# 12 Using preprocessors

This chapter covers

- CSS preprocessors
- Examples of how Sass extends CSS functionality

So far in this book, we've been writing all our styles using plain CSS. We can also use preprocessors, however. Each processor has its own syntax, and most preprocessors extend the existing CSS functionality. The most commonly used are

- Sass (**https://sass-lang.com**)
- Less (**https://lesscss.org**)
- Stylus (**https://stylus-lang.com**)

They were created to facilitate writing code that's easier to read and maintain as well as to add functionality that's not available in CSS. Styles written for use with preprocessors

have their own syntax and must be built or compiled into CSS. Although some preprocessors provide browser-side compilation, the most common implementation is to preprocess the styles and serve the output CSS to the browser (**http://mng.bz/Wzex**).

The benefit of using a preprocessor is the added functionality it provides, examples of which we cover in this chapter. The drawback is that now we need a build step for our code. The choice of preprocessor is based on what functionality is needed for the project, the team's knowledge, and (if the project uses a framework) which frameworks are supported. For our project, we're going to choose based on popularity. When developers were surveyed about their sentiments regarding CSS preprocessors, the majority favored Sass (figure 12.1), so that's what we're going to use.

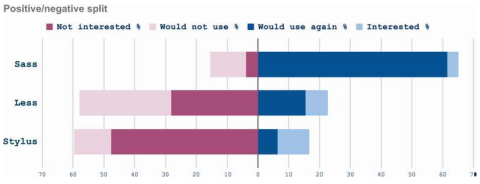


Figure 12.1 Preprocessor sentiment (data source **http://mng.bz/8ry2**)

## 2.1 Running the preprocessor

Our project consists of styling a how-to article—something we might see in a wiki or documentation (figure 12.2).


Figure 12.2 Finished project

As in earlier chapters, the starting code is available at GitHub (**http://mng.bz/EQnl**) and CodePen (**https://codepen.io/michaelgearon/pen/WNpNoGN**). But running the project is going to be a little bit different. Because we're going to write our styles with Sass, which

outputs the CSS rather than writing it directly, we'll need a build step. To run this project and code along with this chapter, you have two options:

- npm
- CodePen

**NOTE** *npm* (Node.js package manager) is a software library, manager, and installer. If you aren't familiar with npm, that's OK. You can run this project in CodePen, following the instructions in section 12.1.3.

### 12.1.1 Setup instructions for npm

Via the command line from the `chapter-12` directory, install the dependencies using `npm install`; then start the processor using `npm start`. This command starts a watcher that will monitor changes in `styles.scss` (in the `before` and `after` directories) and output the `styles.css` and `styles.map.css` files.

The second file—`styles.map.css`—is a source map. Because the CSS was generated from another language,

the source map allows the browser's developer tools to tell us where the piece of code originated in the preprocessed file (for this project, `styles.scss`).

### 12.1.2 .sass versus .scss

Although we're using Sass, our file extension is `.scss`. Sass has two syntaxes we can choose—indented and SCSS—and the file extension reflects the syntax.

#### Indented syntax

Sometimes referred to as *Sass syntax, indented syntax* uses the `.sass` file extension. When writing rulesets using this syntax, we omit curly braces and semicolons, using tabs to describe the format of the document. The following listing shows two rules using indented syntax, the first handling margin and padding on the body text and the second changing the line height of the paragraphs.

Listing 12.1 Sass using indented syntax

```
body
     margin: 0
```

```
      padding: 20px

p
      line-height: 1.5
```

### SCSS SYNTAX

The second syntax is SCSS, which uses the file extension `.scss`. We'll use that syntax in this project. *SCSS syntax* is a superset of CSS that allows us to use any valid CSS in addition to Sass features. The following listing shows the rules from listing 12.1 in SCSS syntax.

Listing 12.2 Sass using SCSS syntax

```
body {
  margin: 0;
  padding: 20px;
}

p {
  line-height: 1.5;
}
```

The code looks like CSS, which is exactly the point. In SCSS, we can write CSS the way we're used to writing it and have access to all the functionality Sass provides as well. Because of its similarity to CSS, and because it doesn't require

developers to learn a new syntax, SCSS is the more popular of the two syntax options.

### 12.1.3 Setup instructions for CodePen

To set up the project for CodePen, follow these steps:

1. Go to **https://codepen.io**.
2. In a new pen, using the code in the `chapter-12/before` folder, copy the HTML inside the `body` element to the HTML panel.
3. Copy the starting styles in the `.scss` file to the CSS panel.
4. To make the panel use Sass with SCSS syntax instead of CSS, click the gear in the top-right corner of the CSS panel (figure 12.3).

Figure 12.3 Settings button

5. Choose SCSS from the CSS Preprocessor drop-down menu (figure 12.4).

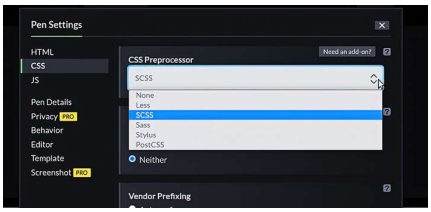


Figure 12.4 CodePen CSS preprocessor settings

6. Click the green Save & Close button at the bottom of the Pen Settings dialog box.

### 12.1.4 Starting HTML and SCSS

Our project is composed of headers, paragraphs, links, and images (listing 12.3). Notice that in our head, we reference the CSS stylesheet, not the SCSS. The browser uses the compiled version.

Listing 12.3 Starting HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Chapter 12: Pre-processors | Tiny CSS Projects</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="styles.css">                    ①
</head>
```

```
  <body>
    <h1>Keeping it Sassy</h1>
    <h2>Step 1</h2>
    <img src="https:/ /bit.ly/3VUzJ7g" alt="blue print">
    <p>
      Lorem ipsum dolor sit amet...
      <a href="">tincidunt purus</a>
      eu, gravida enim. Vestibulum...
    </p>
    <p class="success">You did it!</p>                          ②
    <p>Ut maximus id erat et mollis...</p>
    <h2>Step 2</h2>
    <img src="https:/ /bit.ly/3F4vd0f" alt="crane">
    <p>Aenean non lorem tincidunt...</p>
    <p class="warning">Don't press the big red button</p>       ③
    <p>
      Proin pharetra, urna et sagittis lacinia...
      <a href="">orci luctus</a>
      et ultrices posuere cubilia curae...
    </p>
    <h2>Step 3</h2>
    <img src="https:/ /bit.ly/3N42oD1" alt="wrong way">
    <p>Nullam ut auctor nisi...</p>
    <p class="error">Mistakes have been made</p>                ④
    <p>Vestibulum interdum eleifend...</p>
  </body>

  </html>
```

① Links to the processed CSS file

② Green success callout

③ Orange warning callout

④ Red error callout

Our starting styles set up our typography and constrain the content's width when the page

gets wide, as shown in the following listing.

Listing 12.4 Starting SCSS

```
@import url('https:/ /fonts.googleapis.com/css2?
➥ family=Nunito:wght@300;400;500;800&display=swap');

body {
  font-family: 'Nunito', sans-serif;
  font-weight: 300;
  max-width: 72ch;
  margin: 2rem auto;
}

p { line-height: 1.5 }
```

So far, we're not using any of the extended functionality that Sass provides. As a matter of fact, if we look at the CSS output (listing 12.5), we notice that the file contents are the same except for the map reference at the bottom of the file. This comment tells the browser where to find the source map. Figure 12.5 shows our starting point.

## Keeping it Sassy

### Step 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed porta erat nec ipsum volutpat ultrices. Pellentesque ac mi lobortis, tincidunt purus eu, gravida enim. Vestibulum pharetra a arcu ac suscipit. Ut et lorem dui. Donec non vehicula orci. Nunc non ornare mi, ac aliquam risus.

You did it!

Ut maximus id erat et mollis. Aenean sit amet fringilla augue. Donec convallis vel nibh vitae porttitor. Phasellus elementum nibh at erat semper consectetur. Praesent convallis iaculis mauris, sit amet egestas nunc gravida in. Donec dapibus mattis nibh, sed iaculis libero blandit et.

### Step 2

Aenean non lorem tincidunt, vulputate nibh et, convallis felis. Donec at tristique sem. Aenean id leo non lectus hendrerit sodales. Maecenas vulputate scelerisque dignissim. Integer purus nisl, blandit in odio a, gravida interdum velit. Etiam consectetur risus ante, vel pulvinar felis eleifend ut. Phasellus nec tellus vitae sem semper ultrices at et ligula.

Don't press the big red button.

Proin pharetra, urna et sagittis lacinia, quam metus vulputate eros, ac congue quam leo suscipit est. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Vestibulum nec suscipit ipsum. Vestibulum dapibus, neque vel lacinia mattis, magna sapien hendrerit justo, sed laoreet sapien enim quis mauris.

### Step 3


Nullam ut auctor nisi. Vestibulum pretium vitae erat et hendrerit. Donec velit ipsum, fringilla sed aliquam non, tincidunt a mauris. Mauris sit amet diam lacus. Donec gravida felis nec ligula ultrices, et molestie tellus tristique.

Mistakes have been made.

Vestibulum interdum eleifend suscipit. Nullam imperdiet dignissim nulla, et mattis erat dignissim ut. Proin dui felis, venenatis sit amet lacus at, commodo elementum dolor. Vestibulum et justo eu est pharetra pulvinar. Duis fermentum iaculis velit, in hendrerit metus efficitur vel. Fusce vitae mollis nisl. Fusce eu viverra erat. Vivamus nunc risus, consectetur at eros ac, bibendum viverra massa. Aliquam metus lacus, condimentum in ligula eget, molestie faucibus odio. Integer eros tellus, tristique non elementum eget, congue scelerisque quam.

Figure 12.5 Starting point

**NOTE** If you're using CodePen, you can view the compiled CSS by clicking the down arrow next to the gear in the top-right corner of the CSS panel (refer to figure 12.3) and choosing View Compiled CSS from the drop-down menu.

Listing 12.5 Starting CSS output

```
@import url("https:/ /fonts.googleapis.com/css2?
➥ family=Nunito:wght@300;400;500;800&display=swap");
body {
  font-family: "Nunito", sans-serif;
  font-weight: 300;
  max-width: 72ch;
  margin: 2rem auto;
}

p {
  line-height: 1.5;
}

/*# sourceMappingURL=styles.css.map */      ①
```

① Source map reference

**NOTE** If you aren't seeing the CSS file being created and styles being applied, make sure that you're running the Sass watcher ( `npm start` ). When the watcher starts, let it run in the background; it updates the CSS file automatically when you save your changes in the SCSS file. You'll

still need to refresh the browser manually.

## 2.2 Sass variables

One reason why preprocessors became popular early on is that they had variables before browsers supported custom properties. Sass variables are quite distinct from CSS custom properties in that they have different syntax and function differently. Let's first look at the syntax. To create a variable, we start with a dollar sign ( `$` ) followed by the variable name, a colon ( `:` ), and then a value (figure 12.6).

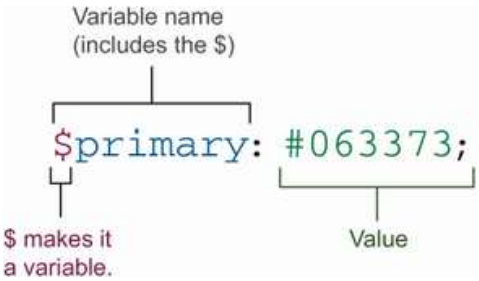


Figure 12.6 Sass variable syntax

In terms of functionality, Sass variables aren't aware of the Document Object Model (DOM) and don't understand cascading or inheritance. They're block-scoped: only properties within the curly braces they're defined in know about their existence.

Therefore, the scenario presented in the following listing would throw an undefined variable error at compile time because the variable is defined and used in two different rules or blocks.

Listing 12.6 `$myColor` variable undefined in second rule

```
body {
  $myColor: blue;                ①
}

body p {
  /* $myColor is undefined */
  color: $myColor                ②
}
```

① Defines the $myColor variable inside the body rule

② $myColor is undefined because it was created inside a different rule.

To prevent this problem, we can place our variables outside a rule, which would make them available to the entire document, as shown in the following listing.

Listing 12.7 Defining variables

```
$myColor: blue;        ①
```

```
body p {
  color: $myColor;    ②
}
```

① Defines the $myColor variable outside any ruleset

② $myColor is now defined and has a value of blue.

Unlike custom properties, which are dynamic, Sass variables are static. If we define a variable, use it, change its value, and then use it again, any property it was assigned to before the change will retain the original value, and those assigned after the change will have the new value. The examples shown in listings 12.8 and 12.9 make this situation a bit clearer. Note that the examples aren't part of our project; we present them here only to illustrate the concept. You can find the code on CodePen at **https://codepen.io/martine-dowden/pen/QWxLjWy**.

Listing 12.8 Custom properties versus variables (HTML)

```
<p class="first">My first paragraph</p>
<p class="second">My second paragraph</p>
```

Listing 12.9 Custom properties versus variables (SCSS)

```
body { --myBorder: solid 1px gray;  }        ①
$primary: red;                               ②

.first {                                     ③
  color: $primary;                           ③
  border: var(--myBorder);                   ③
}                                            ③

body { --myBorder: dashed 1px purple;  }     ④
$primary: blue;                              ⑤

.second {                                    ⑥
  color: $primary;                           ⑥
  border: var(--myBorder);                   ⑥
}                                            ⑥
```

① Assigns the --myBorder custom property a solid gray border

② Assigns the color red to our $primary variable

③ Applies the --myBorder custom property and $primary variable to the color and border properties

④ Changes the --myBorder custom property value to a dashed purple border

⑤ Changes the $primary value to the color blue

⑥ Applies the --myBorder and $primary to the second paragraph

The first big difference between the custom properties and the variables is that we aren't required to have our variables inside a rule. Also, the border styles of both paragraphs are the same, but the color of the text is not (figure 12.7), even though both the custom property and the variable were reassigned between the first and second rule.


Figure 12.7 Example output

When we reassign the value of the custom property (the border), it's applied everywhere, whereas the color doesn't change retroactively; only the rule after the change is affected. The reason is that custom properties are dynamic and variables are static.

With this understanding, let's get back to our project and define some variables for the colors we'll use. At the top of the file, we'll define four color variables. Then we'll apply the

primary color to all our headers, as shown in the following listing.

Listing 12.10 Color variables (SCSS)

```
@import url('https:/ /fonts.googleapis.com/css2
➥ ?family=Nunito:wght@300;400;500;800&display=swap');

$primary: #063373;                ①
$success: #747d10;                ②
$warning: #fc9d03;                ③
$error: #940a0a;                  ④

p { line-height: 1.5 }

h1, h2 { color: $primary; }       ⑤
```

① Blue

② Green

③ Orange

④ Red

⑤ Makes our headers blue

We place our variables at the beginning of our file and outside any rule, so that from that point on and inside any rule, we can have access to them. We notice in our CSS output (listing 12.11) that our variables aren't visible in the compiled CSS. But in the rule defining our header color, the

place where we used one of our variables has been replaced by its value.

Listing 12.11 Heading-color CSS output

```
@import url("https:/ /fonts.googleapis.com/css2
➥ ?family=Nunito:wght@300;400;500;800&display=swap");
body {
  font-family: "Nunito", sans-serif;
  font-weight: 300;
  max-width: 72ch;
  margin: 2rem auto;
}

p {
  line-height: 1.5;
}

h1, h2 {
  color: #063373;
}

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

Now our project headers look like figure 12.8. Let's style our images next.


Figure 12.8 Updated header color

### 12.2.1 @extend

Sass gives us several new at-rules, two of which are `@extend` and `@include`. These rules allow us to build generic classes that we can reuse throughout our code. One way we can reuse classes in CSS is to have multiple selectors for a single rule, as we did when we styled our headers. Instead of creating two identical rules for each header (`<h1>` and `<h2>`), we created one rule and gave it two selectors: `h1, h2 { }`.

`@extend` allows us to create a base rule that we can point to from a different rule later. Then the selector will be added to the base rule's list of selectors. Let's use this technique to style our images and see it at work.

First, we create the base rule that will define the `height`, `width`, `object-fit`, and `margin` for our image. Because we have three images, and because we want to give each image a slightly different border radius and positioning, we point each image individually back to our `base-`

`image` rule. The following listing shows how.

Listing 12.12 Extending image styles (SCSS)

```
.image-base {                                    ①
  width: 300px;                                  ①
  height: 300px;                                 ①
  object-fit: cover;                             ①
  margin: 0 2rem;                                ①
}                                                ①

img:first-of-type { @extend .image-base; }       ②
img:nth-of-type(2) { @extend .image-base; }      ②
img:last-of-type { @extend .image-base; }        ②
```

① Base rule

② Images extending the base rule

The following listing shows the CSS output.

Listing 12.13 Extending image styles (CSS output)

```
.image-base, img:last-of-type, img:nth-of-type(2), img:first-of-type {
  width: 300px;
  height: 300px;
  object-fit: cover;
  margin: 0 2rem;
}
```

By creating a base rule and then using `@extend`, we can create some defaults and apply them to any other selector

without duplicating our CSS code. We can also keep all our code related to a selector in one rule. With our default image styles applied (figure 12.9), let's customize them individually.


Figure 12.9 Base image styles

## 2.3 @mixin and @include

We want to customize each image's `border-radius`, `position`, and `object-position`. To do this, we're going to use a mixin. *Mixins* allow us to generate declarations and rules. Like functions, they take parameters (although they're not mandatory) and return styles. Let's write one that will return our three declarations for each image. A mixin is an at-rule, so it starts with `@mixin` followed by the name we want to give it. Next, we add parentheses with any parameters we want to pass in. Finally, we add a set of curly braces, inside which we define the styles we want the mixin to return. Figure 12.10 shows the syntax.

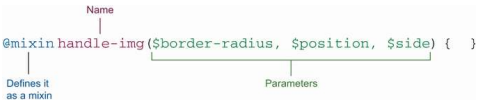


Figure 12.10 Mixin syntax

Notice that each parameter starts with a dollar sign. In Sass, the name of the parameter is defined the same way as a variable starting with `$`.

Inside the mixin, we assign these parameter values to properties, as shown in listing 12.14. We alter the border radius, float the image, and remove the margin on the side it is being floated to. Note that the mixin needs to be defined before it can be used, so it's common to place mixins at the beginning of the file.

Listing 12.14 Building the mixin (SCSS)

```
@mixin handle-img($border-radius, $position, $side) {
  border-radius: $border-radius;
  object-position: $position;
  float: $side;
  margin-#{$side}: 0;    ①
}
```

① Interpolation (section 12.3.2)

At this point, we don't see a change in the project. We've defined the mixin but haven't used it yet. Before we apply it, let's take a closer look at some of its properties.

### 12.3.1 object-fit property

In our base rule, we set our `object-fit` property value to `cover`. The `object-position` property, which we also use in

our mixin, works hand in hand with `object-fit` and determines the alignment of the image within its bounding box. Remember that `cover` makes the browser calculate the optimum size of the image based on the dimensions provided so that as much of the image that can be shown without distortion appears.

If the dimensions provided to the image don't have the same aspect ratio as the image, the excess is clipped. `object-position` changes where the image is positioned inside the container, allowing us to manipulate which part of the image is clipped when the ratios don't match (figure 12.11).

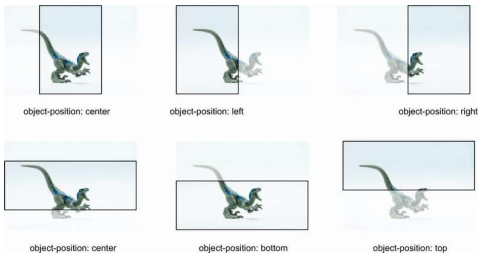


Figure 12.11 Visible vs. clipped portions of the image when using `object-position` in conjunction with `object-fit: cover`

### 12.3.2 Interpolation

Notice the syntax for the margin: `margin-#{$side}: 0;`. We added a hash ( `#` ) and

curly braces around the variable. This syntax, called *interpolation*, allows us to insert a value into our parameter. It embeds the result of the expression inside the curly braces in our CSS, replacing the hash. If the value of `$side` is equal to `"left"`, for example, our declaration will compile to `margin-left: 0;`.

You may have encountered interpolation in JavaScript in the context of string interpolation in template literals: `` `margin-${side}` ``. In our project, we're trying to concatenate `margin-` and the value of the `$side` variable. Because `'margin-' + $side` isn't a valid property declaration, we use interpolation to insert the value.

### 12.3.3 Using mixins

Next, we're going to use our mixin in each image rule. To do that, we use `@include` followed by the mixin's name and, in parentheses, the parameters it requires (figure 12.12).

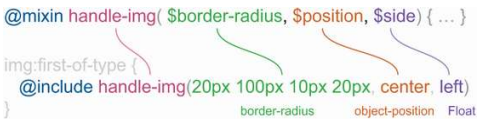


Figure 12.12 `@mixin` syntax

In all three image rules, we use `@include handle-img()` and pass in the `border-radius`, `object-position`, and `float` property values we want to use (listing 12.15). All three images have rounded corners (the first parameter of our mixin). Our first and second image use the `border-radius` shorthand property, which we'll talk about in section 12.3.4.

Listing 12.15 Using the mixin (SCSS)

```
@mixin handle-img($border-radius, $position, $side) {
  border-radius: $border-radius;
  object-position: $position;
  float: $side;
  margin-#{$side}: 0;
}

img:first-of-type {
  @extend .image-base;
  @include handle-img(20px 100px 10px 20px, center, left);
}

img:nth-of-type(2) {
  @extend .image-base;
  @include handle-img(100px 20px 10px 20px, left top, right);
}
```

```
img:last-of-type {
  @extend .image-base;
  @include handle-img(50px, center, left);
}
```

In our output CSS, the mixin itself isn't there, but we have three new rules, one for each image, as shown in the following listing.

Listing 12.16 Using the mixin output (CSS)

```
.image-base, img:last-of-type, img:nth-of-type(2), img:first-of-type {    ①
  width: 300px;                                                           ①
  height: 300px;                                                          ①
  object-fit: cover;                                                      ①
  margin: 0 2rem;                                                         ①
}                                                                         ①

img:first-of-type {                                                       ②
  border-radius: 20px 100px 10px 20px;                                    ②
  object-position: center;                                                ②
  float: left;                                                            ②
  margin-left: 0;                                                         ②
}                                                                         ②

img:nth-of-type(2) {                                                      ②
  border-radius: 100px 20px 10px 20px;                                    ②
  object-position: left top;                                              ②
  float: right;                                                           ②
  margin-right: 0;                                                        ②
}                                                                         ②
img:last-of-type {                                                        ②
  border-radius: 50px;                                                    ②
  object-position: center;                                                ②
  float: left;                                                            ②
  margin-left: 0;                                                         ②
}                                                                         ②
```

① Selectors added to the base class by using @extend

② Generated by using the mixin (@include)

This output exposes the difference between using `@extend` and using a mixin ( `@include` ). When we extend a rule, Sass doesn't copy or generate code; it only adds the selector to the base. When we use a mixin, Sass generates code. If we're setting properties dynamically, we want to use a mixin. But if the property values are static, we want to extend; otherwise, we'd be copying those values every time we used the mixin, bloating our stylesheet. At this point, our project looks like figure 12.13.


Figure 12.13 Styled images

### 12.3.4 border-radius shorthand

For our first and second images, we're using the `border-radius` shorthand. The first image's generated CSS has a `border-radius` property value of `20px 100px 10px 20px`. Just as we set different padding values for all four sides of an element in one declaration, `border-radius` allows us to use a similar syntax (figure 12.14). Each value defines the radius of the corner starting at top left and rotating clockwise.

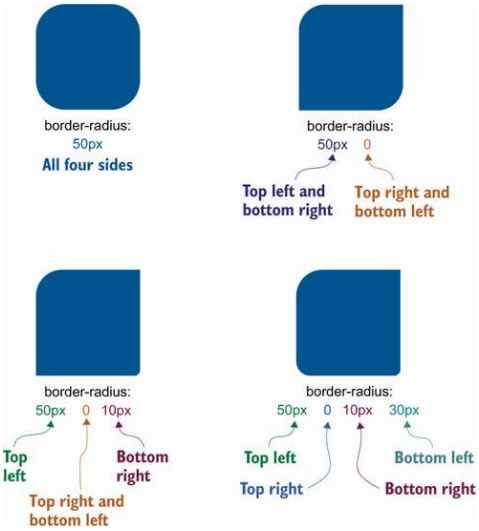


Figure 12.14 The `border-radius` property

Now that our images are styled, let's take a closer look at our text. In some paragraphs, we have links to style.

## 2.4 Nesting

One cool thing that Sass lets us do is nest rules. When we style links, we often write several rules so that we can handle the various states (link, visited, hover, focus, and so on). We can nest them together as shown in listing 12.17. Nesting our rules clearly shows the ancestor–descendant relationships in our code and keeps our rules grouped and organized.

To select the parent selector, we use an ampersand ( `&` ). In our rule, the parent rule is for

the anchor element. Inside this rule, we need to reference the parent ( `a` ) to use with the `:link`, `:visited`, `:hover`, and `:focus` pseudo-classes, so we precede them with `&`.

We make all our anchor elements bold, make them blue by using our `$primary` variable, and edit the underline of our links from solid to dotted. On hover, we make the underline a dashed line. Finally, we make the focus underline a solid line. On focus, we also remove the default outline that exists in some browsers.

Listing 12.17 Nesting rules (SCSS)

```
a {                                              ①
  font-weight: 800;
  &:link, &:visited {                            ②
    color: $primary;
    text-decoration-style: dotted;               ③
  }
  &:hover { text-decoration-style: dashed;}      ④
  &:focus {                                      ⑤
    text-decoration-style: solid;                ⑥
    outline: none;
  }
}
```

① All anchor elements: the parent

② Anchor elements that contain an href, both visited and not

③ Changes the underline style to a dotted line

④ On link hover, changes the underline style to a dashed line

⑤ On link focus

⑥ Changes the underline style to a solid line

In our CSS output, shown in the following listing, our nested rule has been flattened, creating individual rules for the anchor element and each of its states. Now our links look like figure 12.15.

Listing 12.18 Nesting rules (CSS output)

```
a {
  font-weight: 800;
}
a:link, a:visited {
  color: #063373;
  text-decoration-style: dotted;
}
a:hover {
  text-decoration-style: dashed;
}
a:focus {
  text-decoration-style: solid;
```

```
    outline: none;
  }
```

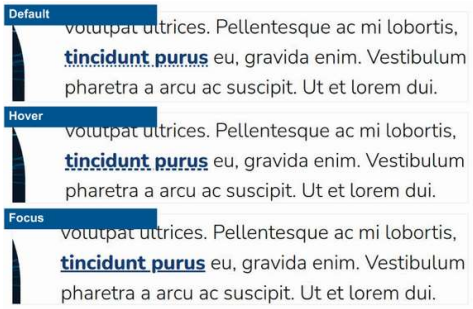


Figure 12.15 Styled links: (top to bottom) default, hover, and focus

**NOTE** Nesting is a great way to keep our rules grouped and organized. But for every level of nesting, there is another level of specificity. In listing 12.17, we nest the hover and focus inside the anchor ( `a` ) rule. The selector in the output (listing 12.18) for the inner rules are more specific than the outer rule: `a:hover` is more specific than `a` . By nesting rules, we can easily end up creating overly specific rules, which decrease performance. We need to be on the lookout for excessive nesting in our code. If we notice that nesting becomes more than three levels deep, we should examine how our rules are nested and see whether some of the rules could be unnested.

With links styled, the next pieces of text we want to turn our attention to are the callout paragraphs.

## 2.5 @each

In our text, we have three callout paragraphs that have classes of `success`, `warning`, and `error`. As we did when we styled our images (section 12.4), we'll create a base rule and then extend it (listing 12.19). The rule defines the `border`, `border-radius`, and `padding` we want our callouts to have, and it includes the styles all three types have in common.

Listing 12.19 Callout base rule

```
.callout {
  border: solid 1px;
  border-radius: 4px;
  padding: .5rem 1rem;
}
```

Next, instead of writing individual rules for each callout type, we're going to create a map, a list of key-value pairs that we can iterate over to generate the rulesets. Because the differentiating factor of our callouts is the color, our

key will be the type, and our value will be the color variable we defined at the beginning of this chapter. Our map, therefore, will be `$callouts: (success: $success, warning: $warning, error: $error);` . Figure 12.16 breaks down the syntax.

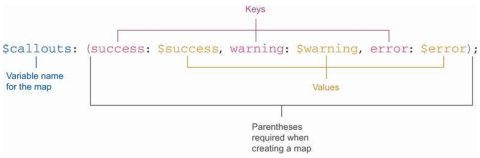


Figure 12.16 Sass map syntax

With the map created, we can loop over each key-value pair to generate our classes. For looping, we'll use `@each` . This at-rule iterates over all the items in a list or map in order, which is perfect for our use case. We'll add the following rule to our SCSS: `@each $type, $color in $callouts {}` . The first variable ( `$type` ) gives us access to the key, the second ( `$color` ) is the value of the key pair, and the last ( `$callouts` ) is the map we want to iterate over. We'll put the code to generate our rules inside the curly braces. To test our loop, we can add an `@debug` declaration inside the curly braces to

check that our variable values are what we expect (listing 12.20).

**NOTE** `@debug` is the Sass equivalent of JavaScript's `console.log()`. It allows us to print values to the terminal. Unfortunately, CodePen doesn't seem to have a way to expose Sass debug statements in its console. These statements won't show up in the browser's console, either. You'll be able to see the debug output only if you're running the project locally.

Listing 12.20 `@debug` statement inside our loop (SCSS)

```
$callouts: (success: $success, warning: $warning, error: $error);     ①
@each $type, $color in $callouts {                                    ②
  @debug $type, $color;                                               ③
}
```

① The map

② Sets up the loop

③ The debug statement that will print our $type and $color values to the terminal

In the terminal where we have our Sass watcher running, the `@debug` statement outputs the file name, line number, the

word *Debug*, and the values for our two variables (listing 12.21). Note that your line numbers may differ slightly from those displayed in the listing.

Listing 12.21 Output in terminal

```
before/styles.scss:70 Debug: success, #747d10       ①
before/styles.scss:70 Debug: warning, #fc9d03       ②
before/styles.scss:70 Debug: error, #940a0a         ③
Compiled before/styles.scss to before/styles.css.
```

① First key-value pair

② Second key-value pair

③ Third key-value pair

Now that we know our loop is working correctly, we can create rules for our callout types. In each ruleset, we extend our `.callout` base rule and add the correct border color for each type by using `border-color`. The value of the `border-color` property is the `$color` variable that comes from our `@each` loop. We mentioned earlier that Sass variables are static (section 12.2). As a result, the `$color` variable's value is reassigned for each key-value pair in the

map, assigning the `border-color` correctly for each callout type.

Next, we add the type name before the paragraph by using the `::before` pseudo element so that we have a visual indicator other than color telling the user what type of callout it is. Because the type value is lowercase in our map, we also use `text-transform` to capitalize it. Listing 12.22 shows our updated loop.

**NOTE** Never use color alone to convey meaning. Some users, such as those who are color-blind, may have difficulty perceiving colors or may not be able to see them at all. In our case, the color conveys the type of callout, so we should include some other indicator (the text).

Listing 12.22 Adding to the loop (SCSS)

```
.callout {
  border: solid 1px;
  border-radius: 4px;
  padding: .5rem 1rem;
}

$callouts: (success: $success, warning: $warning, error: $error);
@each $type, $color in $callouts {
```

```
    @debug $type, $color;
    .#{$type} {                          ①
      @extend .callout;
      border-color: $color;
      &::before {
        content: "#{$type}: ";           ②
        text-transform: capitalize;
      }
    }
  }
```

① Interpolation to create the class name

② Interpolation to get the type name in the content

As we did when we used interpolation to create a margin declaration in section 12.3.2, we use it here to create the class name and add the type to the content. By looping over the map, our `@each` rule creates three rules, one for each type. Each selector also gets added to the `.callout` rule via the `@extend`, as shown in the following listing.

Listing 12.23 Loop CSS output

```
.callout, .error, .warning, .success {      ①
  border: solid 1px;
  border-radius: 4px;
  padding: 0.5rem 1rem;
}

.success {
```

```
  border-color: #747d10;
}
.success::before {
  content: "success: ";
  text-transform: capitalize;
}

.warning {
  border-color: #fc9d03;
}
.warning::before {
  content: "warning: ";
  text-transform: capitalize;
}

.error {
  border-color: #940a0a;
}
.error::before {
  content: "error: ";
  text-transform: capitalize;
}
```

① All three class selectors (.error, .warning, .success) are added to the .callout base class.

Now our three callouts have colored borders (figure 12.17). But we still need to boldface *Error:* in the error callout and add the background colors.

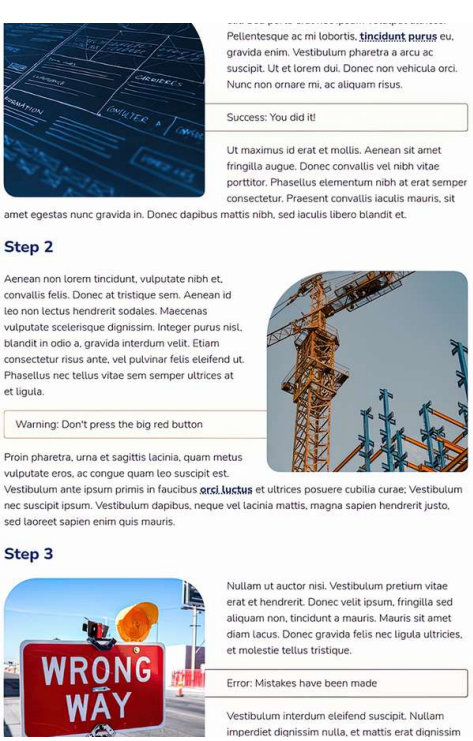


Figure 12.17 Callout styles including colored borders

## 2.6 Color functions

We want the background colors for each callout to be significantly lighter than the colors we currently have stored in our variables. To make working with colors easier, Sass provides functions for manipulating colors. We're going to use `scale-color()`. The `scale-color()` function is incredibly versatile and can be used to change the amount of red, blue, and green in a color; change the saturation or opacity; and make a color lighter or darker (figure 12.18).

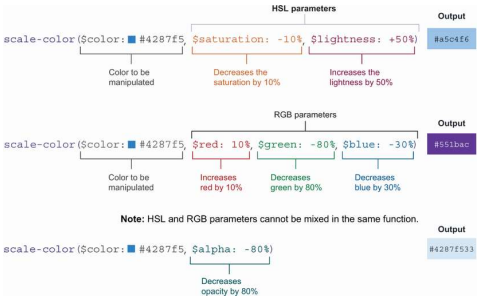


Figure 12.18 The `scale-color()` function

Worth noting is the fact that `scale-color()` operates with either HSL (hue, saturation, and lightness) or RGB (red, green, and blue) parameters; they can't be mixed. The alpha (transparency) parameter, however, can be used with either set of parameters. Also, parameters can be omitted. So if we want to change only the opacity, we need to pass only the initial color and the parameter(s) with which we want to manipulate the color.

For our backgrounds, we need to increase the lightness of the color, so we use HSL parameters. We don't need to change the saturation, so we'll omit the saturation parameter and pass in only the color and the amount by which we want to increase the lightness ( `86%` ), as shown in the following listing.

Listing 12.24 Adding the background color (SCSS)

```
$callouts: (success: $success, warning: $warning, error: $error);
@each $type, $color in $callouts {
  @debug $type, $color;
  .#{$type} {
    @extend .callout;
    background-color: scale-color($color, $lightness: +86%);      ①
    border-color: $color;
    &::before {
      content: "#{$type}: ";
      text-transform: capitalize;
    }
  }
}
```

① Increases the lightness of the color provided in the map by 86%

The following listing shows the color generated by the `scale-color()` function in our CSS output.

Listing 12.25 `scale-color()` function output (CSS)

```
.callout, .error, .warning, .success {
  border: solid 1px;
  border-radius: 4px;
  padding: 0.5rem 1rem;
}

.success {
  background-color: #f6f9d1;
  border-color: #747d10;
}
.success::before {
```

```
    content: "success: ";
    text-transform: capitalize;
  }

  .warning {
    background-color: #fff1dc;
    border-color: #fc9d03;
  }
  .warning::before {
    content: "warning: ";
    text-transform: capitalize;
  }

  .error {
    background-color: #fcd1d1;
    border-color: #940a0a;
  }
  .error::before {
    content: "error: ";
    text-transform: capitalize;
  }
```

Now that we've added the background colors (figure 12.19), all we have left to do is boldface *Error:* as part of the `::before` content for the error callout.

## 2.7 @if and @else

Another set of at-rules that are available thanks to Sass are `@if` and `@else`, which control whether a block of code is evaluated and provide a fallback condition if the condition isn't met. We're going to use them inside our loop to boldface only the contents of the

`::before` pseudo-element if the type of callout is `error` and increase the font weight to medium ( `500` ) for the others.

If you're used to JavaScript, a couple of gotchas can trip you up when evaluating equality in Sass, because Saas doesn't have truthy/falsy behaviors. Values are considered to be equal only if they have the same value and type. Also, Sass doesn't use the double pipe ( `||` ) or double ampersand ( `&&` ) but `or` and `and` for considering multiple conditions. The following listing shows examples of some of Sass's equality operators and what they resolve to.

Listing 12.26 Equalities (SCSS)

```
@debug '' == false;              // false    ①
@debug 'true' == true;           // false    ①
@debug null == false;            // false    ①
@debug Verdana == 'Verdana';     // true     ②
@debug 1cm == 10mm;              // true     ③
@debug 4 > 5 or 8 > 5;           // true
@debug 4 > 5 and 8 > 5;          // false
```

① true, false, and null are equal only to themselves.

② Both values are considered to be strings.

③ Converted to the same unit, they're equal in size; therefore, they're equal.

To check that our `$type` variable is equal to `'error'`, our condition will be `$type == 'error'` coupled with `@if` and `@else`. Our rule looks like the following listing.

Listing 12.27 Conditionally boldfacing the callout type (SCSS)

```
$callouts: (success: $success, warning: $warning, error: $error);
@each $type, $color in $callouts {
  @debug $type, $color;
  .#{$type} {
    @extend .callout;
    background-color: scale-color($color, $lightness: +86%);
    border-color: $color;
    &::before {
      content: "#{$type}: ";
      text-transform: capitalize;
      @if $type == 'error' {          ①
        font-weight: 800;             ①
      } @else {                       ②
        font-weight: 500;             ②
      }                               ②
    }
  }
}
```

① The type is error; therefore, we add a font width of 800.

② The type isn't error (it's either success or warning), so

font-weight is set to 500.

The following listing shows that font weights have been added to each type in the CSS output.

Listing 12.28 Conditionally boldfacing the callout type (CSS output)

```
.callout, .error, .warning, .success {
  border: solid 1px;
  border-radius: 4px;
  padding: 0.5rem 1rem;
}

.success {
  background-color: #f6f9d1;
  border-color: #747d10;
}
.success::before {
  content: "success: ";
  text-transform: capitalize;
  font-weight: 500;
}

.warning {
  background-color: #fff1dc;
  border-color: #fc9d03;
}
.warning::before {
  content: "warning: ";
  text-transform: capitalize;
  font-weight: 500;
}

.error {
  background-color: #fcd1d1;
  border-color: #940a0a;
}
```

```
.error::before {
  content: "error: ";
  text-transform: capitalize;
  font-weight: 800;
}
```

The text added as part of the `::before` pseudo-element has a `font-weight` of `500` for both `.success` and `.warning`. The `.error::before` rule, on the other hand, has a `font-weight` of `800`.

With this last detail added, our project is complete. Figure 12.19 shows the final output.


Figure 12.19 Finished project

## 2.8 Final thoughts

This chapter illustrates several things Saas lets us do that we can't do with CSS alone, but it covers only a small percentage of Saas's features and delves into only one preprocessor. Preprocessors can do much more; this chapter only scratches the surface. The takeaway is that preprocessors provide cool functionality that can make code more efficient to write and also more complex. They also require a build step and slightly more complicated setup.

Although we didn't dive into Less or Stylus, here are some questions that may help when you're choosing a preprocessor:

- Do I need a preprocessor?
- What functionality does the preprocessor need?
- How is using a preprocessor going to help the development of my project?
- If the project uses a user-interface framework or library, does it support one or more preprocessors? If so, which ones?

- What will having a preprocessor change about my build-and-deploy process because now the CSS needs to be built?
- What skills do my team members have, and which preprocessors are they familiar with?

Whether or not preprocessors are for you, the important thing to remember is that every project is different. Keep learning, exploring, and trying new things, and have some fun. Happy coding!

## ummary

- Sass has two syntaxes: indented and SCSS.
- Variables and CSS custom properties work differently.
- Sass variables are block-scoped.
- `@extend` extends existing rules, whereas mixins generate new code.
- Mixins can take parameters.

- When used in conjunction with `object-fit: cover`, `object-position` helps position an image within its bound box when the image doesn't have the same aspect ratio as the dimensions it's given.
- Interpolation is used to embed the result of an expression, such as when creating rule names from variables.
- The `border-radius` property can take multiple values to assign different curvature to each corner of an element, starting from top left and rotating clockwise.
- Sass allows us to nest rules.
- We can use `@each` to loop over lists and maps.
- `@debug` allows us to print values in the terminal output.
- Sass provides functions such as `scale-color()` to manipulate and alter colors.
- `@if` and `@else` can be used to determine conditionally whether a block of code should be evaluated.