-medium { max-width: 600px; }
  &-small { max-width: 400px; }
}
```

Very handy to generate automatically classes names for slightly different containers:

responsive-containers.css

```
.container { max-width: 800px; }
.container-medium { max-width: 600px; }
.container-small { max-width: 400px; }
```

You can go one step further and use variables to manage entirely the names of your classes right from one place!

one-place-to-rule-them-all.scss

```
$container: '.container';

// Remember to use interpolation to make it work here!
#{$container} {
  max-width: 800px;

  &-medium { max-width: 600px; }
  &-small { max-width: 400px; }
}
```

Code Architecture Best Practices

We already talked about project architecture best practices. But what about your code architecture? How do you manage your code, variables, etc.?

BEST PRACTICES

1. Apply the **DRY** principle (**D**on't **R**epeat **Y**ourself): if you need to change a color, a padding or whatever for the whole website you're coding, it should be easily done by changing only one value at one place.
2. Extract common parts of your source files across all your projects and put them into separate files. Reuse them for new projects to speed up development. It will also ensure design consistency across all your work. This is called factorization or modularization.
3. Apply the **KISS** principle (**K**eep **I**t **S**tupid, **S**imple): your source files should be small. If they get too big,

split them into several files. One file for one purpose. It's better to work on several small files than one huge file.

4. Extract all your personal style to a specific stylesheet: your preferred margins, font families, border-radius values, shadows… so you can apply them to a project in seconds.
5. Take the time to name your variables properly. Remember that you may read your code several months after writing it so your variables names should be meaningful and easy to use.
6. Use hyphen-delimited variables names like this `$my-variable`.
7. Prefix your names to the purpose of the variable, for a color use `$color-`, for a margin, use `$margin-`, etc. Why? Because your IDE auto-completion feature will kick in right from the start and you'll save a ton of time!

All the best practices have been listed, how can you apply them? Well, that's the purpose of the next chapter.

You’ll see how to use **partials** and the **import** feature to create a clean and flexible code architecture.

Partials and Import

Using several Sass source files in your project will make it easier to maintain and more flexible.

You could already do this with the CSS built-in import feature but it's bad for network performances.

Indeed it requires one HTTP request for each file imported through CSS import. That slows down the loading of your website. Bad for users and for search engines ranking!

Always ship the least amount of CSS stylesheets possible to minimize network payload.

How can you use several small source files and output only one single file?

You must combine these several small source files together into a single CSS stylesheet.

Import

Thankfully, Sass comes with a handy feature for that called `@import`. The keyword `@import` will allow you to import any source file within another source file. This way at the end you can combine any number of files together into a single one.

Most of the time you'll use the **import** feature in conjunction with **partial** files.

Partials

Partial files are regular Sass source files with one particularity: their name starts with an underscore '_'.

This underscore prefix will tell Sass to not generate the corresponding CSS output for this file. This is extremely useful to keep your output directory clean.

Thanks to the `@import` feature, these partial files will allow you to modularize your project into small pieces and bring them back together into one single file. Let's take an example.

my-current-main-big-source-file.scss

```scss
/* Color branding */
$color-primary: #C69;
$color-accent: #699;
$color-background: white;
$color-text: #6b717f;

/* Spacing */
$margin-h1: 2em;
$margin-h2: 1.2em;
$margin-h3: 0.8em;

$padding-p: 1.4rem 1rem;
$padding-pre: 2rem 1rem;
$padding-div: 1rem;

/* Visual style */
$border-thickness: 2px;
$border-radius: 6px;
$shadow-blur: 4px;
$shadow-spread: 6px;

/* Imagine lots of rules using these variables below */
// […]
```

This file can really start to get big quickly. To ease the maintenance and flexibility you should split it into 4 files: 3 partials **_colors.scss**, **_spacing.scss**, **_visual.scss** and 1 main source file called **style.scss**:

_colors.scss

```
$color-primary: #C69;
$color-accent: #699;
$color-background: white;
$color-text: #6b717f;
```

_spacing.scss

```
$margin-h1: 2em;
$margin-h2: 1.2em;
$margin-h3: 0.8em;

$padding-p: 1.4rem 1rem;
$padding-pre: 2rem 1rem;
$padding-div: 1rem;
```

_visual.scss

```
$border-thickness: 2px;
$border-radius: 6px;
$shadow-blur: 4px;
$shadow-spread: 6px;
```

To polish the code architecture you can separate those partial files from the regular ones and put them into a **partials** directory like this:

```
project
├── index.html
├── src
|    ├── partials
|    |    ├── _colors.scss
|    |    ├── _spacing.scss
|    |    └── _visual.scss
|    └── style.scss
└── stylesheets
     └── style.css
```

This code architecture will allow you to use a single command line to manage your whole project simply. From the **project** directory, type:

command line

```
sass --watch src:stylesheets
```

This simple command will tell Sass to take all the **.scss** files contained in **src** and output the corresponding CSS stylesheets into **stylesheets**.

Note that the **sass** program looks also for **.scss** files in subdirectories.

So if you add a subdirectory **unicorn** in the **src** subdirectory containing a new Sass file **beautiful-unicorn.scss**, it will be reflected in the **stylesheets** directory like this:

```
project
├── index.html
├── src
|   ├── partials
|   |   ├── _colors.scss
|   |   ├── _spacing.scss
|   |   └── _visual.scss
|   ├── style.scss
|   └── unicorn
|       └── beautiful-unicorn.scss
└── stylesheets
    ├── style.css
    └── unicorn
        └── beautiful-unicorn.css
```

Let’s see what the **style.scss** will be made of now.

Bring Everything Back Together

To include those partial files into the main source file, we must use the Sass `@import` feature.

It’s very easy to use: just provide the path to the file you want to include, but without the leading underscore and the extension. To include the file **partials/_colors.scss**, you just add this line at the beginning of the main file:

```
@import “partials/colors”;
```

Let’s build our **style.scss** related to our previous example to see this feature in action:

style.scss

```
@import “partials/colors”;
@import “partials/spacing”;
@import “partials/visual”;

/* Imagine now lots of rules using these variables */
// […]
```

That’s it! Modularizing your Sass source files like this is extremely powerful, here are some advantages:

1. If you’re on a big team, a designer that wants to tweak colors can do it modifying only one value in one file.

2. You avoid the nightmare of merging several persons work on the same huge file every day.
3. If you want to see how a website feels with a new color palette, you can simply switch the path of the import line in the main style.scss file to use your brand new partials/_colors-awesome.scss palette file!
4. Working on small files reduces the cognitive load and the need to scroll up and down to find the things you need to modify.

But that's not finished! You can go one step further.

You can also split the CSS rules into several files. There is no restriction here.

For example you could create a _buttons.scss that handles whatever CSS rules is related to buttons. You could create a _forms.scss partial file to manage forms, etc.

The more separated files you have the more flexibility you'll get to create from scratch a website with a specific combinations of all these files.

Feel free to explore the best architecture for your needs and constraints, the possibilities are endless.

Mixins

Mixins allow you to write Sass code once and reuse it wherever you want in your source file.

The basics

You declare the code with `@mixin` and use it with `@include` (don't confuse it with `@import`).

a-very-simple-mixin-example.scss

```scss
@mixin apply-green-border {
  border: 1px solid green;
  border-radius: 6px;
}

blockquote {
  @include apply-green-border;
}
```

will produce:

a-very-simple-mixin-example.css

```
blockquote {
  border: 1px solid green;
  border-radius: 6px;
}
```

This is useful to avoid to repeat styles. You just have to code them once, then you `@include` them.

Add Flexibility With Variables

Copying a defined style is great, but you can also use variables in mixins. Let's rewrite our simple example to something a little bit more flexible.

use-variables-in-mixins.scss

```
@mixin apply-border($color, $radius) {
  border: 1px solid $color;
  border-radius: $radius;
}
```

```
blockquote {
  @include apply-border(green, 6px);
}
```

This code will produce the exact same CSS as before.

When calling the mixin through `@include`, we just pass arguments `(green, 6px)` to the mixin. The value `green` and `6px` will be stored respectively in the variable `$color` and `$radius` and used in place within the mixin.

Instead of passing `green` and `6px` values, we can use global variables instead:

use-global-variables-in-mixins.scss

```
$color-border: green;
$size-radius: 6px;

@mixin apply-border($color, $radius) {
  border: 1px solid $color;
  border-radius: $radius;
}
```

```
blockquote {
  @include apply-border($color-border, $size-radius);
}
```

This will still produce the same CSS output. The difference is that you have now a more generic mixin that can be used to modify slightly the type of border you need.

You can even imagine a dedicated mixin to create custom borders like this:

custom-borders-mixin-generator.scss

```
@mixin apply-border($size-border, $type-border, $color,
$radius) {
  border: $size-border $type-border $color;
  border-radius: $radius;
}

blockquote {
  @include apply-border(2px, dotted, blue, 8px);
}
```

Nested Rules and Special & Character

Mixins are even more useful while using nested rules and the special & character (parent selector).

You can apply some very advanced styles and layouts using only mixins.

nested-rules-and-parent-selector-in-mixins.scss

```scss
@mixin blockquote-model($color) {
  color: #ff0000;
  border: 1px solid $color;

  // Use special & char and nested rules
  &:before {
    display: block;
    content: '"';
    font-size: 4rem;
  }

  p {
    margin-left: 2rem;
  }
}
```

```
blockquote {
  @include blockquote-model(green);
}
```

will produce several new rules:

nested-rules-and-parent-selector-in-mixins.css

```
blockquote {
  color: #ff0000;
  border: 1px solid green;
}

blockquote:before {
  display: block;
  content: '"';
  font-size: 12px;
}

blockquote p {
  margin-left: 2rem;
}
```

Declaring Entire CSS Rules

For now you always used mixins within a CSS rule. But you can also use `@include` to declare entirely new CSS rule via a mixin:

declare-full-css-rule-via-mixin.scss

```scss
@mixin awesome-p {
  p {
    font-size: 2rem;
    color: #FF9800;
    margin: 2rem;
  }
}

@include awesome-p;
```

will simply produce:

declare-full-css-rule-via-mixin.css

```
p {
  font-size: 2rem;
  color: #FF9800;
  margin: 2rem;
}
```

Include Mixins in Other Mixins

You can include a mixin into another mixin, making compounds easy.

simple-mixin-within-another-mixin.scss

```
@mixin text-blue { color: blue; }
@mixin background-grey { background-color: grey; }

@mixin info-style {
  @include text-blue;
  @include background-grey;
}
```

```
div#info {
  @include info-style;
}
```

will produce:

simple-mixin-within-another-mixin.css

```
div#info {
  color: blue;
  background-color: grey;
}
```

You can even include a mixin within itself since Sass version 3.3. I didn't find any practical use for this yet though.

Mixins Arguments Default Values

You can set a default value to the arguments passed in mixins. Just use "`$arg-name: <default-value>`" when declaring the mixin like this:

using-default-values-in-mixins.scss

```scss
@mixin apply-border($thickness: 1px, $color: red) {
  border: $thickness solid $color;
}

div {
  @include apply-border;
  // You’ll get the same result with parenthesis:
  // @include apply-border();
}
```

If the `$thickness` and `$color` are not set when including the mixin, then the default values will be used and it will produce:

using-default-values-in-mixins.css

```css
div {
  border: 1px solid red;
}
```

Now imagine you want to use this mixin by setting only the `$color` value, how can you do that?

Sass allows you to explicitly set specific argument when calling the mixin by mentioning the argument name first:

```
@include apply-border($color: blue);
```

Let's see some possible combinations and their results:

explicit-arguments-calls-combinations.scss

```
// Mixin declaration is:
// @mixin apply-border($thickness: 1px, $color: red)

div {
  @include apply-border;
  @include apply-border();
  // These 2 above will produce: border: 1px solid red;

  @include apply-border($color: blue);
  // Will produce: border: 1px solid blue;
```

```
    @include apply-border($color: blue, $thickness: 3px);
    // border: 3px solid blue;

    @include apply-border(5px);
    // border: 5px solid red;
  }
```

Using Arguments List

Let’s say you want to use a mixin to manage the padding top, right, bottom and left values.

You could write it like this:

mixin-paddings.scss

```
@mixin paddings($top, $right, $bottom, $left) {
  padding: $top, $right, $bottom, $left;
}
```

Pretty straightforward. The issue with this declaration is that you have to provide the 4 values each time. You can’t use the CSS padding shorthand with 2 values (that sets top and bottom paddings the same and left and right the same). You can't use the 1 value shorthand that sets all the paddings to the same value either.

To get back this flexibility you can use an arguments list (also called variable arguments). Here is how it works:

arguments-list-paddings-mixin.scss

```scss
// Declare arguments list parameter thanks to '...'
@mixin paddings ($padding-list...) {
  padding: $padding-list;
}

.shorthand-2-values { @include paddings (1em, 2em); }
// top & bottom are 1em, left and right: 2em

.shorthand-1-value { @include paddings (3em); }
// top, right, bottom and left are 3em
```

You must use the arguments list parameter at the end of your declaration.

Thanks to this feature, the previous source file will produce this CSS:

arguments-list-paddings-mixin.css

```css
.shorthand-2-values {
  padding: 1em, 2em;
}

.shorthand-1-value {
  padding: 3em;
}
```

The padding example was a simple example to demonstrate how it works. Let’s see a more useful example with a multiple box-shadow list separated by a comma.

my-complex-multiple-box-shadow-generator.scss

```scss
@mixin box-shadow($shadows...) {
  box-shadow: $shadows;
}
```

```
// Include multiple box-shadow values to create a
// more complex layout
.shadows {
  @include box-shadow(0 10px 0 -5px #be6700,
                      0 20px 0 -10px #66ccff,
                      0 30px 0 -16px #dedcb9);
}
```

will produce:

my-complex-multiple-box-shadow-generator.css

```
.shadows {
  box-shadow: 0 10px 0 -5px #be6700,
              0 20px 0 -10px #66ccff,
              0 30px 0 -16px #dedcb9;
}
```

You can spice things up using some variables to achieve a more flexible box-shadow generator. For example nothing prevents you to also set a border-radius like this:

rounded-multiple-box-shadow-generator.scss

```scss
@mixin box-shadow($border-radius, $shadows...) {
  border-radius: $border-radius;
  box-shadow: $shadows;
}

.rounded-box-with-multiple-shadows {
  @include box-shadow(6px,
                      0 10px 0 -5px #be6700,
                      0 20px 0 -10px #66ccff,
                      0 30px 0 -16px #dedcb9);
}
```

will produce:

rounded-multiple-box-shadow-generator.css

```css
.rounded-box-with-multiple-shadows {
  border-radius: 6px;
  box-shadow: 0 10px 0 -5px #be6700, 0 20px 0 -10px
#66ccff, 0 30px 0 -16px #dedcb9; }
```

Inheritance

Using inheritance allows you to keep your source files short and clean by applying the DRY principle. Indeed it helps you declare a CSS rule and reuse it by extending it through the `@extend` keyword. Let's see an example.

inheritance.scss

```scss
.message {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

.success {
  @extend .message;
  border-color: green;
}
```

will produce:

inheritance.css

```css
.message, .success {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

.success {
  border-color: green;
}
```

By using `@extend`, we want the `.success` class to extend from the `.message` one. This means that the `.success` class will get all the CSS properties from `.message` and add new ones on top of it: `border-color: green` in this example.

Using inheritance helps also to avoid using several classes in your HTML code within the `class` attribute.

For example you find this kind of HTML quite often nowadays:

multiple-classes-for-success-message.html

```
<div class="message success">Success!</div>
```

The author of this code wants to create a success message. So he has to use both classes: message first (the base class to display a message) and then style it for success with the success class.

Wouldn’t it be better if you could simply use `class=“success”` only? Thanks to inheritance, you can! Our previous example would let this author use simply `class=“success”` and it would have worked the same way.

Now you may wonder about 2 things:

- I could do the same with a mixin
- I could simply write the CSS for this instead of using `@extend`

And you would be right… But!

Using a mixin instead of inheritance would duplicate your code, making it bigger and that’s not good for web performances.

Always use inheritance over mixin when possible to reduce your website content size.

Also writing the CSS yourself may lead to errors and the waste of countless hours looking for the selectors you need to modify to make things work. Just adding `@extend` within the rule you work on is far more convenient.

Last but not least, using `@extend` will automatically manage additions and modifications related to the selector you inherit from.

Let’s see a more advanced example so you can understand the real power of inheritance and `@extend`.

Let’s start with the first example we used with a new rule for hovering:

advanced-example-inheritance.scss

```scss
.message {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

// The new rule you want to add
.message:hover {
  background-color: blue;
}

.success {
  @extend .message;
  border-color: green;
}
```

Using inheritance allows you to add new rules related to `.message` and make all other rules that extend it automatically inherit these modifications!

You don’t have to keep track of your modifications, Sass will manage them for you.

advanced-example-inheritance.css

```
.message, .success {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

.message:hover, .success:hover {
  background-color: blue;
}

.success {
  border-color: green;
}
```

I used only classes in examples to simplify them but inheritance works with any type of CSS selector.

Moreover, you can extend a rule using several different selectors (multiple extends). You can extend selectors that already inherit from other selectors (chaining extends). Learn more about this here.

Placeholder Selectors

Sass comes with a special selector called a placeholder selector. A placeholder selector tells Sass to not generate the corresponding CSS for that selector.

That nice little feature will help you keep your CSS leaner and avoid names collisions. You remember that thanks to inheritance we've been able to get rid of our `.message` class from the HTML code?

This `.message` class doesn't have to be in your CSS anymore. But on the other hand you need to keep it so the `.success` class can inherit from it. How do you do this? Using a placeholder selector like this:

placeholder-selector.scss

```
%message {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}
```

```
.success {
  @extend %message;
  border-color: green;
}

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

will produce:

placeholder-selector.css

```
.success, .info {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

.success {
  border-color: green;
}
```

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

As you can see the `%message` selector disappears from the CSS but you can still use the code declared within `%message` to extend other classes `.success` and `.info`.

This is very useful to be able to write blocks of CSS rules and reuse them without worrying about classes names collisions.

Using placeholders selectors can let you create classes quickly based on existing code while keeping your CSS clean and small.

To declare a placeholder selector, you need to add a '%' symbol in your selector followed by a name. You can construct more advanced selectors than the simple ones we saw:

advanced-placeholder-selector.scss

```scss
// This won't be present in the CSS because it contains
// a placeholder selector
div %message-action #action {
  font-weight: bold;
  color: red;
}

// This won’t be present in the CSS either, same reason
%message {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}

// Use several placeholder selectors
.success {
  @extend %message;
  @extend %message-action;
}
```

will produce:

advanced-placeholder-selector.css

```css
div .success #action {
  font-weight: bold;
  color: red;
}

.success {
  border: 1px solid #C69;
  padding: 1em;
  color: #EEE;
}
```

The `.success` class name is used at the exact place of the placeholder selector.

Now that we've seen a lot about managing rules properties and sharing them across your source files, let's talk about numbers and properties values.

Operators

Sass lets you use mathematical operators like `+`, `-`, `*`, `/` and `% (modulo)` to compute property values easily.

Instead of talking about theory, I'll simply show you several examples of how to use operators. You'll easily deduct how they work:

simple-example-operators.scss

```
div {
  // Correct                        // CSS output
  width: 100px + 25px;              // width: 125px;
  width: 100px - 25;                // width: 75px;
  width: 10 * 25px;                 // width: 250px;
  width: 3.2rem + 2rem;             // width: 5.2rem;
  width: 1em + .8;                  // width: 1.8em;
  width: 5 + 12%;                   // width: 17%;
  line-height: 3 * .5;              // line-height: 1.5;

  // Incorrect: you can't mix incompatible units
  width: 10em + 25px;
  width: 20px + 12%;
```

```
    // Incorrect: the CSS output has no meaning
    width: 100 + 25;             // width: 125;
    font-size: (100 / 4);        // font-size: 25;

    // Incorrect: produces invalid (squared) CSS units
    width: 4px * 20px;
    width: 3em * 2em;
  }
```

Enforcing Divisions

The division symbol may be used in CSS to separate values, for example:

```
// There is no division here, it's pure CSS code
// This line sets font-size: 20px, line-height: 30px
font: 20px/30px helvetica;
```

Sass will detect it and won't compute a division. But sometimes, you need to enforce a division. To do so, you must use parenthesis and take care of the unit. Let's see what works and what doesn't:

enforcing-division.scss

```scss
div {
  // Correct: use parenthesis to enforce division
  width: (100px/4);           // width: 25px;

  // Incorrect use of parenthesis
  width: ((100/4)px);  // These 2 properties produce:
  width: (100/4)px;    // width: 25 px; There is a
                       // space between ‘25’ and ‘px’

  width: (100/4px);    // Trying to divide unitless
                       // value by pixels

  // Incorrect: produce invalid CSS values
  width: 100px/4;             // width: 100px/4;
  width: 100/4px;             // width: 100/4px;
  width: 100px/4px;           // width: 100px/4px;
  width: (100px/4px);         // width: 25;
}
```

Some special cases exist. When you use Sass variables or other operators, you don’t need to use parenthesis:

automatically-enforced-division.scss

```scss
div {
  $container-width: 400px;
  $padding: 20px;
  width: $container-width / 2;
  height: $padding * 2 + 100px / 2;
}
```

will produce:

automatically-enforced-division.css

```css
div {
  width: 200px;
  height: 90px;
}
```

Preventing Divisions

If you want to use variables with the CSS value separation symbol `/`, use interpolation:

use-interpolation-to-avoid-division.scss

```scss
$default-font-size: 1.8em;
div {
  font: #{$default-font-size}/1.5;
}
```

will produce what you want:

use-interpolation-to-avoid-division.css

```css
div {
  font: 1.8em/1.5;
}
```

What’s next?

This book covers Sass basics so you can be up and running quickly. But if you’re looking for a front-end developer job, you’ll need to read the Sass intermediate book I’m writing (link coming soon).

Support

This book is free but I put lots of hours into it. If you learned a lot from it and enjoyed reading it, please consider supporting my work by purchasing this book for whatever price you like.

Thanks for your support.

Feedback

I definitely want to improve this book over time so I would love to hear your feedback to improve it!

How can I improve this book? Do you have suggestions? Were some parts harder than others to understand?

Please don’t be shy and take the time to send me a little email so I can improve this book to fit your needs.

Email me at: jeremy.mouzin@gmail.com, use email subject “Sass ebook”.

Thanks!