1 Sass and Compass make stylesheets fun again

This chapter covers

- Getting started with Sass and dynamic stylesheets
- Writing stylesheets more efficiently with Sass features
- A quick introduction to Compass
- Compass solutions to real-world stylesheet challenges

Sass is an extension of CSS3 that helps you create better stylesheets with less effort. Sass frees you from repetition and gives you tools to be creative. Since you can implement changes much faster, you'll be free to take risks in your designs. Your stylesheets will be able to keep pace with changing colors and changing HTML markup, all the while producing standards-based CSS you can use in any environment. The Sass processor is written in Ruby, but unless you want to hack the language itself, you need not care.

Throughout this book, we speak to two sets of readers, hoping to find some common ground with each camp. If you find yourself in both groups, even better.

To our web designer friends: You have all the Adobe app keyboard shortcuts memorized. You choose complementary colors based on RGB values alone. You may or may not sport a pair of dark-rimmed glasses, but chances are you start your

day with coffee or tea and the latest from *Smashing Magazine*. By your own admission, you know enough jQuery to be dangerous and don't know why your developer friends chuckle when you talk about CSS as a language.

We'll set you free from the tedious and let you do what you do best—be creative. We know you have opinions on resets, typographic scales, color palettes, and layouts. We'll show you how to create stylesheets faster with less repetition. You'll start doing less in graphics software and more in your stylesheets.

To our front-end developer pals: You take pride in your ability to slice-and-dice a Photoshop comp into semantically sound HTML and CSS, but there's a problem. Your server templates are DRY because you *Don't Repeat Yourself*, but your stylesheets are as soggy as a doorbell-interrupted Raisin Bran breakfast. As the project grows, you also find that organizing your stylesheets is a challenge. If only you could author stylesheets in the same way you write the other code in your software project—with variables, reusable parts, and control flow. Take heart, have we got a project for you!

In this chapter, we'll look at powerful Sass features such as nested rules, variables, mixins, and selector inheritance, and how Compass leverages these into reusable patterns to free you from mindless repetition and let you focus on your design instead of your styles. If you don't already have Sass installed, go ahead and jump to appendix A and follow the steps outlined there. If you're reading this at the coffee shop on your iPad, you can still run these basic examples online at the Sass website: http://sass-lang.com/try.html.

1.1 Getting started with Sass

Before we jump into some examples, it's important to nail down some keys to success with Sass. Sass isn't a silver bullet or pixie dust. It won't instantly help your color, typography, or layout choices, but it can help you implement your ideas faster, with less friction. Before we get into syntax and features, let's take a look at the big picture. When using Sass, the Sass engine compiles your stylesheet source files into 100% pure CSS during your development workflow, as shown in figure 1.1.

Though there are many options for running the Sass engine, ranging from the command line to server framework integration to GUI tools, the key takeaway is that Sass produces CSS during your *development* workflow. You deploy *static* CSS as you normally would; you just benefit from Sass language features to write that CSS much faster and maintain it more easily.

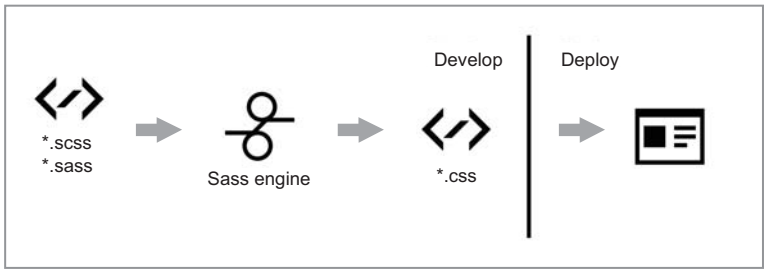

Figure 1.1 The Sass authoring and compilation workflow

1.1.1 From CSS to Sass

If you're skilled in creating CSS, you'll find the on-ramp to using Sass a short one. Sass focuses on *how* to create awesome stylesheets, not *what* goes into them. We'll cover tools like Compass that provide you with CSS best practices, but ultimately you'll benefit from this book if you have a firm grasp of CSS. As with anything in computing, *garbage in, garbage out*. If you need a CSS primer, you might want to check out another Manning title, *Hello! HTML5 and CSS3*.

Sass supports two syntaxes. The original *indented syntax* has a .sass extension and is whitespace aware, so instead of surrounding properties with braces, you indent them underneath their selector. Rather than using semicolons, each property is separated by a new line:

```
h1
  color: #000
  background: #fff
```

SCSS, or *Sassy CSS*, was introduced in Sass 3.0 and is a superset of CSS3. SCSS files have a .scss file extension and are chock-full of familiar braces and semicolons:

```
h1 {color: #000; background: #fff}
```

This demonstrates the primary differences between the two syntaxes, but there are other differences which are discussed in appendix C.

Sass will continue to support both syntaxes. You can even mix and match each syntax within the same Sass project (just not within a single file). It's important to choose a syntax that's right for you and your team. If you work in a Python or Ruby environment, perhaps the whitespace-aware indented syntax will fit nicely. If your team deals with outside design agencies, then Sassy CSS provides a lower barrier to entry.

In addition to sound CSS skills and a grasp of Sass syntax, it's important to take a dynamic view of stylesheets.

1.1.2 Think dynamic

Outside of basic brochure sites, who really writes much static HTML anymore? You take your HTML and carve it up for your blog engine, CMS, or application framework to *preprocess*, mixing markup and dynamic content. These tools give life to your HTML and it's crazy to imagine the web without them. So *why do you still write static stylesheets?* You'll see how the concepts you use in creating static markup, *dynamically*, can be applied to creating static stylesheets, *dynamically*. What does it mean to write dynamic stylesheets? It means that when you author Sass stylesheets, you're no longer limited by how the browser thinks about CSS. With conditional logic, reusable snippets, variables, and various other tools, you can bring your stylesheets to life. Changing a website's layout and color scheme can be as simple as tweaking a few variables. Of course, though Sass lets you write stylesheets in a dynamic fashion, the output is still 100% pure static CSS. Once you're working with dynamic stylesheets, you can now listen to that inner voice that keeps shouting *Don't Repeat Yourself*.

1.1.3 Don't Repeat Yourself

Sass gives stylesheet authors powerful tools that remove the tedium from many CSS tasks you do over and over and over. Many features of Sass embrace the familiar programming axiom *Don't Repeat Yourself*, letting you *DRY* up your stylesheets. As you create your stylesheets, repetition should be a red flag. Constantly ask yourself, *how can I work smarter, not just harder?* In the next few sections, we'll show you how to let Sass squeeze more reuse out of your stylesheets.

1.2 Hello Sass: DRYing up your stylesheets

We've been harping on DRY-DRY-DRY up to this point. So what does a soggy stylesheet look like? Consider the following CSS.

Listing 1.1 A soggy stylesheet in need of DRYing

```
h1#brand {color: #1875e7}

#sidebar { background-color: #1875e7}

ul.nav {float: right}
ul.nav li {float: left;}
ul.nav li a {color: #111}
ul.nav li.current {font-weight: bold;}

#header ul.nav {float:right;}
#header ul.nav li {float:left;margin-right:10px;}
#footer ul.nav {margin-top:1em;}
#footer ul.nav li {float:left;margin-right:10px;}
```

Even in this extremely simplified example, the duplication is apparent. What happens if the marketing team wants to tweak that lovely shade of blue from `#1875e7` to `#0f86e3`? Sure, two occurrences is manageable, but when it's a dozen or more across several stylesheets, find-and-replace seems archaic, don't you think? Eight instances of `ul.nav` in a 10-line stylesheet also seems excessive.

In the next few sections, you'll discover a cool breeze of syntactic sugar that will DRY up this stylesheet and blow you away, including variables, mixins, nested selectors, and selector inheritance. If we seem to move fast, don't fret. We dig deeper into each of these concepts in chapter 2.

1.2.1 Reuse property values with variables

Are you using search-and-replace to swap hex code values and manage color palette changes in your stylesheets? With Sass, you can assign values to *variables* and manage colors, border sizes, and virtually any stylesheet property value in a single location:

```
$company-blue: #1875e7;

h1#brand {
color: $company-blue;
}

#sidebar {
background-color: $company-blue;
}
```

Sass variables start with the `$` symbol and can contain any characters that are also valid in a CSS class name, including underscores and dashes. In this simple example, if you want to tweak the shade of blue, you can update it in one spot and the rest of your stylesheet falls in line.

If you come from a development background, variables should feel natural. If you're coming to Sass from a design background, variables may seem intimidating at first glance. But they're really nothing new. You already use named values in CSS such as `blue`, `green`, `inherit`, `block`, `inline-block`, `serif`, and `sans-serif`. Think of variables as your own special values. Next up, using nested selectors to create deep descendant CSS selectors with less typing.

1.2.2 Write long selectors more quickly with nesting

Did you ever hear about the Texan who went to work for the state painting dashed center lines on the highway? He was a top performer his first week, painting 10 miles of road. Production tailed off quickly, as he covered five miles in his second week, and only two in the third. When he rounded out the last week of the month with only a single mile, his supervisor asked him what seemed to be the problem. "Well," the worker remarked, "it keeps getting farther and farther back to the bucket."

That's exactly how it can feel working with deep descendant CSS selectors. Consider the following CSS:

```
ul.nav  {float: right}
ul.nav li {float: left;}
ul.nav li a {color: #111}
ul.nav li.current {font-weight: bold;}
```

Sass lets you *DRY* that up a bit. Find the file 1.1.2.nesting.scss in the code examples folder for chapter 1 or create your own by saving a text file with the following contents.

Listing 1.2 Nesting CSS selectors

```
ul.nav {
  float: right;

  li {
    float: left;
    a {
      color: #111;
    }
    &.current {
      font-weight: bold;
    }
  }
}
```

From your terminal, run the `sass` command and pass it the path to the file:

```
sass 1.2.nesting.scss
```

You should get the following CSS results in your terminal output.

Listing 1.3 Resulting CSS after using nested selectors

```
ul.nav {
  float: right; }
  ul.nav li {
    float: left; }
    ul.nav li a {
      color: #111; }
    ul.nav li.current {
      font-weight: bold; }
```

Other than some formatting differences, that's the same CSS we started with. (Don't sweat the format just yet. We'll discuss more about Sass's output options a bit later.)

Using Sass, you can *nest* rules and avoid duplicating the same elements in your selectors. Not only does this save time, the added benefit is that if you later change `ul.nav` from an unordered list to an ordered list, you only have one line to change. This is especially true with the last selector in the example. The `&` is a *parent selector.* In this case `&.current` evaluates to `li.current`. If the markup were to change to using the `current` class on some other element, this line in the stylesheet would just work. Now that you've seen how to reuse values with variables and write longer selectors with nesting, let's put the ideas together and look at Sass mixins.

1.2.3 Reuse chunks of style with mixins

Variables let you reuse *values*, but what if you want to reuse large blocks of rules? Traditionally in CSS, as you see duplication in your stylesheets, you factor common rules out into new CSS classes.

Listing 1.4 Traditional CSS refactoring

```
ul.horizontal-list li {
  float: left;
  margin-right: 10px;
}

#header ul.nav {
  float: right;
}

#footer ul.nav {
  margin-top: 1em;
}
```

You then need to give your `ul.nav` elements an additional class of `horizontal-list`. This works fine, but what if you wanted to keep your classes more semantic and still get the reuse?

Let's open or create 1.1.2.mixins.scss, our second example.

Listing 1.5 Reusing code with `@mixin` and `@include`

```
@mixin horizontal-list {
  li {
```

```
    float: left;
    margin-right: 10px;
  }
}

#header ul.nav {
  @include horizontal-list;
  float: right;
}

#footer ul.nav {
  @include horizontal-list;
  margin-top: 1em;
}
```

Just as the name suggests, Sass mixins *mix in* rules with other rules. You've extracted the rules for the horizontal list into an aptly named mixin using the `@mixin` directive. You then *include* those rules into other rules using the `@include` directive. You no longer need the `.horizontal-list` class, since those rules are now mixed into your `ul.nav` rules in your resulting CSS.

Listing 1.6 Mixins help you remove redundant styles

```
#header ul.nav {
  float: right;
}

#header ul.nav li {
  float: left;
  margin-right: 10px;
}

#footer ul.nav {
  margin-top: 1em;
}

#footer ul.nav li {
  float: left;
  margin-right: 10px;
}
```

As handy as this is, the real power of Sass mixins comes from combining them with variables to make reusable, parameter-driven blocks of styles. For example, let's suppose you wanted to vary the item spacing in your horizontal list. Find the next code example, 1.1.2.2.mixins-parameters.scss, and consider the following changes.

Listing 1.7 Mixins with variables

```
@mixin horizontal-list($spacing: 10px) {
  li {
    float: left;
    margin-right: $spacing;
  }
}
```

```
#header ul.nav {
  @include horizontal-list;
  float: right;
}

#footer ul.nav {
  @include horizontal-list(20px);
  margin-top: 1em;
}
```

You've updated the mixin and added a `$spacing` parameter with a default value of `10px`. Parameters are no different than the variables we looked at earlier. In this case, you've specified a default value so that in the case of the navigation list in the header, you get the default spacing. In the footer, you now can pass in a value of `20px` to get more spacing between the list elements, as you can see in the CSS output.

Listing 1.8 Final CSS output after using mixins

```
#header ul.nav {
  float: right;
}

#header ul.nav li {
  float: left;
  margin-right: 10px;
}

#footer ul.nav {
  margin-top: 1em;
}

#footer ul.nav li {
  float: left;
  margin-right: 20px;
}
```

Sass mixins save you a lot of time, letting you reuse chunks of properties, but the astute reader might notice that what you've gained in productivity, you may have given back in stylesheet weight, since mixin styles are duplicated in each instance where they're included. Fear not: with Sass, you always have options. Selector inheritance deals with just this issue.

1.2.4 Avoid property duplication with selector inheritance

As you've seen, Sass mixins can be a powerful way to avoid duplication when writing your stylesheets. Bur since rules are mixed into other classes in the compiled CSS, you're not avoiding duplication entirely. Because CSS file size is important, Sass includes another slightly more complex way of avoiding duplication altogether. Selector inheritance instructs a selector to *inherit* all the styles of another selector without duplicating the CSS properties. Take, for instance, the styles for a set of form error messages.

Listing 1.9 Some CSS for error messages

```
.error {
  border: 1px #f00;
  background: #fdd;
}

.error.intrusion {
  font-size: 1.2em;
  font-weight: bold;
}

.badError {
  @extend .error;
  border-width: 3px;
}
```

Using selector inheritance, you can instruct `.badError` to inherit from the base `.error` class, yielding the results shown next.

Listing 1.10 Reducing redundancy with selector inheritance

```
.error, .badError {
  border: 1px #f00;
  background: #fdd;
}

.error.intrusion,
.badError.intrusion {
  font-size: 1.2em;
  font-weight: bold;
}

.badError {
  border-width: 3px;
}
```

In this case, it makes sense to have both the `error` and `badError` classes, since you expect to use both of them in your HTML, but occasionally your base class isn't something you expect to use in your markup. In Sass 3.2, the placeholder selector was introduced to allow you to use selector inheritance without creating throwaway base classes.

Listing 1.11 Selector inheritance with the placeholder selector

```
%button-reset {
  margin: 0;
  padding: .5em 1.2em;
  text-decoration: none;
  cursor: pointer;
}

.save {
  @extend %button-reset;
  color: white;
```

```
  background: #blue;
}

.delete {
  @extend %button-reset;
  color: white;
  background: red;
}
```

As the name *placeholder* implies, the classes that extend `%button-reset` take its place in the generated CSS.

Listing 1.12 Resulting CSS after using selector inheritance

```
.save, .delete {
  margin: 0;
  padding: .5em 1.2em;
  text-decoration: none;
  cursor: pointer;
}

.save {
  color: white;
  background: #blue;
}

.delete {
  color: white;
  background: red;
}
```

Placeholders give you a safe way to store common styles without worrying that they'll interfere with any of your class names. Also, if a placeholder is never extended, the styles inside of it are never compiled to CSS, keeping your stylesheets light and free from the bloat of unused styles.

With a little planning, selector inheritance is a nice way to keep your Sass DRY and your CSS lean. Now that you've seen how Sass helps you avoid repeating yourself, in the next section you'll see what Compass brings to the table.

1.3 What is Compass?

Compass helps Sass authors write smarter stylesheets and empowers a community of designers and developers to create and share powerful frameworks. Put simply, Compass is a Sass framework designed to make the work of styling the web smooth and efficient. Much like Rails as a web application framework for Ruby, Compass is a collection of helpful tools and battle-tested best practices for Sass.

Compass is made up of three main components. It includes a library of Sass mixins and utilities, a system for integrating with application environments, and a platform for building frameworks and extensions. Expanding the big picture diagram from earlier in this chapter, let's see how Compass fits into your development workflow in figure 1.2.

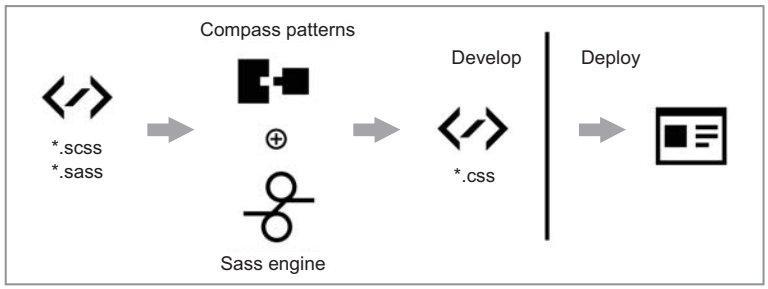

Figure 1.2 Compiling with Compass

1.3.1 The Compass library

Compass comes with a host of Sass mixins and functions that are organized into modules, all of which are thoroughly documented with examples on the Compass website. This library insulates you from cross-browser quirks and provides a great set of proven design patterns for resets, grid layouts, list styles, table helpers, vertical rhythm, and more. Compass also comes with helpers for CSS3, handling vendor prefixes and abstracting away different browser implementations of emerging CSS3 features, making it much easier to write cutting-edge stylesheets.

Compass can do some really handy tasks like measuring images from the filesystem and writing them into your stylesheets. Asset URL functions are available that make it easy to move assets around in a project or even switch to a content delivery network (CDN) without having to rewrite your stylesheets. Compass can even combine a directory of images into a single sprite image and do the otherwise tedious task of calculating coordinates and writing the spriting CSS for you.

These are tasks you could tackle yourself, and sometimes will, but Compass bundles proven solutions from the design community, letting you focus on getting more done in less time.

The Compass Core stylesheet framework isn't going to make your website pretty. In fact, all features in the core framework are *design agnostic* so that they can be used with any website design. Website design aesthetics, like all fashions, come and go. So the task of providing well-designed website features is left to the Compass community of front-end developers and designers through the use of plugins.

1.3.2 Simple stylesheet projects

Both Sass and Compass are written in Ruby and have their origins in the Ruby on Rails community, but Compass provides tools and configuration options to make it easy to write Sass stylesheets outside of Ruby-based projects. Whether you need to simply build an HTML mockup or to integrate Sass into a large application framework like Django, Drupal, or .NET, Compass makes it a snap (see figure 1.3).

Compass understands that you aren't building stylesheets. You're building a design. As such, Compass wants to know where you keep things like image, font, and JavaScript files so that it can simplify the management of and references to those files

from within your stylesheets. For example, Compass will help you construct sprite maps and refer to those within your stylesheets; Compass will warn you if you reference an image that doesn't exist via the `image-url()` helper; and Compass can embed an image or font into your CSS so that the browser doesn't have to make another round trip to get that asset.

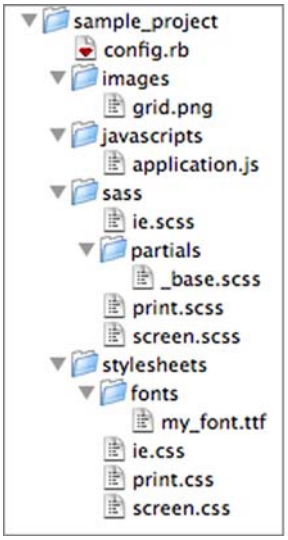

Figure 1.3 A standalone compass project

1.3.3 Community ecosystem

If you've been in web development for a while, you might remember the dark ages before JavaScript frameworks. It was truly a terrible world—the smallest quirk in the DOM might send you on a bug hunt for hours. These days, JavaScript frameworks isolate you from the browsers' inconsistencies and give you a foundation for sharing your code through plugins that others can easily drop into their projects. Thanks to the hard work of the web development community at large, developing with JavaScript is actually enjoyable these days.

As a framework for Sass, Compass is a foundation for designers and developers to share their libraries and frameworks, empowering you to participate in an ecosystem of open source stylesheet development. Fading quickly are the days when sharing a bit of CSS wizardry meant embedding code snippets and demo files in a blog post. This strategy leaves each user owning their code without a way for the original developer to fix bugs and provide additional enhancements over time. With Compass, stylesheet libraries can be distributed like other software, which means fixing a bug or getting support for the latest browsers may just be a simple matter of upgrading and recompiling your stylesheets.

Many community members package up their bag of tricks into Compass extensions for others to begin using immediately, without requiring them to rewrite a nasty nest of static stylesheets. (See chapter 10 to learn how to write your own Compass extension.) Responsive layouts, typographic scales, custom animations, fancy buttons, icon sets, and color palettes can all be made into Compass extensions written in Sass. Compass extensions get you past the drudgery of building the basics so you can focus on what's unique and special about your website. As you progress from Sass novice to Sass assassin, if you're grateful for all the time Sass, Compass, and the community save you, you'll be able "pay it forward" by sharing your hard work with others.

1.4 Create a Compass project

If you haven't installed Compass already, go ahead and jump to appendix A and follow the instructions outlined there. After you have the bits installed, you'll be ready to start using Compass. Your first task will be creating a Compass project.

Like any good command-line interface (CLI), Compass provides substantial help messages for its many options. Let's check your Compass install. Open a terminal window in the root of a new stylesheet project. Now, run `compass help`. If you're greeted with help text and command-line options, you're good to go. If not, circle back to appendix A one more time and we'll see you on the flip side.

Let's start by creating a new Compass *project*, which is a configuration file and folders for your Sass source and CSS output. We'll call it *sample*:

```
compass create sample
```

Now list the contents of your new folder:

```
total 8
drwxr-xr-x  6 wynn  staff  204 Jan  3 12:11 .
drwxr-xr-x  3 wynn  staff  102 Jan  3 12:12 ..
drwxr-xr-x  4 wynn  staff  136 Jan  3 12:11 .sass-cache
-rw-r--r--  1 wynn  staff  315 Jan  3 12:11 config.rb
drwxr-xr-x  5 wynn  staff  170 Jan  3 12:11 sass
drwxr-xr-x  5 wynn  staff  170 Jan  3 12:11 stylesheets
```

Using the defaults, Compass has unfurled a `config.rb` configuration file, a `sass` folder for your Sass source, and a `stylesheets` folder for your CSS output. For a full list of Compass configuration options, please consult appendix B. For now, we'll work with the default settings and set out to tackle some real-world CSS problems using Compass.

1.5 Solve real-world CSS problems with Compass

Now that you've seen how to create a skeleton Compass project, let's take a look at how Compass can help solve some stylesheet challenges you probably face every day. In the next few sections you'll apply Compass's built-in modules (which are only nice bundles of Sass mixins and other features) to CSS resets, grid layouts, table formatting, and CSS3 features.

1.5.1 Clear the canvas with resets

Made popular by Eric Meyer and other standards advocates, adding a *CSS reset* has become the first task for designers when creating a stylesheet. If you've ever used a CSS grid framework, you've used a CSS reset, perhaps without even knowing it. A CSS reset simply removes all intrinsic browser styling from all elements, providing a common blank canvas to add back the styling you want.

Eric's classic reset looks like this.

Listing 1.13 Classic CSS reset

```
/* v1.0 | 20080212 */

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, font, img, ins, kbd, q, s, samp,
```

```
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td {
  margin: 0;
  padding: 0;
  border: 0;
  outline: 0;
  font-size: 100%;
  vertical-align: baseline;
 background: transparent;
}
body {
  line-height: 1;
}
ol, ul {
  list-style: none;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
  content: '';
  content: none;
}

/* remember to define focus styles! */
:focus {
  outline: 0;
}

/* remember to highlight inserts somehow! */
ins {
  text-decoration: none;
}
del {
  text-decoration: line-through;
}

/* tables still need 'cellspacing="0"' in the markup */
table {
  border-collapse: collapse;
  border-spacing: 0;
}
```

You might have noticed from the default Sass file, screen.css, that Compass ships with its own reset based on Eric's, allowing you to put all browsers on equal footing with a single line in your Sass file:

```
@import "compass/reset"
```

There's a lot going on in this one line, so let's break it down. You use the Sass `@import` rule to import the Compass Reset module. A module is a standalone portion of the

Compass framework that can be added independently to your project. With this one line, the contents of your CSS output file include your CSS reset.

Listing 1.14 CSS output file, including CSS reset

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, font, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td {
  margin: 0;
  padding: 0;
  border: 0;
  outline: 0;
  font-weight: inherit;
  font-style: inherit;
  font-size: 100%;
  font-family: inherit;
  vertical-align: baseline;
}

body {
  line-height: 1;
  color: black;
  background: white;
}

ol, ul {
  list-style: none;
}

table {
  border-collapse: separate;
  border-spacing: 0;
  vertical-align: middle;
}

caption, th, td {
  text-align: left;
  font-weight: normal;
  vertical-align: middle;
}

q, blockquote {
  quotes: "" "";
}
q:before, q:after, blockquote:before, blockquote:after {
  content: "";
}

a img {
   border: none;
}
```

It should be noted that adding styles to your stylesheet isn't the default behavior of most Compass modules, but since the usual use case is to apply the CSS reset, the Compass Reset module goes ahead and applies the `global-reset` mixin upon import. Let's take a look at that mixin.

Listing 1.15 CSS `global-reset` mixin

```
@mixin global-reset {
  html, body, div, span, applet, object, iframe,
  h1, h2, h3, h4, h5, h6, p, blockquote, pre,
  a, abbr, acronym, address, big, cite, code,
  del, dfn, em, font, img, ins, kbd, q, s, samp,
  small, strike, strong, sub, sup, tt, var,
  dl, dt, dd, ol, ul, li,
  fieldset, form, label, legend,
  table, caption, tbody, tfoot, thead, tr, th, td {
    @include reset-box-model;
    @include reset-font; }
  body {
    @include reset-body; }
  ol, ul {
    @include reset-list-style; }
  table {
    @include reset-table; }
  caption, th, td {
    @include reset-table-cell; }
  q, blockquote {
    @include reset-quotation; }
  a img {
    @include reset-image-anchor-border; } }
```

Note that Compass is using the Sass `@mixin` and `@include` features we looked at earlier to build the reset. In addition to the `global-reset`, the Reset module includes a number of more surgical reset mixins, including one for HTML5 elements. By adding `@include reset-html5` to your Sass file, you get an additional CSS rule in your output for all the HTML5 elements that need some basic styling.

Listing 1.16 Resulting code after HTML5 reset

```
article, aside, canvas, details, figcaption, figure,
footer, header, hgroup, menu, nav, section, summary {
  margin: 0;
  padding: 0;
  border: 0;
  outline: 0;
  display: block;
}
```

For additional Compass Reset module mixins, be sure and check out the Compass online docs. Now that you have a handle on resets, let's look at how Compass can help you more effectively use CSS grid frameworks.

Figure 1.4 **960.gs—the 960 Grid System CSS framework**

1.5.2 *Create layouts without a calculator*

One of the major trends in CSS in the last couple of years has been the emergence of popular CSS grid frameworks such as Blueprint and 960 Grid System (see figure 1.4). Grid layouts, which have long been a cornerstone of good print design, have made their way online as the medium has matured. Grid frameworks allow you to allot a certain number of columns for your layout and then apply a column-based layout with uniform gutters to your content.

Basically, grid frameworks reduce the math needed to create a nice column layout. They do this with CSS rules that set the layout and widths for a container element as well as for each possible column width in the grid. Let's look at a snippet from Blueprint.

Listing 1.17 Blueprint grid layout

```
.container {
  width: 950px;
  margin: 0 auto;
}

/* Sets up basic grid floating and margin. */
.column, .span-1, .span-2, .span-3, .span-4, .span-5,
.span-6, .span-7, .span-8, .span-9, .span-10, .span-11,
.span-12, .span-13, .span-14, .span-15, .span-16,
.span-17, .span-18, .span-19, .span-20, .span-21,
.span-22, .span-23, .span-24 {
  float: left;
  margin-right: 10px;
}

/* The last column in a row needs this class. */
.last { margin-right: 0; }

/* Use these classes to set the width of a column. */
.span-1 {width: 30px;}

.span-2 {width: 70px;}
.span-3 {width: 110px;}
.span-4 {width: 150px;}
.span-5 {width: 190px;}
.span-6 {width: 230px;}
.span-7 {width: 270px;}
.span-8 {width: 310px;}
.span-9 {width: 350px;}
.span-10 {width: 390px;}
.span-11 {width: 430px;}
.span-12 {width: 470px;}
.span-13 {width: 510px;}
.span-14 {width: 550px;}
.span-15 {width: 590px;}
.span-16 {width: 630px;}
.span-17 {width: 670px;}
.span-18 {width: 710px;}
.span-19 {width: 750px;}
.span-20 {width: 790px;}
.span-21 {width: 830px;}
.span-22 {width: 870px;}
.span-23 {width: 910px;}
.span-24 {width:950px; margin-right:0;}
```

With these CSS rules in place, you can create a 16-column layout simply by adding the `container` class to a container element and a `span-xx` class to each element you want to place on the grid. Laying out content in this way also lets you prototype more quickly by not having to remember the multiples of 40 between 30 and 950.

So how does Compass improve upon CSS grid frameworks? First, Compass provides support for grid framework styles as mixins, allowing you to pull in just the features you want to use while avoiding littering your HTML markup with extra classes. The

second, and perhaps most important, way Compass supports grid frameworks is in the way it changes how you create these frameworks, as you'll see in chapter 4.

Let's create a Compass project using Blueprint. Run the following in a terminal window:

```
compass create my_grid --using blueprint
```

Just as in section 1.4, you should find a freshly stamped Compass project in a folder called my_grid, only this time the screen.scss file has more content. The file is well annotated and provides a quick survey of most of the Blueprint modules at your disposal along with a set of styles for a basic layout. The first thing to notice here is that column layouts can be *mixed in* to a set of styles. So instead of setting a class of `span-8` in your HTML, you use the `column` Sass mixin:

```
@include column($sidebar-columns);
```

Also note the variable `$sidebar-columns`. This is extremely powerful because now, thanks to Sass, you can make your layouts variable-driven. You can rapidly prototype and play with different layouts including number of columns, gutter width, and sidebar sizes all by changing a few variables at the top of your Sass file. To do this in traditional CSS grid frameworks, you'd have to do the math to create those CSS layouts, and then change the CSS classes in your markup as well.

We won't go into all the aspects of the Blueprint grid here. We jump into using Blueprint with Compass later in chapter 6. We'll continue our survey of real-world Compass applications by taking a look at the Compass table helper.

1.5.3 Zebra-stripe like a pro with table helpers

Continuing our overview of Compass features, let's look at the Compass table helpers, a set of Sass mixins that make prettifying your HTML tables easier. Let's look at an example.

Listing 1.18 Compass table helpers

```
@import "compass/reset"
@import "compass/utilities/tables";

table {
  $table-color: #666;
  @include table-scaffolding;
  @include inner-table-borders(1px, darken($table-color, 40%));
  @include outer-table-borders(2px);
  @include alternating-rows-and-columns($table-color,
              adjust-hue($table-color, -120deg), #222222); }
```

Now let's break this down. You import the table helpers using the `@import` rule. This provides four mixins for your use. The `table-scaffolding` provides base styles for your `th` and `td` elements that you stripped with your CSS reset, as well as an often-used pattern of right alignment for numeric columns. Here's the source for this mixin.

Listing 1.19 Table helper mixin

```
@mixin table-scaffolding {
  th {
    text-align: center;
    font-weight: bold; }
  td,
  th {
    padding: 2px;
    &.numeric {
      text-align: right; } } }
```

The `inner-table-borders` and `outer-table-borders` mixins work as advertised, adding borders to the table and to cells within the table.

Lastly, the `alternating-rows-and-columns` mixin is an easy way to add some zebra-striping to your HTML table. You might ask why you wouldn't use the `:nth-child`, `:even`, or `:odd` CSS pseudo selectors for this task, and you'd be right to ask. That's exactly what Compass is doing under the hood. But this mixin provides some additional support for class-name-based striping as well as color intersections. Let's look at the source.

Listing 1.20 A mixin for alternating colors by row or column

```
@mixin alternating-rows-and-columns(
        $even-row-color,
        $odd-row-color,
        $dark-intersection,
        $header-color: white,
        $footer-color: white) {

   th {
     background-color: $header-color;
     &.even, &:nth-child(2n) {
     background-color: $header-color - $dark-intersection; }
  }
  tr.odd {
    td {
      background-color: $odd-row-color;
      &.even, &:nth-child(2n) {
        background-color: $odd-row-color - $dark-intersection; }
      }
  }
  tr.even {
    td {
       background-color: $even-row-color;
       &.even, &:nth-child(2n) {
         background-color: $even-row-color - $dark-intersection; }
       }
    }
    tfoot {
      th, td {
        background-color: $footer-color;
        &.even, &:nth-child(2n) {
           background-color: $footer-color - $dark-intersection; }
      }
    }
```

Note that the color values are not only variables; they're employing a bit of math to ensure proper contrast for readability. You'll learn more about how Sass deals with variables and math in the next chapter. Let's keep moving with a look at how Compass means never having to write vendor prefixes again.

1.5.4 Easy CSS3 without vendor prefixes

When CSS3 started gaining adoption by modern browsers, designers were excited to start using CSS for tasks that used to require stupid stylesheet tricks. We were so excited that we could now make those glorious rounded corners with a few lines of CSS that we didn't mind the vendor prefixes that came with them very much. *Vendor prefixes* are those `-webkit` and `-moz` bits that browsers add on to CSS features that have experimental support. In its simplest form, this means that to give a `<div>` a set of rounded corners with a `5px` border radius, you have to resort to CSS like this:

```
.rounded {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
}
```

As usual, Compass can save you from the repetition with a set of border radius mixins found in the Compass CSS3 module. Import the module into your Sass file and include the mixin:

```
@import "compass/css3";
.rounded {
  @include border-radius(5px);
}
```

This will yield the following CSS:

```
.rounded {
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  -o-border-radius: 5px;
  -ms-border-radius: 5px;
  border-radius: 5px;
}
```

Not only did you save your fingers from the common repetition of `-webkit` and `-moz` but you're also being good designers and supporting the other common vendor namespaces as well. Though that bit of repetition isn't horrible, what if you only want one of the four corners to be rounded? Well, the Mozilla folks don't yet see eye to eye with the rest of the field on the best way to make that happen, so you're left with this:

```
.rounded-one {
  -moz-border-radius-topleft: 5px;
  -webkit-border-top-left-radius: 5px;
}
```

That's where Compass shines. You can target a single corner for a border radius with the `border-corner-radius` mixin:

```
.rounded-one {
  @include border-corner-radius(top, left, 5px);
}
```

This will give you the CSS you want, Mozilla quirk included:

```
.rounded-one {
  -moz-border-radius-topleft: 5px;
  -webkit-border-top-left-radius: 5px;
  -o-border-top-left-radius: 5px;
  -ms-border-top-left-radius: 5px;
  border-top-left-radius: 5px;
}
```

That's just the tip of the tip of the tip of the CSS3 iceberg in Compass. We take a deeper look at all the time-saving features in chapter 9.

1.6 Summary

In this first chapter, we've looked at the case for CSS preprocessing. We took a quick look at four key features of Sass: variables, nested selectors, mixins, and selector inheritance. We also looked at some real-world applications of Sass included in the Compass framework, including CSS resets, grids, table styling, and CSS3 rounded corners.

In the next chapter, we'll dive deeper into Sass syntax, including color functions and scripting support. After you get a bit more Sass under your belt, we'll take a deeper look at Compass.