Sass Essentials

Develop efficient and streamlined CSS styles using Sass for any website or online application with minimal effort and maximum scope for reusability in future projects

Alex Libby

Sass Essentials

Develop efficient and streamlined CSS styles using Sass for any website or online application with minimal effort and maximum scope for reusability in future projects

Alex Libby

Sass Essentials

Credits

Author
  Alex Libby

Reviewers
  Nat Burns

  Ricardo Zea

Acquisition Editor
  Harsha Bharwani

Content Development Editor
  Athira Laji

Technical Editor
  Humera Shaikh

Copy Editors
  Relin Hedly

  Sonia Mathur

Project Coordinator
  Bijal Patel

Proofreader
  Safis Editing

Indexer
  Monica Ajmera Mehta

Production Coordinator
  Arvindkumar Gupta

Cover Work
  Arvindkumar Gupta

About the Author

Alex Libby comes from an IT support background. He has been involved in supporting end users for almost 20 years in a variety of different environments. Currently, Alex works as a technical analyst and supports a medium-sized SharePoint estate for an international parts distributor in the UK. Although he gets to play with different technologies in his day job, his first true love has always been the open source movement, particularly, experimenting with CSS/CSS3, jQuery, and HTML5. To date, Alex has written nine books based on jQuery and CSS3, among others, for Packt. *Sass Essentials* is Alex's tenth book for Packt. He has reviewed several books and videos as well.

About the Reviewers

Nat Burns is a professional software developer with experience in both frontend and backend technologies. He graduated from Calvin College in Grand Rapids, Michigan. Nat holds degrees in both computer science and music composition. He has lectured at the Great Lakes Software Excellence Conference and is an active contributor to a variety of open source projects and online forums. Currently, Nat is employed at MSCI, a leading provider of investment decision support tools with software products and services used by investors worldwide.

Special thanks to my wife, Anna, whose love and support for me goes beyond words.

Ricardo Zea is originally from Medellín, Colombia. He is a passionate and seasoned full-stack designer, who is now located in Dayton, OH (USA). Ricardo is always looking for ways to level up his skills and of those around him. He constantly wonders how things are made on the Web, how they work, and why, which has made him a very good technical designer. This aspect allows Ricardo to explain to the intricacies of design and the technicalities of the Web to others in ways that are very easy to understand and assimilate.

With a master's degree in publicity and advertising and a deep passion for understanding human behavior along with a fierce and competitive PC gaming hunger, he is able to switch from the creative side of the brain to the rational side very easily. This allows him to visualize and create technically sound web and mobile designs that are responsive, perform well, and convey the proper message through design.

Ricardo is the co-organizer of the CodePen Dayton meetup group. He is also a member of the Dayton Web Developers and UX Dayton meetup groups. Ricardo is one of the first members of SitePoint's Ambassadors program. He authors the monthly newsletter *Level Up! Web Design & Development Newsletter*. *Sass Essentials* is the second book that Ricardo has participated in as a technical reviewer with Packt. He's also the author of the upcoming book *Mastering Responsive Web Design with HTML5 and CSS3* with Packt as well. For several years, he was also a Flash and CorelDRAW professor at different universities in Colombia, his home country.

Ricardo has 15 years of experience in web design and 20 years of experience in visual and graphic design.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Picture this scene—you consider yourself pretty au fait with developing CSS styling; you've produced some stunning work over the years, spending time carefully crafting styles for demanding clients.

A colleague starts to band this word "Sass" around the office—you soon hear others talk of this "Sass" on Twitter and begin to wonder what it's all about.

Well, fear no more—welcome to the world of Sass! Throughout the course of this book, we'll delve into the world of CSS preprocessing, where we will explore tips and tricks that can help you make your CSS development more efficient, using Sass. Of course, you're probably wondering how; all will be revealed. Consider this as a taster though—how often have you spent time choosing colors for buttons only to find out that the client doesn't like the color scheme.

Sound familiar? I'll bet the answer is yes. In the next few pages, I will show you the tricks you need to know to create a color scheme from a single color. Let's just say that should you find out that your clients want the color changed, long evenings of changing code will become a thing of the past.

Let's dive in and start our journey through the world of Sass.

What this book covers

Chapter 1, *Introducing Sass*, takes you through the journey of Sass and discusses its essentials. This chapter will introduce Sass, tell you how to install it, and explore how it will make your development workflow smarter and more efficient, ultimately saving you valuable time. It will also help you create a simple layout and show you how to use Sass to compile the code to produce valid CSS.

Chapter 2, *Creating Variables and Mixins*, explores a key facet of Sass in the form of using variables. It discusses how editing one value can automatically update a whole series of rules at a single stroke. Just imagine altering the font sizes for your website in one go by simply editing one number!

Chapter 3, *Building Functions, Operations, and Nested Styles*, talks about how to use functions and operators to construct an entire site from just a handful of variables. We will also touch on nesting styles to help make code easier to read and maintain.

Chapter 4, *Directing Sass*, takes a look at another key element of Sass in the form of directives. It provides information on how to use some of the more popular options, such as @extend or @media. These will be used to a great effect in a series of demos, which quickly illustrates how careful use of these directives can have a significant positive effect on our code.

Chapter 5, *Incorporating Sass into Projects*, examines how to apply some of the tips and tricks we have covered throughout the book into more practical uses. It talks about several popular frameworks, such as Bootstrap, WordPress, and CSS grids. This chapter also shows you how Sass can be used to great effect to help construct sites, while reducing the amount of code we need to write.

What you need for this book

All you need to work through most of the examples in this book is a simple text or code editor, Internet access, and a web browser. I recommend installing Sublime Text, preferably version 3. It has an excellent add-in package system that works well; we will make use of this at various stages throughout the book.

Some of the examples make use of additional software, such as Node or Grunt—details are included in the appropriate chapter along with links to download the application from source.

Who this book is for

The book is for frontend developers who want to quickly get up to speed with using Sass. It will make writing CSS styles more efficient. To get the most out of this book, you should have a good working knowledge of HTML, CSS, and jQuery; we will use a little of the latter later in the book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We'll start by extracting the relevant files from the code download for this book—for this demo, we'll need `clicktoggle.scss` and `clicktoggle.html`."

A block of code is set as follows:

```
$baseWidth: 800px;
$mainWidth: round($baseWidth / 1.618);
$sidebarWidth: round($baseWidth * 0.382);

@function calculateRem($size) {
  $remSize: $size / 16px;
  @return $remSize * 1rem;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
min-height: 100px;

#container
  //media query - screen and 320px
  @include breakpoint($phone) {
    #container { max-width: 800px; background: #ccc; box-
      shadow: none; margin: 0 auto; }
  }
```

Any command-line input or output is written as follows:

```
npm install
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Once completed, click on Finish."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/4301OS_Graphics.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introducing Sass

A key part of any website development is creating the styling. Over the years, we've moved from simple styles, such as `<i>`, `<b>`, and `<h1>`, to creating complex special effects using CSS3. Building the style code takes time and effort. In this chapter, we'll introduce Sass and see how it will make your development workflow smarter and more efficient, ultimately saving you valuable time. We will also take a look at the following:

- Installing and compiling Sass using the command line
- Adding support to compile Sass within text editors or using Node
- Working with GUI-based compilers
- Using source maps
- Creating a simple layout using Sass

Ready to make a start?

> All the examples will be geared towards the Windows platform because this is my normal platform of use; where possible, alternatives for Mac and Linux platforms will be given.

Getting acquainted with Sass

Let's set the scene. I can imagine that you're an accomplished developer who is adept at crafting web pages and styling them with carefully coded CSS.

You're probably thinking that you don't have anything to learn, right? After all, you're happy with handling vendor prefixes; you know exactly what needs to be done to ensure that the latest CSS3-based styles work in all browsers. You're not keen on changing an established workflow that favors writing even less code, creating reusable blocks of code that will help you save even more time later, or even reusing code that others have already written for you, or are you?

Well, if you are, you've definitely come to the right place. Welcome to the world of Sass! What is it all about? Simply put, Sass is a CSS preprocessor or an extension of CSS. In some respects, it's akin to taking shorthand notes from a presentation and then pushing them through a process that spits out valid CSS at the other end.

Now, before you panic and run for the hills, dreading the thought of having to learn a whole new way of working, there's no need to worry. Sass can be as easy or as hard as you make it. Sure, there's a lot of useful functionalities that you will likely come across in time, but for now, with a few changes to your workflow, you'll see how easy it is to incorporate enough of the basics of Sass to get you started.

When working with Sass, I've found it wise to keep three principles in mind:

- **Follow the KISS principle**: Don't try to overcomplicate matters at this stage. Sass allows you to produce very complex functions. Chances are that if you're getting too complex, then it is likely that you need to rethink your styling.

- **Sass is just an extension of CSS**: There is no benefit (or need) to change everything in one go. As long as you have the basic compilation process set up with the source and destination files in the right place, then one of the larger parts of the work is done.

- **Automate or reuse whenever you work with Sass**: With tools such as Grunt, Compass, Sass, and so on, a large part of your work is done for you. We'll take a look at various tricks you can use to help get Sass to do the heavy Grunt work for you throughout the book.

Keep these principles in mind; you will see that with only a few modifications to your workflow, Sass will help you be more effective at writing CSS. The beauty of Sass is that if you already use well-known tools, such as Grunt or Node, then we can easily use both tools to compile Sass. We can even use autoprefixers to deal with those vendor prefixes. I don't know about you, but I am sure you have better things to do than work out what vendor prefixes are needed for your code.

Okay, let's make a start. The first task we need to look at is setting up Sass; before we do so, we just need to set up our project area, so let's get that out of the way first.

Preparing our project area

For each project we work on, we'll save all the files in a project area. So, go ahead and create a new folder called `projects`; we will refer to this throughout the book as our project workspace. Within this `projects` folder, we need to add some subfolders. So, let's add `css`, `img`, and `sass`, as shown in the following screenshot:

> The `node_modules` folder will be added automatically later in the chapter.
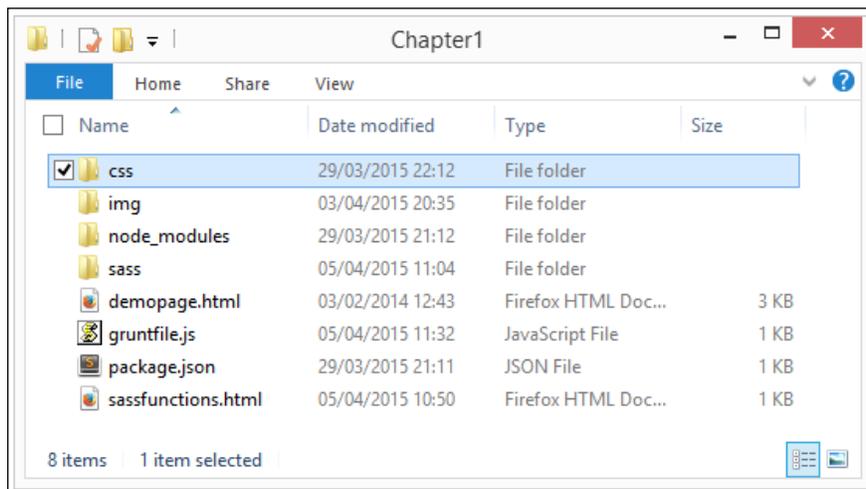
Now that preparing our project area is out of the way, let's get on and set up Sass, which is ready for use.

Installing and compiling Sass

The first step in our journey is to get Ruby installed; this is a dependency to work with Sass. The easiest route to do this is to use the prebuilt RubyInstaller application, which is available at `http://rubyinstaller.org`. This includes both Ruby: the execution environment and associated documentation as part of the same package.

Let's make a start. Perform the following steps:

1. We'll start by installing Ruby. Navigate to `http://rubyinstaller.org/downloads` and click on the **Download** button to download the application.

2. From the list under **RubyInstallers**, click on the top most entry. Make sure that you pick the right version for your platform (32-bit and 64-bit versions are available). Double-click on `rubyinstaller-2.2.1.exe` to begin the setup process, select your language of choice, tick **I accept the license**, and click on **Next**.

3. On the **Installation Destination and Optional Tasks** dialog, set the options and then click on **Install**:
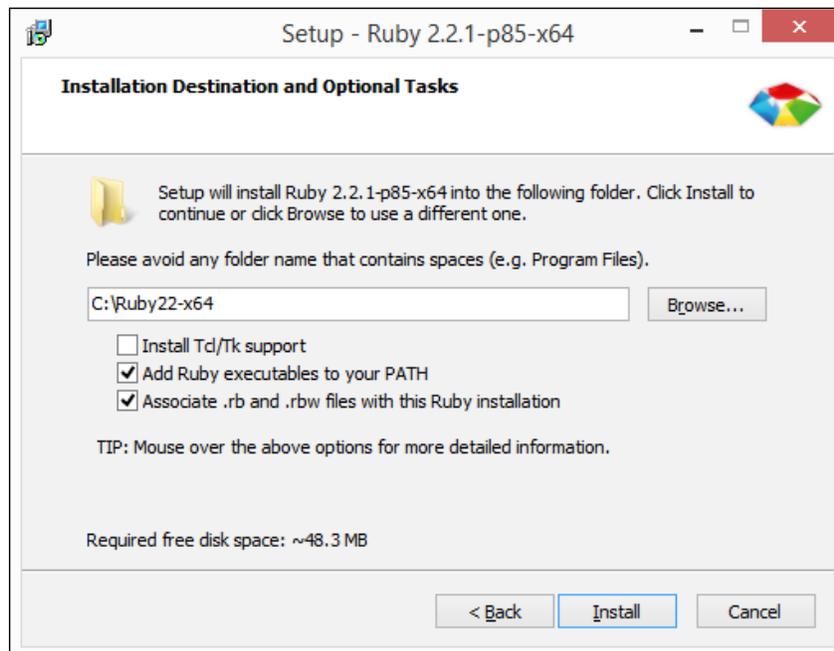
4. Once completed, click on **Finish**.

> If you're using Linux, you can install Ruby through the apt package manager, `rbenv`, or `rvm` with the `sudo su -c "gem install sass"` command. For Mac users, you can skip this stage because Ruby comes preinstalled.

At this point, we will have a working Ruby installation, so you can go ahead with the installation of Sass with the Gems package management framework:

1. Bring up Command Prompt. Then, enter the following command:

    gem install sass

2. Gems will run through the installation, as shown in this screenshot:

3. Once completed, enter the following command at Command Prompt to confirm a successful installation:

    sass -v

4. If it is successful, then Sass will respond with the version number and the working name of the installation (in this case, `Selective Steve`).

Right, okay. We have Sass installed, so what's next? Ah yes! Of course, we can use any decent text editor to modify Sass files (as they are plain text). However, to make editing easier, we should install some form of syntax highlighting.

Let's do this now. For the purpose of this exercise (and throughout the book), I will assume that you are using the cross-platform Sublime Text editor, which can be downloaded from `http://www.sublimetext.com`. It's not free, but there is no time limit to evaluate the package. For a per-user license of USD 70, you can use this package on any number of PCs without requiring extra licenses. Now, what if other editors offer that flexibility, I wonder?

Adding Sass support to Sublime Text

While we ponder how powerful and flexible Sublime Text is, let's make a start with getting it installed and adding Sass support:

1. The first step is to install Sublime Text; if you don't already have it, head over to `http://www.sublimetext.com`. Then, click on the link for the version appropriate to your platform. If you already have it installed, then move onto the next step.

2. Once completed, we need to install Package Control. This is Sublime Text's package manager, which we will use to install any new package in the application. Head over to `https://packagecontrol.io/installation` and follow the instructions at this link.

Once installed, we can now add the highlighting support; we will use the package available at `https://github.com/P233/Syntax-highlighting-for-Sass`, which can be installed from within Sublime Text itself.

1. Open Sublime Text. Then, click on **Preferences | Package Control** to bring up Package Control and enter install. Click on **Package Control: Install Package** when it appears:
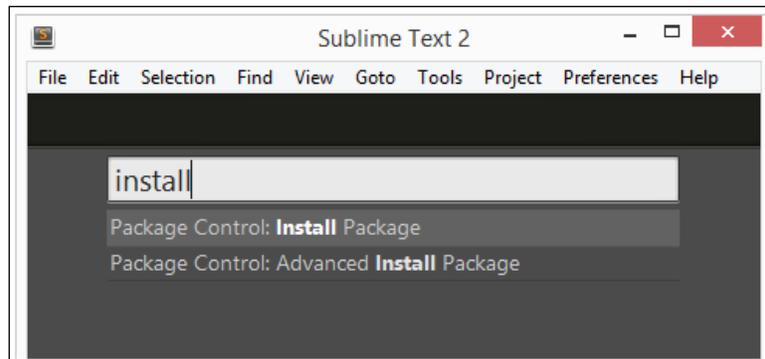
> You can open Package Control from the keyboard by hitting *Command + Shift + P* (Mac) or *Shift + Ctrl + P* (Windows).

2. When the **Install** option shows a list of packages, enter `sass` in the textbox to show the Sass package; once displayed, click on it to begin the installation:
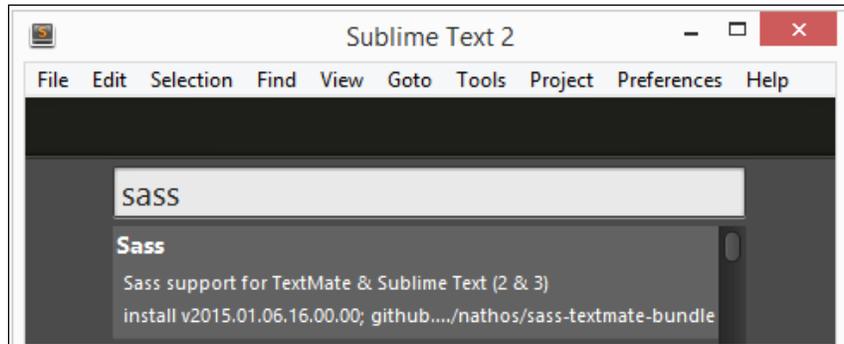
3. The installation of Sass highlighting support will be completed once the following screenshot is displayed in the status bar of Sublime Text:
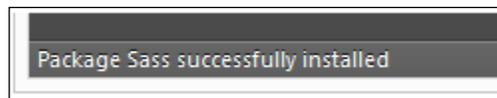
4. At this stage, if you were to try editing a Sass file, you may find that it is still devoid of color; if this is the case, to fix this, we may need to tell Sublime Text which syntax to use when editing Sass files. To do this, click on **Preferences | Package Control**. Then, remove any text in the drop-down field. Enter **Sass** in the text field and select **Set Syntax: Sass** from the drop-down field:
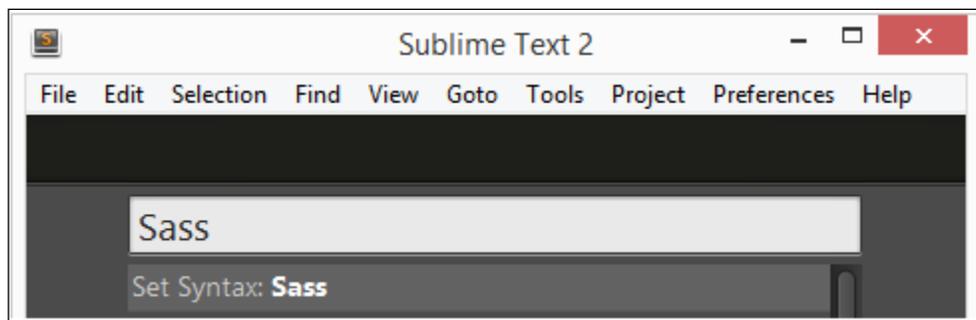
You can open Package Control using *Ctrl + Shift + P* (for Linux/Windows) or *Command + Shift + P* (for OS X).

If all is well, the text will now appear in color. Once we start creating Sass files later in this chapter, we should see something akin to this:
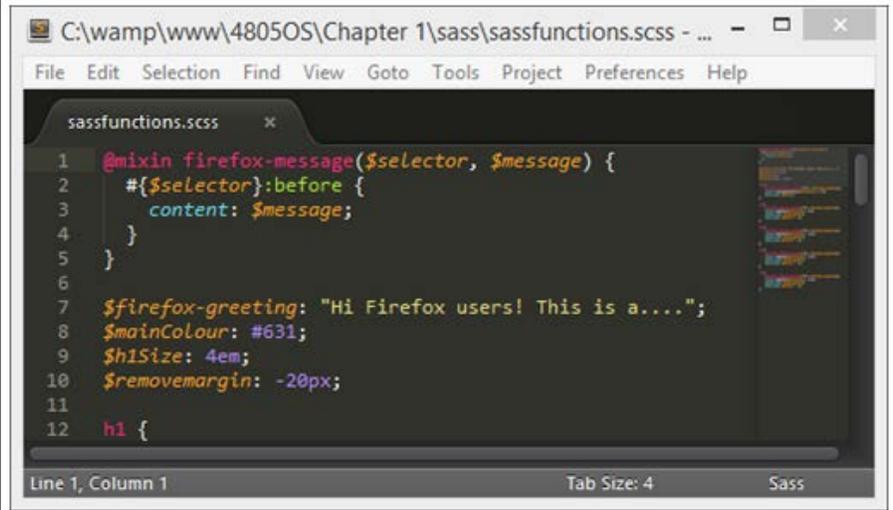
Okay, let's move on. Now that we can view Sass files with the correct syntax highlighting, we need to compile our Sass content in valid CSS files. There are several ways to achieve this; we can compile Sass content with a task runner—such as Grunt, in a preprocessor, or directly from Sass. We will cover examples of all three, starting with taking a look at compiling directly from Sublime Text.

Compiling Sass files within Sublime Text

So far, we've added the syntax support for Sass within Sublime Text. To make Sublime Text really useful, we need to go one step further and add support to compile Sass code in a valid CSS file. Let's take a look at what is involved in getting it working, using the package available at `https://github.com/nathos/Sass-textmate-bundle/tree/sublime`:

1. Open the command panel by pressing *Ctrl + Shift + P* (Linux/Windows) or *Command + Shift + P* (OS X) and select **Package Control: Install Package**.

2. When the packages list appears, enter `SASS Build` and click on **SASS Build** to begin installing it:
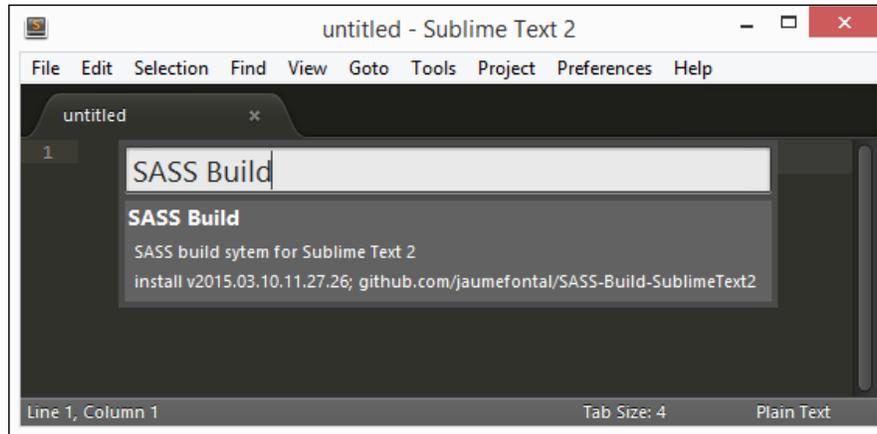
3. The installation is complete when the following screenshot is displayed on the status bar in Sublime Text:
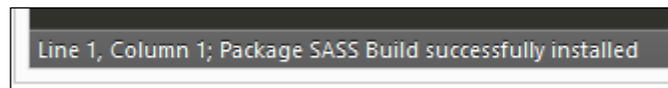
At this point, we can now compile Sass files directly from within Sublime Text. To do this, we need to press *Control + B* (Linux/Windows) or *Command + B* (OS X); to view the results, it's worth having Windows Explorer open with the Sass file listed. We can then check to see whether it has been updated once the compilation process is complete.

Removing installed packages

So far, we've installed support to compile within Sass that works, but this has raised an irritating issue. It displays a dialog box to verify that the compilation is complete. Once we've made a few changes, this becomes very irritating.

Thankfully, we can fix this—although we can edit the Python file behind the Sass Build plugin, my preference is to use Node and Grunt to provide the compilation process instead; we can tie in additional tasks, such as validating our CSS, for any potential errors.

> Node is an event-driven framework that, with Grunt (a task runner), allows you to automate repetitive tasks, such as compiling Sass, so that we can focus on developing our code. We can tie in additional tasks such as cleaning CSS code or applying vendor prefixes.

To remove the package that we've just installed, press *Ctrl + Shift + P* to bring up Package Control. Then, enter **Package Control: Remove Package** and select the **Sass Build** package from the list. You will likely see two packages there—we also need to remove this package too:
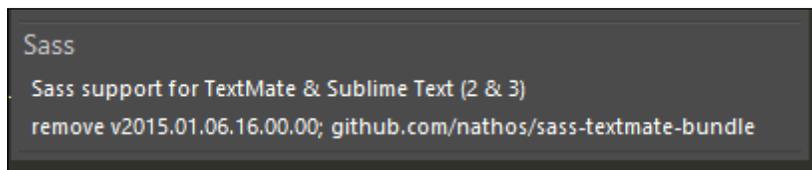
Once these packages have been removed, go ahead and restart Sublime Text. It's not always necessary, but it is good practice to do so when removing packages. Now that they are, it leaves us ready to install Node and Grunt, which we will do later in this chapter.

Using GUI-based compilers

Throughout this chapter, we've concentrated on using Sublime Text to both edit and compile our Sass code. Although this works perfectly well, it would be remiss to not at least cover some of the other options available to compile Sass.

Why would we need to use a separate application if we can do what we need to do directly from within Sublime Text? I hear you ask. Simple. You may already have a text editor that you don't want to change, but unfortunately, it doesn't come with the support to compile Sass. So, you need something to compile your code. Here are a few options to consider:

- **CodeKit**: This is a shareware applet from `http://incident57.com/codekit/`; this one is for Mac users only and the license for it is 32 USD.

- **Hammer**: This is available at `http://www.hammerformac.com`; this web development tool is for the Mac platform. It can be purchased from the Mac App Store for USD 23.99 and contains support to compile Sass files automatically.

- **Koala**: This is my personal favorite; this free application is available for Windows, Mac, and Linux platforms at `http://www.koala-app.com`.

- **Prepros**: This is another cross-platform app available at `https://prepros.io` for Windows and Mac platforms; this requires a license and is available for USD 29.

- **Scout**: This open source applet to compile Sass files is available at `http://mhs.github.io/scout-app/`. It comes in versions available for both the Windows and Mac platforms.

You may come across a couple of other tools once you have gained more experience with Sass:

- **LiveReload**: This is available at `http://livereload.com/`. It automatically refreshes the page when an action is completed and is now largely redundant.

- **Compass.app**: This is available at `http://compass.handlino.com/`. It is a style sheet authoring framework that has built-in support to work with Sass.

Let's change tack at this point. Although we will concentrate on using Sass from within Sublime Text throughout most of this book, it's still worth taking a moment to install and try out at least one of the preprocessors for Sass. With this in mind, let's take a look at installing and using the cross-platform open source application: Koala.

Installing Koala

Although we can easily write and compile our code within Sublime Text, there may be instances where we have to use a third-party application. For example, if we have to share the CSS files with other developers who are yet to experience the joys of using Sass. As we've already seen, there are a few options available. Let's take a look at installing one of these in the form of Koala:

1. We'll start by downloading the application. So head over to `http://www.koala-app.com` and click on the **Download** button from the main page of the site.

2. Click on the filename and then on **Download** at the bottom of the page to commence downloading.
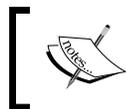
> For anyone using Mac or Linux, look for the **Other system versions** link below the download button on the main page of the site.

3. Double-click on the application once the download is complete to begin the installation; accepting all the default settings is fine for the purposes of this demo.

Once completed, we should have something akin to the following screenshot when double-clicking on the application icon. In this instance, it has already been configured to monitor a project folder that we will use later in this chapter:
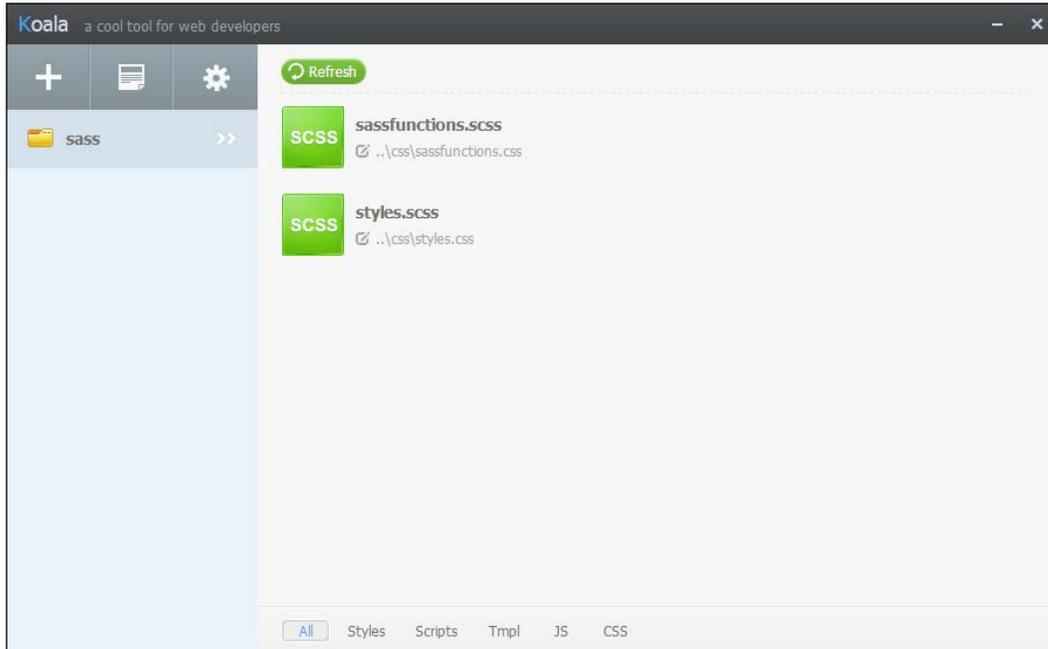
Okay, let's put our newfound knowledge to the test and compile some Sass code. For now, keep Koala open; we're about to use it in the next exercise.

Compiling Sass manually

As someone once said, "It's time..."

In the next couple of pages, we will take an initial look at how to compile Sass in practice; don't worry if it doesn't mean much now because all will become clearer later in the book. For now, let's concentrate on familiarizing ourselves with compiling Sass in anger. For this demo, we will simply render a number of statements of varying sizes from `<h1>` right down to `<h5>` on screen; we'll work out the size automatically before rendering it on screen.

Let's make a start with the following steps:

1. From the code download that accompanies this book, extract a copy
   of `sassfunctions.html` and `sassfunctions.scss`. Then, place the
   HTML markup file at the root of our project folder and the Sass file
   in the `sass` subfolder.

2. Switch to Koala (if you've not already closed it). Then, position it onscreen
   so that you have it and Windows Explorer both showing onscreen; the latter
   must show the list of subfolders in the project folder area.

3. Drag and drop the `sass` folder to Koala so that it displays the contents of the
   `sass` folder.

4. Open up a copy of `sassfunctions.scss` from our project area. Go ahead
   and change the greeting to `Hi Chrome users` in place of Firefox:

   ```
   $firefoxGreeting: "Hi Chrome users! This is a...";
   $mainColour: #845;
   $h1Size: 4em;
   $removeMargin: -20px;
   ```

5. In Windows Explorer, check the details of our Sass file. If all is well, the
   greeting will have been updated with the same dates and times showing on
   the `.scss`, `.map`, and `.css` files (the latter two in the `css` folder).

6. Open up a copy of `sassfunctions.css` in Sublime Text; we should see
   something similar to this screenshot:
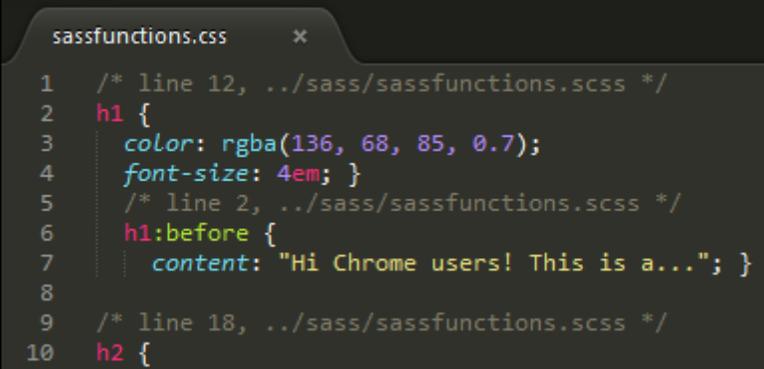
```
sassfunctions.css        ✕

1    /* line 12, ../sass/sassfunctions.scss */
2    h1 {
3      color: rgba(136, 68, 85, 0.7);
4      font-size: 4em; }
5      /* line 2, ../sass/sassfunctions.scss */
6      h1:before {
7        content: "Hi Chrome users! This is a..."; }
8
9    /* line 18, ../sass/sassfunctions.scss */
10   h2 {
```

We've now successfully compiled our first Sass file; any subsequent changes we make to the file will automatically be picked up and recompiled by Koala. Onwards we go; we've got Sass installed and it compiles fine, but there's still something wrong.

Ah! Yes. I know. In this age of automation, why are we spending time completing tasks manually using a separate piece of software if we're already using packages, such as Node and Grunt, elsewhere in our normal daily work? It doesn't make sense to introduce yet another application if we're already using Node or Grunt. Instead, we can easily automate them. I'll show you how as part of the next exercise.

It's worth noting that if the file hadn't compiled correctly, then it will still produce a CSS file, but instead, contain details of the error.

Using Node and Grunt

A frequent drawback of using Sass, until recently, was the need to always have to use an additional piece of software to compile our code. Not any more; as many developers already use the Node.js platform, it makes sense for those who do use it to take advantage of its power to use it to perform the compilation work for us.

Thankfully, it's really easy to make it happen, although it involves a slight change in how we process our Sass files. We can then couple this with the `grunt-contrib-sass` plugin available at `https://github.com/gruntjs/grunt-contrib-sass` and use it to compile our code.

Let's take a look and see what is involved:

1. The first step in our process is to install Node. For this, head over to `http://www.nodejs.org` and click on the version appropriate for your platform. Double-click on the installer to begin the process, accepting all defaults.

2. Now that Node is installed, extract copies of the `package.json` and `gruntfile.js` files from the code download that accompanies this book and then save them to the project area.

3. We need to set Grunt so that it is available from the command line. To do this, go ahead and run the following command from the Node.js command prompt:

   npm install -g grunt-cli

4. Next, fire up the Node.js command prompt. Then, change the location to our project area and enter this command:

   npm install grunt-contrib-sass --save-dev

   Node will go away and install all three packages. Once completed, we will see a confirmation similar to the following screenshot:
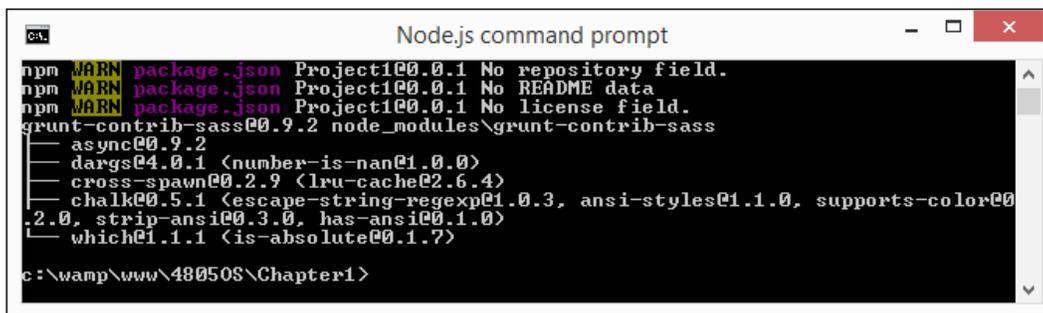
5. At this stage, we're ready to run Grunt. So, go ahead and enter this command at the prompt:

   grunt

If all is well, Grunt will display the following message:

Node is now configured to automatically watch for any changes and get Sass to recompile the .scss files in valid CSS when we make a change to the original source code.

Testing our installation

Now that we have our automatic compilation process set up, lets put it to the test:

1. We'll start by extracting a copy of `styles.scss` from the code download and saving it to the project area.

2. Next, make a note of the date and time in the `Date modified` column of Windows Explorer. We'll use this to prove that Sass has updated our file.

3. Open up `styles.scss` and look for this code block at or around lines 7-10:

```
 7    $firefoxGreeting: "Hi Firefox users! This is a...";
 8    $mainColour: #845;
 9    $h1Size: 4em;
10    $removeMargin: -20px;
```

4. Change the text of `$firefoxGreeting`, as shown in the following command (for now, don't worry too much about what this means; all this will become clear in the next chapter):

   ```
   $firefoxGreeting: "Hi Google users! This is a….";
   ```

5. Save the change. As soon as it detects a change, it will automatically recompile the code and produce valid CSS:
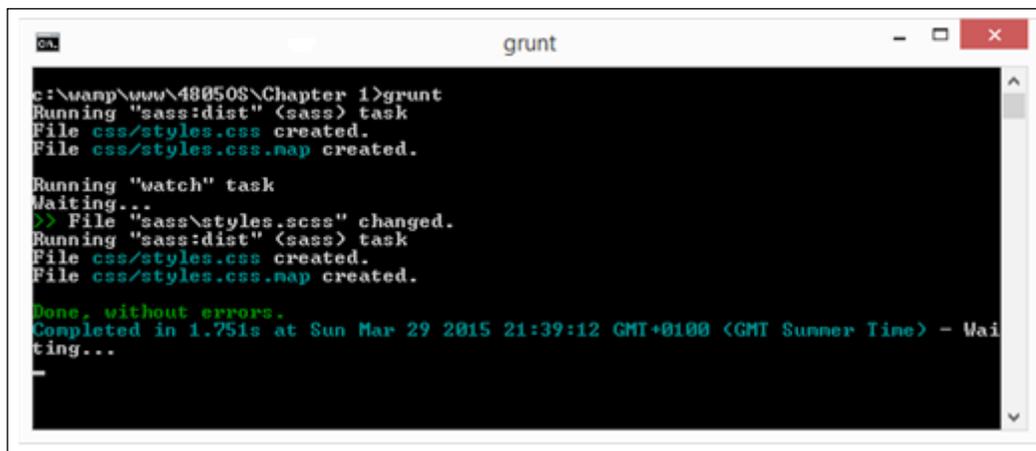
```
c:\wamp\www\48050S\Chapter 1>grunt
Running "sass:dist" (sass) task
File css/styles.css created.
File css/styles.css.map created.

Running "watch" task
Waiting...
>> File "sass\styles.scss" changed.
Running "sass:dist" (sass) task
File css/styles.css created.
File css/styles.css.map created.

Done, without errors.
Completed in 1.751s at Sun Mar 29 2015 21:39:12 GMT+0100 (GMT Summer Time) - Wai
ting...
```

We can now set this running in the background. So, any future changes we make to this file will be compiled automatically.

If you prefer to simply use Node and not Grunt, then the same plugin is available for Node only at `https://github.com/sass/node-sass`; it performs the same function as `grunt-contrib-sass`.

Okay, so we've set up a process that will take care of compiling our code automatically. Now, it's worth spending a little time to go through what we've set up because there are a few key points we need to cover, which will help with future projects. We'll take a look at the plugins that we've used to create our automated system, starting with `grunt-sass`.

Exploring the compilation process

If you're new to working with Node and Grunt, then it is worth spending a little time going through what we created in the last exercise; it contains some useful tricks to help remove some of the tedium behind making changes to Sass code.

The two key elements are `package.json` and `gruntfile.js`; `package.json` is used by Node's package manager to help manage dependencies for any project; `gruntfile.js` details the tasks that should be run with Grunt. Let's take a look at `package.json` first.

This is how the `package.json` file for our project looks:

```
{
  "name": "Project1",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "0.4.5",
    "grunt-contrib-watch": "~0.6.1",
    "grunt-sass": "^0.16.0"
  }
}
```

It's a simple affair. We first name the project and give it an appropriate version number. Then, we state the dependencies for the project. In this instance, we will use Grunt (`http://www.gruntjs.com`), a watcher (`https://github.com/gruntjs/grunt-contrib-watch`), and the Sass plugin for Grunt (`https://github.com/gruntjs/grunt-contrib-sass`).

The key to this file is that once it has been created, we can install Grunt, `grunt-contrib-watch`, and `grunt-contrib-sass`, using this command from the project folder:

`npm install`

A key point related to this is the use of `grunt-contrib-watch`; this has a facility to automatically reload your code in the event of a change. This makes using any external application, such as LiveReload, surplus to requirements.

> Details of how to set up `grunt-contrib-watch` are available at `https://github.com/gruntjs/grunt-contrib-watch#optionslivereload`.

The second key part of this demo is the `gruntfile.js` file, which contains the steps required for Grunt so that it knows what to do once it has been invoked. This is a little more involved, so let's break it down into sections, beginning with the initialization and fetching of details from the `package.json` file:

```
module.exports = function(grunt) {
  grunt.initConfig({
```

Next up, comes our watcher; it watches for any changes to our Sass files and invokes the Sass task at the appropriate moment, as shown in the following code:

```
watch: {
  sass: {
    files: ['sass/**/*.{scss,sass}','sass/_partials/**/*.
      {scss,sass}'],
    asks: ['sass:dist']
  },
```

In `livereload`, we will watch for any changes to HTML, JavaScript, CSS, or image files and force the browser to refresh if it detects that a change has been made:

```
  livereload: {
    files: ['*.html', 'js/**/*.{js,json}',
      'css/*.css','img/**/*.{png,jpg,jpeg,gif,webp,svg}'],
    options: {
      livereload: true
    }
  }
},
```

This is the most important part of `gruntfile.js`. This task covers the compilation of Sass files in valid CSS. It is set to produce source files (more anon) with CSS being compressed. At present, it is set to only compile if changes are made to `sassfunctions.scss`, but this can easily be changed to a wildcard entry:

```
sass: {
  options: {
    sourceMap: true,
    outputStyle: 'compressed'
  },
  dist: {
    files: {
```

```
            'css/sassfunctions.css': 'sass/sassfunctions.scss'
          }
        }
      }
    });
```

Last but by no means least, we register the default task and set Grunt to load the tasks when this is called:

```
    grunt.registerTask('default', ['sass:dist', 'watch']);
    grunt.loadNpmTasks('grunt-contrib-sass');
    grunt.loadNpmTasks('grunt-contrib-watch');
};
```

> We can use `grunt` or `grunt default` at Command Prompt; either will call the Grunt tasks for this project.

Right, okay. Our Sass is now compiling. Now is a good opportunity to take a quick look at a useful feature, which will help with debugging CSS produced from Sass files. Source maps can be a little fiddly to get working, but thankfully, there is a really easy option we can use, which we included in the `gruntfile.js` file that we just used to compile Sass. Let's delve in and learn more about using it to produce our source map.

Using source maps

Source maps are a useful tool to help us troubleshoot issues with CSS styles by providing a reference back to the source line in our code.

They have had something of a chequered past; the most high-profile instance of it not working is likely to be within jQuery itself. Automatic support was added a few years ago, but this was subsequently changed to become manual due to issues with generating the source map file.

Thankfully, they are a cinch to generate within Sass; most modern browsers of at least 1-2 years old will have support available (either by default or once it has been switched on).

The beauty of the Grunt file that we just created is that we've already set Grunt to create a source map for us. This is shown in the highlighted line:

```
options: {
  sourceMap: true,
  outputStyle: 'compressed'
},
```

There are more options available at `https://github.com/sass/node-sass#sourcemap` that we can configure, but this should be sufficient to get us started.

Once the Sass file is compiled, we'll see an addition at the end of the compiled CSS file, which points to the source map:
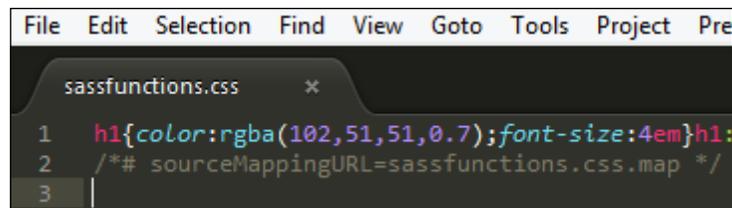
If we open up `sassfunctions.css.map`, we should find this:

```
{
  "version": 3,
  "mappings": "AAWA,EAAG;EAEF,KAAK,EAAE,sBAAgC;EACvC,SAAS,EAND,
  GAAG;EAPV,SAAoB;IAClB,OAAO,EAIQ,iCAAiC;;AAWpD,EAAG;EAEF,
  KAAK,EAAE,OAAwB;EAC/B,UAAU,EAXI,KAAK;EAYnB,SAAS,EAAE,KAAY;
  EApBtB,SAAoB;IAClB,OAAO,EAIQ,iCAAiC;;AAkBpD,EAAG;EAEF,KAAK,
  EAAE,OAAyB;EAChC,UAAU,EAlBI,KAAK;EAmBnB,SAAS,EAAE,KAAY;EA3BtB,
  SAAoB;IAClB,OAAO,EAIQ,iCAAiC;;AAyBpD,EAAG;EAEF,KAAK,EAAE,OAAyB;
  EAChC,UAAU,EAzBI,KAAK;EA0BnB,SAAS,EAAE,KAAY;EAlCtB,SAAoB;
  IAClB,OAAO,EAIQ,iCAAiC;;AAgCpD,EAAG;EAEF,KAAK,EAAE,OAAyB;EAChC,
  UAAU,EAhCI,KAAK;EAiCnB,SAAS,EAAE,KAAY;EAzCtB,SAAoB;IAClB,OAAO,
  EAIQ,iCAAiC",
  "sources": ["sassfunctions.scss"],
  "names": [],
  "file": "sassfunctions.css"
}
```

It won't mean much, but it contains references to where we can find the original sources in `sassfunctions.scss` and how they translate to the compiled versions in `sassfunctions.css`. If we take a look at the output of `sassfunctions.html` in a DOM inspector, we would normally expect to see the line reference in a SCSS file, as shown in this screenshot:

However, if we right-click on the rule, and select **Show original sources**:
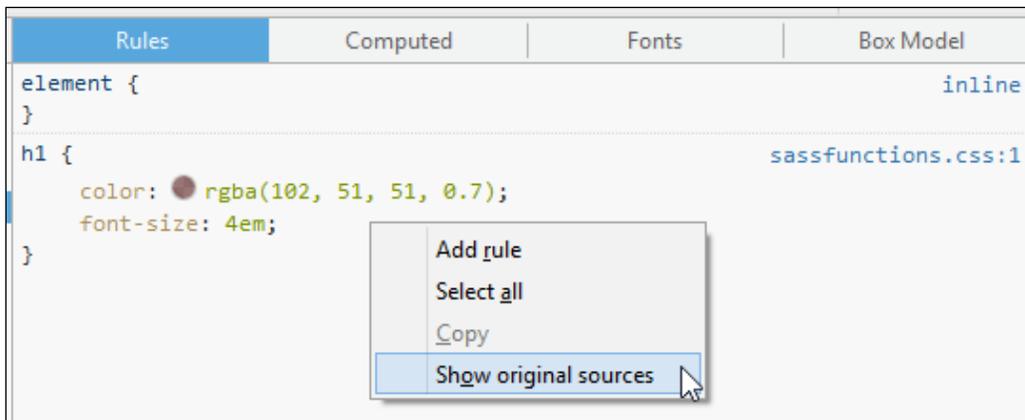
The inspector will replace the CSS reference with the equivalent one from our Sass source:

The beauty of this is that if we click on the **sassfunctions.scss:12** line to the right, it takes us to the start of the rule in the Sass file so that we can see the source rule being used to create our styles.

> You may find that you need to set the output style to `Expanded` for this to work correctly. Once the code has been finalized, it should be set to `Compressed` for production use.

Let's put some of what we've covered throughout this chapter to good use and create a simple mock-up of a web page that uses some basic Sass-driven styles. It won't win any awards, but it will be the perfect introduction to what is coming later in the book.

Creating a simple layout

In the next few pages, we'll mock-up a simple web page with a side menu. This is to illustrate how easy it is to compile the code and produce valid CSS.

The demo isn't fully functional, but it can be easily used as a basis for something more complex. At this stage, we'll concentrate more on how we compile the code. Before we begin adding markup, we need to make a change to our compilation process, so let's sort this now.

Setting up our compilation process

Remember earlier in the chapter where we set up Node and Grunt to compile our CSS? Let's revisit that functionality now; we will use it to compile the code for our demo, but set it to compile a specific file in the previous demo. We need to alter it to watch for changes in our file for this demo, so let's fix this now. Perform the following steps:

1. Open up a copy of `gruntfile.js` from our project area and look for the values highlighted in the following code:

   ```
   dist: {
      files: {'css/sassfunctions.css': 'sass/sassfunctions.scss'}
   }
   ```

2. We need to change it to allow automatic compilation of our Sass code for the demo page, so alter it as follows:

   ```
   dist: {
      files: { 'css/demopage.css': 'sass/demopage.scss' }
   }
   ```

3. Save the changes and close the file. Then, bring up the Node.js command prompt and change the location to our project area.

4. At the prompt, enter this:

```
grunt
```

5. Minimize the window; we need it running in the background, but it doesn't have to be fully visible on screen.

We're now ready to add our CSS styles and compile them automatically with Grunt. Let's go ahead and set up the markup and Sass rules for our demo. Once completed, we will end up with a demo page similar to this screenshot:

Preparing the markup and styles

We're now ready to set up our demo page, so let's get started with adding the markup. Perform the following steps:

1. We'll start by extracting the `demopage.html` file from the code download that accompanies this book and saving it to the root of our project area. We also need the `header.jpg` image. Go ahead and drop this in the `img` folder in our project area.

2. Next up, we need to add our style sheet. So, go ahead and create a new file called `demopage.scss`. Save this to the `sass` subfolder in our project area.

3. In `pagedemo.scss`, go ahead and add the following styles. Don't worry too much about what they all do in detail; for now, let's concentrate on how to compile the styles to produce valid CSS. Let's start with setting some values:

    ```
    $baseWidth: 800px;
    $mainWidth: round($baseWidth / 1.618);
    $sidebarWidth: round($baseWidth * 0.382);
    ```

4. We now need to set some color variables. Sass will automatically switch the appropriate color values at compilation; it makes our code easier to read:

    ```
    $black: rgb(0, 0, 0);
    $white: rgb(255, 255, 255);
    $light-gray: rgb(204, 204, 204);
    $dark-moderate-blue: rgb(62,122,144);
    $very-soft-orange: rgb(232, 210, 150);
    ```

5. This block of code calculates the font size to use for various elements on our page with `rem` units:

    ```
    @function calculateRem($size) {
      $remSize: $size / 16px;
      @return $remSize * 1rem;
    }
    ```

6. Next up, we will sort out the relevant font sizes to use for elements on our page:

    ```
    @mixin font-size($size) {
      font-size: $size;
      font-size: calculateRem($size);
    }
    ```

7. The next six styles take care of the relevant sections of our page, beginning with the main container:

```
body {
  max-width: $baseWidth;
  padding: 0.3125rem;
  border: 0.0625rem solid black;
  margin: 5% auto;
  border-radius: 0.25rem;
  box-shadow: 0.25rem 0.25rem 0.25rem 0 rgba($black, .5);
  font-family: 'Kite One', sans-serif;
}
```

8. Then, comes the title text for the header:

```
h1 {
  @include font-size(32px);
  color: #FFFFFF;
  font-weight: 400;
  padding: 1.563rem 0 0 6.25rem;
  position: absolute;
}
```

9. We can't have title text without some form of background, so let's add this now:

```
header {
  @include font-size(18px);
  height: 8.125rem;
  background: url("../img/header.jpg") $dark-moderate-blue;
}
```

10. We have a title, but a page is nothing without content, so let's fix this by adding this rule:

```
section {
  width: $mainWidth;
  float: left;
  box-sizing: border-box;
  height: 35.94rem;
  padding: 0.625rem;
}
```

11. We need to finish off the page with a footer. This includes the styling for the social images shown on the right-hand side of the footer:

```
footer {
  @include font-size(12px);
  border-top: 0.0625rem solid $light-gray;
  clear: both;
  height: 4.063rem;
  padding: 0.3125rem 0 0;
  img {
    background: url('../img/sprites.png') $white;
    background-repeat: no-repeat;
    width: 10.94rem;
    height: 3.75rem;
    float: right;
  }
}
```

12. Last but by no means least, we need something in our navigation. So, let's take care of this now. First with the container, followed by the styles used within navigation entries, as shown in the following code:

```
nav {
  width: $sidebarWidth;
  border-right: 0.0625rem solid $light-gray;
  float: left;
  box-sizing: border-box;
  height: 35.94rem;
  margin: 0.3125rem 0;
  li { list-style: none; margin: 0.3125rem; }
  a {
    text-decoration: none;
    &:hover { background-color: $very-soft-orange; padding:
      0.3125rem; transition: .5s ease; }
  }
}
```

13. At this point, our Sass code is ready to be compiled, so let's go ahead and save `demopage.scss`; if all is well, we should see newly compiled CSS and source map files, as shown in this screenshot:
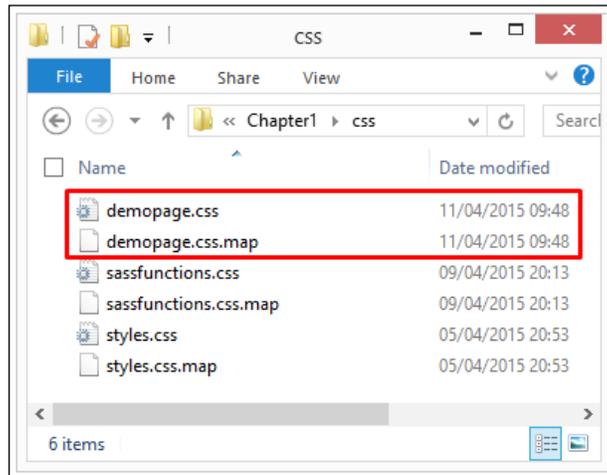
14. To prove that we have valid CSS, a look at `demopage.css` will show something similar to this screenshot:

15. Go ahead and preview the results in a browser. If all is well, we should have something akin to the screenshot at the start of this exercise.

Dissecting the Sass styles

Throughout the demo, you may have noticed that we introduced a few Sass techniques; it's worth exploring these now. To give you a flavor of what to expect later in the book, let's first take a look at these lines:

```
$baseWidth: 800px;
$mainWidth: round($baseWidth / 1.618);
$sidebarWidth: round($baseWidth * 0.382);
```

Here, we've implemented three **variables**. We will start with setting a value for `$baseWidth` to 800 pixels; the second and third variables are implemented as fraction values of `$baseWidth` before being rounded down to the nearest integer.

> The values used in the calculations make up the **Golden Ratio** – this ratio was designed to produce an aesthetically pleasing effect; in this instance, we use it to produce a layout that balances content space in proportion to the navigation. For more details, head over to Wikipedia, at `https://en.wikipedia.org/wiki/Golden_ratio`.

Next comes this little function, which we will use to calculate font sizes in `rems`:

```
@function calculateRem($size) {
  $remSize: $size / 16px;
  @return $remSize * 1rem;
}
```

We will use it in this reusable block of code, known as a **mixin**, to translate pixel-based font sizes; it will render font sizes in `rem` values, falling back to pixels if `rem` values are not supported:

```
@mixin font-size($size) {
  font-size: $size;
  font-size: calculateRem($size);
}
```

Moving on, we introduced our discussion with a look at the variables created at the beginning of our code; now, it's time to see them in use. We've used them in several places, of which this is a perfect example:

```
#content {
  width: $mainWidth;
  ...
}
```

A quick look at the DOM Inspector will show you how the variable has been translated into valid CSS:

The final technique that we will take a quick look at is that of **nesting**—how often have you written code that refers to the same (or similar) selectors, which may be two or three levels deep? Instead, we can reduce the code we need to write by grouping similar (or identical) styles together, as shown in the following code:

```scss
footer {
  @include font-size(12px);
  border-top: 0.0625rem solid $light-gray;
  clear: both;
  height: 4.063rem;
  padding: 0.3125rem 0 0;
  img {
    background: url('../img/sprites.png') $white;
    background-repeat: no-repeat;
    width: 10.94rem;
    height: 3.75rem;
    float: right;
  }
}
```

We can see how this translates into valid CSS from this screenshot:

We'll explore these and more throughout the book; the key though to getting started with Sass is to keep it simple. Once your compilation process is set up and working, it is just a matter of writing code and letting Node/Grunt do the rest!!

Summary

Phew! We've covered a lot in our introduction to working with Sass, so let's take a moment to cover what you've learned.

We kicked off with a look at how to install Sass before adding support for Sass to text editors using Sublime Text as an example; this includes both syntax support and an option to compile Sass files directly from the editor. We also looked at how to remove the build package before switching to using Node and Grunt in order to compile our code automatically.

Next, we discussed how to compile Sass code using Koala, a third-party application. We understood how to install it for use and covered a number of other alternatives that we can explore at a later date if desired. We finished with how to add support for source maps in order to help with troubleshooting Sass code. We then rounded out the chapter with a demo, creating a simple web page that used some of the techniques that will be explored throughout the book.

Ready to learn more? The next step in this awesome journey is to take a look at how to work with variables and mixins, where we can explore how easy it is to create reusable blocks of code for current and future projects.

2
Creating Variables and Mixins

Imagine that classic scenario if you will, where some well-meaning soul says that they want a change made to a proposed style for a website and they want it tomorrow. Oh! And it's 6 o'clock on a Friday…

Sound familiar? How many times have you had to style an element only to have to go and change it to something else? It's a real pain, right?

A key part of Sass is getting familiar with setting variables; they allow you to define values in one place that can be used many times in our CSS output. Mixins go one step further—we can create reusable blocks of code that can be dropped in with care and planning. It saves us valuable time at the compilation stage.

Throughout this chapter, we will cover a number of tricks that will help reduce or remove the need to update elements individually, which include:

- Adding comments to Sass files
- Creating variables
- Building mixins and incorporating them into our pages
- Using prebuilt mixin libraries, such as Compass
- Adapting existing mixins to use libraries
- Building custom mixin libraries

Time to level up. Let's go!

Making a start on our code

We started this book by preparing our environment—now that everything is installed, we can look at writing some code.

Of course, we can jump in, but as you are likely to be relatively new to working with Sass, it's a wise idea to get as much help as possible. One way to achieve this is through the use of **linting** code or running a continual background check to ensure that we are writing valid Sass code.

The great thing about this is that we can easily set this up with the addition of Ruby gems. These are the Ruby equivalent to plugins and work in a similar fashion to using standard plugins such as those we may use when working with jQuery. The plugin we will use is `scss-lint`, which is available at `https://github.com/brigade/scss-lint`. Let's take a look at how to set this up from within Sublime Text, using the SublimeLinter SCSS plugin from `https://github.com/attenzione/sublimeLinter-scss-lint`.

> This exercise assumes that you have Package Control and Sass support for Sublime Text 3 installed along with Node.js. For details on how, refer to *Chapter 1*, *Introducing Sass*, before continuing with the exercise. Note that the linter plugin does *not* work on Sublime Text 2.

Let's start. Perform the following steps:

1. We'll start by installing the `scss-lint` gem. To do this, fire up Command Prompt. Then, enter the following command:

   ```
   gem install scss-lint
   ```

2. Ruby will go away and fetch the package and its dependencies. When completed, it will display a message confirming how many gems have been installed:

3. Next, we need to install the SublimeLinter plugin. This provides the framework to lint our code. Go ahead and fire up Sublime Text. Then, press *Ctrl + Shift + P* to display the command palette.

> The `scss-lint` gem must be installed first; this is a dependency for the Sublime Text linter plugin to work.

4. In the textbox, enter `install` and select **Package Control: Install Package** from the list:

install

Package Control: **Install** Package
Package Control: Advanced **Install** Package
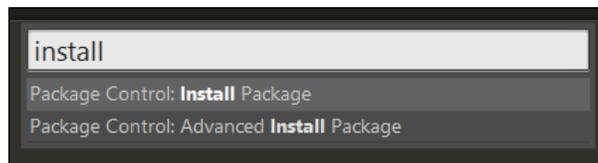
5. Sublime Text will retrieve the list of available packages. In the drop-down box, enter `SublimeLinter` to display the item in the list. Then, click on it to install the plugin:

SublimeLinter

SublimeLinter
Interactive code linting framework for Sublime Text 3
install v3.4.39; sublimelinter.com

SublimeLinter-annotations
SublimeLinter 3 plugin that marks annotations such as TODO, FIXME, etc.
install v1.0.8; github.com/SublimeLinter/SublimeLinter-annotations
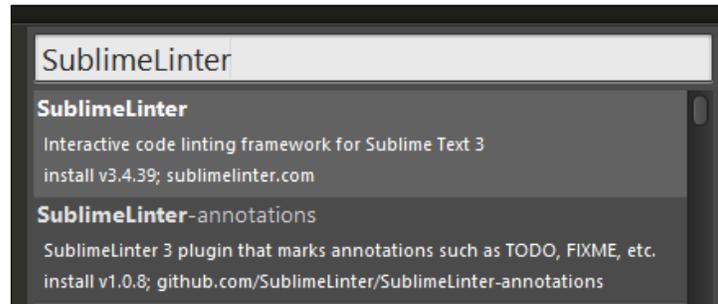
6. When installed, SublimeLinter will display a message similar to the following screenshot:
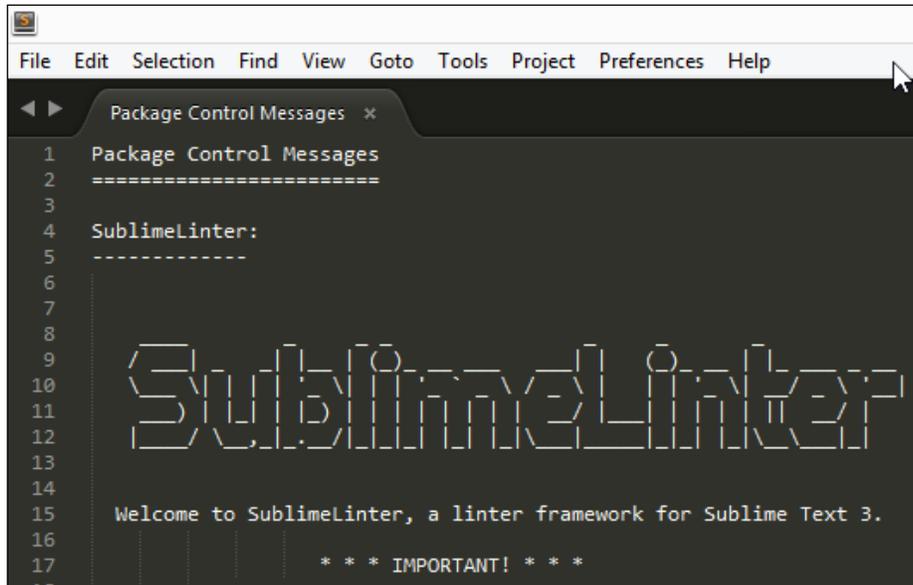
7. Repeat steps 4 and 5, but this time, enter `linter jshint` and click on **SublimeLinter-jshint** to install the SublimeLinter JSHint plugin.

8. Repeat steps 4 and 5 once more, but this time, enter `linter scss` and click on **SublimeLinter-contrib-scss-lint** to install the SublimeLinter SCSS plugin:
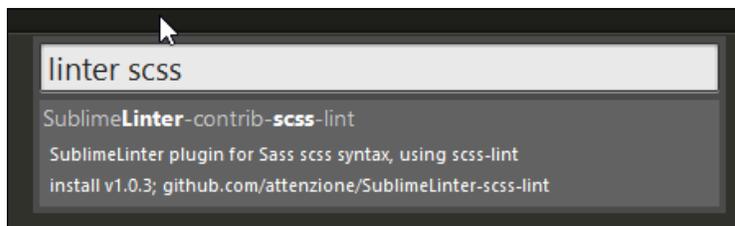
SublimeLinter is now installed. We've set Sublime Text to automatically check our code for any potential errors and alert us in the event of an issue with our code. To make it really effective though, we need to make a few changes:

1. First, bring up the command palette as before (or by pressing *Shift + Ctrl + P* for PC/Linux or *Command + Ctrl +P* for Mac). Then, enter `gutter` and select **SublimeLinter: Choose Gutter Theme** from the list.

2. From the list that appears, choose **Knob – Symbol** to change the icons that show in the left-hand side margin when there is an issue in the code.

3. We will also alter the style used to mark our code if there is an issue; fire up the command palette as before. Then, enter `mark` to select **SublimeLinter: Choose Mark Style** and select **Squiggly underline** from the list.

4. We can now check to see whether it works. Restart Sublime Text. Then, open up the `sassfunctions.scss` file from *Chapter 1, Introducing Sass*.

5. Press *Ctrl + S* to save. While we haven't changed our code, it's enough to trigger the linter to kick in, as shown in the following code:

```
 7   $firefoxGreeting: "Hi Firefox users! This is a...";
 8   $mainColour: #845;
 9   $h1Size: 4em;
10   $removeMargin: -20px;
```

6. Click on `$mainColour`. If all is well, we should see a message appearing in the status bar, which is similar to this screenshot:

```
3 of 38 errors: NameFormat: Name of variable `mainColour` should be written in all lowercase
```
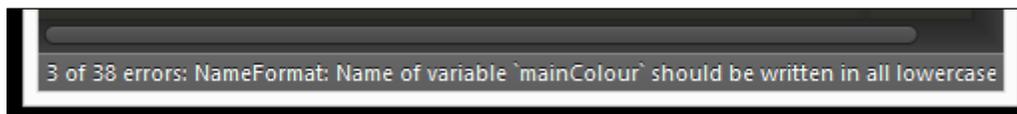
> The error shown isn't really an error as such; variables can be written using camelCase if desired. It's more about maintaining a consistent style in your code.

Okay. Now, we have our helping hand in place. We're now ready to start writing code; we'll begin with creating variables to help with assigning values in our code.

Altering the compilation process

Before we start creating code, we can benefit from a change to our compilation process. At present, our Grunt file is compiling a hardcoded file, which is a little limiting. Instead, it would be better to set it to compile multiple files while working through the book. This removes the need to change the names, and deals with compiling files without our intervention.

Let's go ahead and make this change now:

1. Open up the copy of `gruntfile.js` from *Chapter 1*, *Introducing Sass*, and alter the files block under `dist:`, as shown in the following code:

```
dist: {
  files: [
    {
      expand: true, // Recursive
      cwd: "sass", // The startup directory
      src: ["**/*.scss"], // Source files
      dest: "css", // Destination
      ext: ".css" // File extension
    }
  ]
}
```

> There is a completed copy of `gruntfile.js` in the code download if you get stuck editing the original version.

2. Delete the files in the `css` folder. This is to verify that they have indeed been created when we rerun the process.

3. If the Grunt process is still running, then stop and restart it. If all is well, we should see it recreate the missing CSS and map files. Note that they won't be listed, but a check of the timestamp on each file in the `css` folder will confirm whether or not they have been recreated, as shown in the following screenshot:
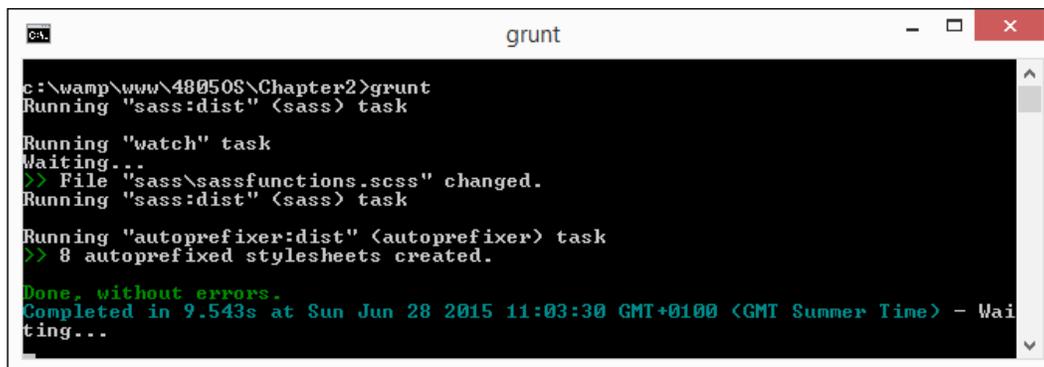
Right! Now we can get on with writing some code.

> You may have noticed the `style` value that is set to `expanded`. If we change this to `compressed`, then it will compress each CSS file that is created. There is also an autoprefixer task included in this Grunt file; we will cover this in more detail later in the *Animating content using Sass* section.

Creating variables

Let's stay with the `sassfunctions.scss` file that is open from the previous exercise. If you take a careful look at lines 7 to 10, you will see this:

```
3    $firefoxGreeting: "Hi Chrome users! This is a....";
4    $mainColour: #458;
5    $h1Size: 4em;
6    $removeMargin: -20px;
```

These are examples of **variables** or placeholders for values within our code. These work in a similar fashion to JavaScript libraries, such as jQuery. To see how they work, let's go through the following steps:

1. Fire up a Node.js command prompt and change the location to our project area. In this command prompt, enter the following code and click on *Enter*:

   grunt

2. Grunt is now ready and waiting to compile.

3. Switch back to Sublime Text. Then, change the `$mainColor` value at line 8 to:

   `$mainColour: #321`

   Grunt will kick in and compile the file. If we open it up, we can see that it has produced a valid CSS file, as shown in this screenshot:

```
1    /* line 8, ../sass/sassfunctions.scss */
2    h1 {
3      color: rgba(51, 34, 17, 0.7);
4      font-size: 4em;
5    }
6    /* line 2, ../sass/_partials/_content.scss */
7    h1:before {
8      content: "Hi Chrome users! This is a....";
9    }
```

Exploring what happened

At this point, we have a valid CSS file, where values have replaced each instance of the relevant variable. If we continue to make changes to it, Grunt will automatically recompile these changes and update the existing CSS file. It's worth exploring a couple of key points about the code that we've produced:

- Sass will convert colors to the RGBA format during compilation. You will note that we have rgba(51, 34, 17, 0.7) shown in the code; this is the RGB equivalent of #321, but is displayed in the RGBA format as a result of setting the additional alpha value of 70%.

> We can refer to a site (such as http://www.rgbtohex.net/) to confirm whether the RGBA values displayed in our code do indeed match those shown in the compiled CSS.

- You will notice that we have comments displayed in our code. Sass uses the /* */ or // format to display comments; we will delve into this later in the *Adding comments* section in this chapter.

Variables can be used to store any value. Once created, they will replace the instance with that value. Variables are controlled by scope, which limits where they can be used; in most cases, it makes sense to define them at the top of the style sheet or even in a separate imported file. Even though the file may be large, Sass will only use those variables that are explicitly referenced within our code.

> For more details on how scope limits the use of variables, refer to http://sass-lang.com/documentation/ file.SASS_REFERENCE.html#variables_.

It is worth getting to know the different data types available in Sass. In brief, the following types are available:

- Numbers (for example, 1.2, 13, and 10px)
- Strings of text, with and without quotes (for example, "foo", 'bar', and baz)
- Colors (for example, blue, #04a3f9, and rgba(255, 0, 0, and 0.5))
- Booleans (for example, true, false)
- Null values (for example, null)

- Lists of values separated by spaces or commas (for example, `1.5em 1em 0 2em`, `Helvetica`, `Arial`, and `sans-serif`)

- Maps from one value to another (for example, `(key1: value1, key2 value2)`)

> I strongly recommend that you carefully read the main documentation on the Sass website at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#data_types`. This is a common source of problem for those getting to know Sass!

For now, let's switch to one use of variables, where it is important to get it right—that of assigning colors.

Using variables to create colors

So far, we've seen how easy it is to create variables. As long as the variable name is preceded with a `$` symbol, we can create any number of variables that can be reused in current or future projects. For now, let's turn our attention to creating colors. This is one area where variables come into their own.

Imagine that you have a block of code where colors are being used. The typical thing to do is to assign them to a style sheet, which is similar to this:

```
#box1 { background-color: #327DA0 }
#box2 { background-color: #a03246 }
#box3 { background-color: #a08c32 }
```

…and so on.

Seems pretty reasonable, right? If we wanted to change a particular shade of color to another value (say, make it lighter), we would go through it and change every instance in our code, right?

You can, but I won't. That's too much like hard work! Instead, let's use Sass to better manage our colors. The sensible way to do this is through a two-step process:

1. First, let's assign proper names to our variables. Instead of trying to decipher hex codes or even RGB(A) values, we will know that `#327DA0` is dark blue, for example:

```
$darkBlue: #327DA0;
$darkRed: #a03246;
$darkYellow: #a08c32;
```

2. Next, let's assign these color names to variables. One way of doing this is from Jim Nielsen, the designer, who recommends following the generic-specific principle, where we use functional names followed by descriptive names:
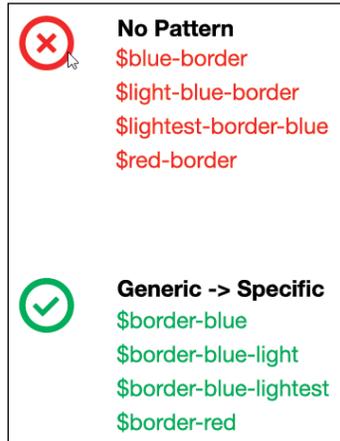
Let's see how this makes sense and avoids any issues with changing color values. We can group all the generic items together (in this case, $border), follow it by the color (blue), and tack on any additional descriptors, such as light or dark.

The key though is to remain consistent. Throughout this book, we will use camelCase, but we could equally use hyphenated names if we wanted to; it is up to you to decide what format suits your needs.

> For more details on this useful tip by Jim Nielsen, head over to
> `http://webdesign.tutsplus.com/articles/quick-tip-name-your-sass-variables-modularly--webdesign-13364`.

Using this principle means that we need to change the color for a specific category of link, for example, we only need to change it in one place and recompile our code. It has the added bonus of avoiding any issues where the same link color variable is reused in different scenarios such as this:

```
$colorText: #333333;
$colorLink: #001eff;

.myClass {
  color: $colorText;
  border-color: $colorLink;
```

```
}

a {
  color: $colorLink;
}
```

In this instance, we've reused the `$colorLink` variable in two different rules. If we wanted to change the border color, and *not* the link color, then we would've created a problem.

Let's put this into practice with a quick and easy demo. We'll set up a row of simple boxes and assign colors using variables:

1. We start with extracting copies of `colorvariables.html`. Go ahead and save this in the root of your project area.

2. In a new file, add the following Sass code. This takes care of styling our boxes, starting with the creation of six color variables:

    ```
    $darkBlue: #327DA0;
    $darkRed: #a03246;
    $darkYellow: #a08c32;
    $darkPink: #a0327d;
    $darkOrange: #a05532;
    $darkCyan: #32a08c;
    $white: #fff;
    ```

3. Next comes the assignment of these color variables to the box colors:

    ```
    $colorBox1: $darkBlue;
    $colorBox2: $darkRed;
    $colorBox3: $darkYellow;
    $colorBox4: $darkPink;
    $colorBox5: $darkOrange;
    $colorBox6: $darkCyan;
    ```

4. Then comes the generic styling for all six boxes, where we set the size and position on screen:

    ```
    body {
      font: 1.2rem bold
      Helvetica, sans-serif;
      color: $white;
    }

    div {
      height: 5rem;
      width: 5rem;
    ```

```
    border: 0.1rem solid $black;
    display: inline-block;
    margin: 1rem;
    padding-top: 1.0rem;
    padding-left: 1.5rem;
}
```

5. We then assign the relevant box color variable to the `background-color` attribute of that box:

```
#box1 { background-color: $colorBox1; }
#box2 { background-color: $colorBox2; }
#box3 { background-color: $colorBox3; }
#box4 { background-color: $colorBox4; }
#box5 { background-color: $colorBox5; }
#box6 { background-color: $colorBox6; }
```

6. Save the file as `colorvariables.scss`. Next, fire up a Node.js command prompt and change the working folder to that of our project folder.

7. In Command Prompt, enter the following code to initiate the compilation process:

    grunt

8. If all is well, we should see a nice set of boxes on screen when previewing the code. It will display boxes using the colors that we've assigned in the `variables` section of our Sass file:
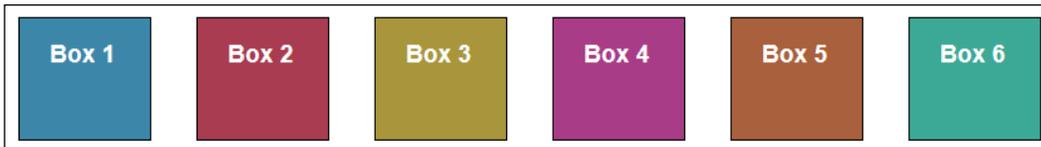
See how easy it is? If we need to make a change, then the least we have to do is change our code in two places and recompile it. For example, if we were to change the shade of color used, then all we need do is change the hex value from step 2 and recompile it.

If, however, we wanted to change the color of the boxes to a completely different color, then we would need to either change or add the value as before, but this time, also alter the value assigned to the affected $ colorBoxX variable from step 3.

> There are pitfalls with how we name our color variable. For insight into some of the pitfalls we may face, take a look at the article by Ben Smithett at `http://bensmithett.com/stop-using-so-many-sass-variables/`.

Working across multiple files

Anyone can spot a potential flaw with our design. It may not be immediately obvious, but what happens when we start to add a lot of variables? We will shoot ourselves in the proverbial foot. We spent all this time creating variables, but completely missed the point of Sass: reusability. Let me explain what I mean.

If we create a standard CSS style sheet for a complex site, it will get very large and unwieldy over time. Instead, we can take advantage of Sass' ability to compile multiple SCSS files in one single CSS style sheet.

Why would we do this? If we group statements into a number of separate files, it makes them shorter and easier to manage, while still maintaining the same end result.

Adapting our color box demo

To see what we mean, let's take a moment to adapt the color box demo that we created in the *Using variables to create colors* section to import the color variables from a separate file into the main style sheet. Granted, it is a little simplistic, but the same principle applies, irrespective of the size of the style sheet. Perform the following steps:

1. We start with firing up a Node.js command prompt. Then, change the working folder to our project area and enter this in the prompt:

   `grunt`

2. This will start Grunt running, which is ready to compile changes to our code.

3. Next, open up a copy of the `colorvariables.scss` file from the previous demo and copy lines 1 to 6 to a new file. Save the new file as `_colors.scss` in the `_partials` folder in our project area.

4. Switch back to `colorvariables.scss`, remove the first six lines, and replace them with the following code at the top of the file:

   ```
   @import "_partials/_colors";
   ```

5. Revert back to the `colorvariables.scss` file and save this file as `cvimport.scss`. If all is well, Grunt will kick in and recompile our code automatically.

6. Check inside the project area. We should not see any compiled CSS files with the color names, but will see the color values used in `cvimport.css`.

The key though is that we should see no visual change to our demo when previewed in a browser, but we can rest in the knowledge that we've restructured our code to make it easier to manage. To really prove that we are indeed importing the files, use a DOM Inspector to view the contents. Here, we can see links to the compiled style sheet and both Sass files from within Firefox's inspector:

```
New        Import...      ⚙        1     /* line 11, ../sass/cvimport.scss */
                                    2 ▾   body {
 ⊙   cvimport.css                   3         font-size: 1.2rem;
     ↳ cvimport.css 8 rules. Save   4         font-weight: bold;
                                    5         font-family: helvetica, sans-serif;
 ⊙   cvimport.scss                  6         color: #fff;
     ↳ cvimport.css 8 rules.        7     }
                                    8
 ⊙   _colors.scss                   9     /* line 18, ../sass/cvimport.scss */
     ↳ cvimport.css 8 rules.       10 ▾   div {
                                   11         height: 5rem;
```

> For a more detailed discussion on how best to use `@import` to structure Sass files, take a look at `http://thesassway.com/beginner/how-to-structure-a-sass-project`. Also, care should be given to how many variables we use. For a useful discussion, take a look at the article by Ben Smithett at `http://bensmithett.com/stop-using-so-many-sass-variables/`.

Now that we've compiled our files, there are a couple of points that we need to be aware of. Take a look at the `@import` statement and note the following points about it:

- The file we've imported doesn't specify an extension. This is perfectly fine in Sass. As long as a file exists with this name, it will import it with or without an extension.

- The underscore in the filename means that it is a **partial**—it's easiest to think of this as a code excerpt that we will put in our main style sheet. The key point though is that there is no need to compile it in a standalone file in its own right. As the `_partials` folder is outside the scope of the monitoring done by our Grunt file, files will only be compiled if imported into our code.

Okay, let's move on to the next step. Now, we will take a look at how to create mixins or blocks of code that you can literally "mix-in" to existing code. We can use these to build up a library of existing code that can be pulled into future projects. However, before we do so, I want to cover one small, but key part of Sass—what if we want to add comments to our code so that we know what it's all about if we revisit it at a later date?

Adding comments

If you've spent any time developing code (which I am sure you have!), then you will no doubt be used to adding comments. We can easily apply the same principles to Sass files.

Comments in Sass usually take the form of either `//` or `/* */`. The former is usually used for one-line statements and the latter for an entire comment block, as shown in this screenshot:

```
1  /*
2  Importing color values
3  from the _colors.scss partial file
4  */
5  @import "_partials/_colors.scss";
6
```

All we need to do is add the appropriate comment with either single line or multiline mode and compile the file. When it comes to compiling our Sass files though, there are a couple of points we should be aware of:

- Single line comments are dropped from the compiled CSS file when working with Sass from the command line; only multiline comments are kept.

- If we use a task runner, such as Grunt, we need to modify `gruntfile.js` to set the style of output; otherwise, comments will be dropped, as shown in the following code:

```
options: {
  sourcemap: 'auto',
  style: 'expanded',
  lineNumbers: true
},
```

When we compile our SCSS file using Grunt, we may also wish to add a comment to indicate the corresponding line in the source code. To do this, we can set `lineNumbers` to `true`.

When we set the style of output in the Grunt file, we can choose to render our compiled code in one of several ways:

Output style value	Result	Output in CSS file, when compiled
expanded	Single line comments using //	This specifies that comments are dropped.
	Multiline comment block using /* and */	This indicates that comments are kept, irrespective of whether the comment is on one line or over several lines.
nested	Single line comments using //	This specifies that comments are dropped.
	Multiline comment block using /* and */	This indicates that comments are kept, irrespective of whether the comment is on one line or over several lines.
compact	Single line comments using //	This denotes that comments are dropped.
	Multiline comment block using /* and */	This indicates that comments are kept, irrespective of whether the comment is on one line or over several lines.
compressed	Single line comments using //	This specifies that comments are dropped, irrespective of whether the comment is single line or multiline when compressed mode is chosen.
	Multiline comment block using /* and */	Note that multiline comments can be kept in compressed mode if the first character of the comment block is an exclamation mark.

Okay. We can now comment on our file (sorry, pun intended!); let's change focus and take a look at how to build mixins using Sass.

Building mixins

At this point, we should be comfortable with making changes to and compiling our code. Let's use some of our newfound knowledge to begin creating mixins that we can reuse in our code.

Mixins do as the name suggests. They allow you to "mix-in" blocks of code into our existing projects, which reduces the amount of code we need to write because the work has already been done.

> This reduction of code is known as the **DRY principle** or **Don't Repeat Yourself**. It's a key tenet of using Sass.

Let's take a look at an example:

```
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}
```

It's a simple block of code. The individual attributes within the block will be recognizable. The two key points of note are that we introduce the block with the `@mixin` keyword, followed by the name of the mixin.

The great thing about mixins is that we can easily reference them within other rules in our code. As an example, we can use it to define the text style for a `.page-title` class; all we need to do is include the highlighted single line of code:

```
.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

If we were to compile our code at this point, we will end up with this:

```
.page-title {
  font-family: Arial;
  font-size: 20px;
  font-weight: bold;
  color: #ff0000;
  padding: 4px;
  margin-top: 10px;
}
```

In this instance, Sass has swapped out the placeholder and replaced it with the contents of the `large-text` mixin. To prove that this is indeed the case, we can add this to an online compiler, such as sassmeister (`http://www.sassmeister.com`), as shown in the following screenshot:
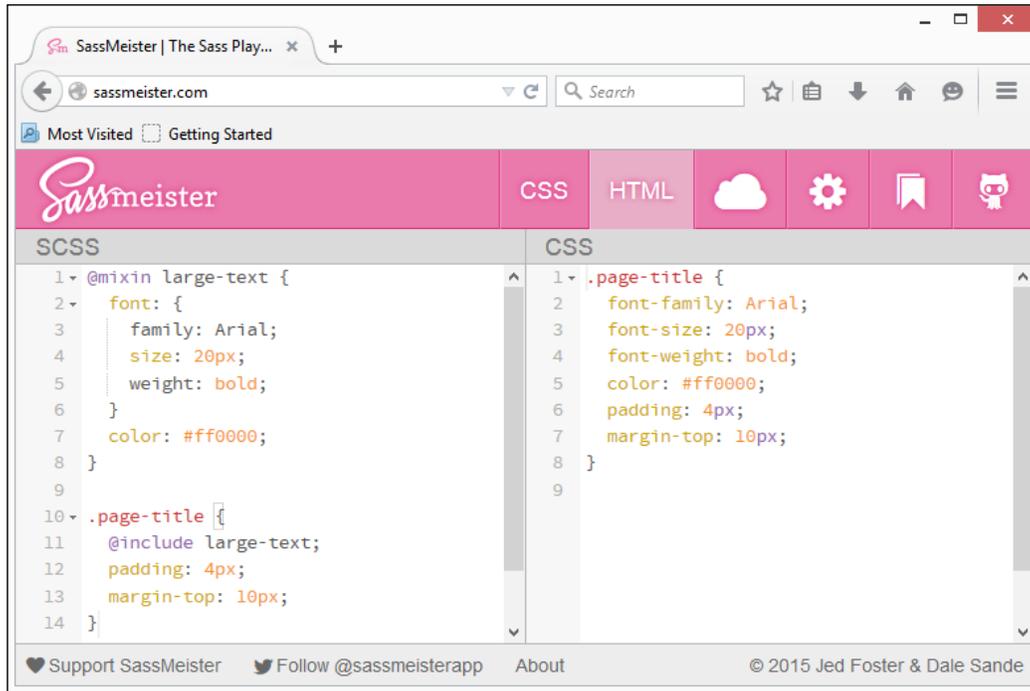
Hold on a moment! I hear you. As the code shown in the screenshot is actually **more**, and not less, why would we want to do this?

Moving mixins to an external file

Well, there is a very good reason for this; the key principle here is to remember that this is about creating a block of code that can be reused in future projects. In our example, we've added the code directly to what would be the main Sass file. In reality, we would hive off the mixins in a separate file and link it to the core Sass file. Let's take a moment to put this into practice by modifying one of our earlier demos. Perform the following steps:

1. Let's begin by firing up a Node.js command prompt. Then, change the working folder to our project area. In this command prompt, enter the following command and click on *Enter*:

   grunt

2. Open up a copy of `sassfunctions.scss` from our earlier demo and then look for this block of code:

```
@mixin firefox-message($selector, $message) {
  #{$selector}:before {
    content: $message;
  }
}
```

3. Copy and paste it into a new file, saving it as `_content.scss` in the `_partials` folder of our project area. Note the addition of the underscore in the filename.

4. In `sassfunctions.scss`, remove the code block shown in step 2 from the top of the file and replace it with this line:

```
@import "_partials/_content.scss";
```

5. Grunt will kick in. If all is well, it will compile the code automatically; we should see no difference visually in the output.

6. To prove that we are indeed importing the file, fire up the DOM Inspector in your browser. If we do this in Firefox, we can see the proof in the **Style Editor** tab:

```
/* line 8, c:/wamp/www/4805OS/Chapter2/sass/sassfunctions.scss */
h1 {
  color: rgba(68, 85, 136, 0.7);
  font-size: 4em; }
  /* line 2, c:/wamp/www/4805OS/Chapter2/sass/_partials/_content.scss */
  h1:before {
    content: "Hi Chrome users! This is a...."; }
```

The two added comment lines (starting with `/*...`) come directly from the source map that we created automatically using Grunt. These are updated every time any changes are made to our code.

> We've only touched on the basics of including mixins. For more details, it's worth taking a look at the official documentation for this subject at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#including_a_mixin`.

At this point, we're free to add additional mixins to our library file. We may only have one mixin now, but over time, we can gradually build up a custom library of mixins that can be used in future projects. If the mixin library gets too big, we can always split them into smaller files and use an `@import` statement to compile the smaller files into one larger library.

Let's change focus and move on. In `sassfunctions.scss`, something different about the mixin that we moved, that is, it had additional parameters as part of the mixin name.

Passing arguments to mixins

We can always remain with creating static mixins, but we will soon hit limitations. One in particular will be that if we had blocks of code that were *similar*, but for some reason could not be adapted to use an existing mixin, then we would have to create multiple variations of what will effectively be the same mixin.

This reduces the level of reusability offered by a mixin. Can we change this? Absolutely! In place of standard mixins with fixed values, we can use **parametric** mixins, where we pass it to different values in each instance of calling the mixin. Let's take a moment to explore what this means for our code.

In a nutshell, parametric mixins are similar to standard mixins, but with one key difference: we can pass values to variables within the mixin. This means that we can effectively use the same style attributes, but get different results depending on the values passed to the mixin. Still not sure what I mean? Let's take a look at an example before adapting one of our earlier demos to use parametric mixins.

Let's assume that we're creating a stylish template to apply our images; it's going to have rounded borders, a shaded background, and will be of a particular size:

```
@mixin image-template($height, $width) {
  border: {
    border-radius: 4px;
    height: $height;
    width: $width;
  }
}
.imgMedium { @include image-template(5rem, 7rem); }
```

We get the following code when we compile the preceding code:

```
.imgMedium {
  border-radius: 4px;
  border-height: 5rem;
  border-width: 7rem;
}
```

See, how easy it is! We will apply a standard border radius of `4px`, but feeding in width and height values of `5rem` and `7rem` respectively, as shown in the following screenshot:
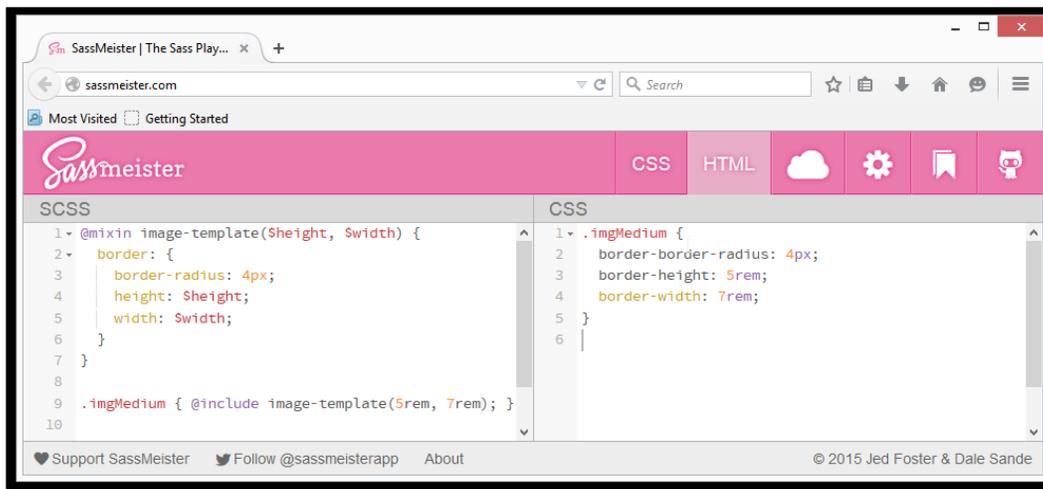
Try changing the values in a browser session at `http://www.sassmeister.com`. You will see the CSS updated automatically with the new values. You can see a similar effect if you change the text being passed to the `$firefox-greeting` variable: it will pass whatever value you drop into this variable. Note that the reason for it showing `border-height` and `border-width` is due to the use of a Sass map; Sass will apply the preceding variable to each attribute during compilation.

> We're not restricted to pass just text arguments; we can pass in other values, such as variables, as we have done in our `sassfunctions. html` example. To learn more about the various options available, take a look at the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#mixin-arguments`.

Exploring the use of @import

Okay. Let's go back to the *Moving mixins to an external file* section for a moment. Now that we've started to use external mixin files, it's worth taking a quick look at a really useful tip that will help us better structure large Sass style sheets and make them easier to manage.

We will begin with setting up a master style sheet, such as `styles.scss`. In it, we will add the following multiple import statements among others that point to individual Sass files or partials:

```
@import "_partials/variables"
@import "_partials/typography"
@import "_partials/elements"
@import "_vendor/grid"
@import "_vendor/mixins"
```

These will contain our individual style rules, which we will add exactly as before.

Although we've used the standard `@import` syntax, it's worth noting that we can use other formats too, such as importing multiple files into one statement, as shown in the following code:

```
@import "_vendor/grid", "_vendor/mixins";
```

This will import both `rounded-corners.scss` and `text-shadow.scss` files.

We can also import multiple files in one go, but separated on a per line basis:

```
@import "_partials/variables",
        "_partials/typography",
        "_vendor/grid";
```

Either method will work. It's a matter of personal preference as to which method you use.

When the main `styles.scss` file is compiled, it will compile the files into one master style sheet. These files can be ones we've created specifically for the project, ones we've reused from earlier projects, or ones from prebuilt mixin libraries available on the Internet (as we will see later in this chapter).

Hold on a moment. Doesn't this mean that we'll end up with a bloated file that contains lots of styles that we don't need? Yes! It will mean that if we are not careful, we could end up with enormous style sheets; the trick is to ensure that we choose the right mixin library to use (to avoid bloating) or extract the mixins we need and discard the rest of the library.

> For more details on how to use the `@import` command in Sass, take a look at the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#import`.

Extending existing styles

As we build more mixins, there is still an inherent risk of bloat through duplication; while Sass helps remove some of it, we can take things further with the use of the `@extend` directive.

Simply put, this can be used to set a core set of styles; if we add additional styles that require minor changes, we can extend them without creating any additional bloat. Let's take a quick look at what we mean.

For a moment, assume that we have a new leather sofa (which we will call sofa **A**). We decide to add a new sofa made of fabric, which becomes sofa **B**.

We *could* simply write duplicate styles for both, but this is old hat and completely misses the capabilities of what Sass can do. Instead, let's hive off the **common** styles into a new style and add two new styles that will deal with the difference by extending the common `.sofa` style:

```
.couch {
  padding: 2em;
  height: 246rem;
  width: 528rem;

}

.sofa-leather {
  @extend .couch;
  background: saddlebrown;
}

.sofa-fabric {
  @extend .couch;
  background: linen;
}
```

If we compile the preceding code, we should get the following:

```
.couch, .couch-leather, .couch-fabric {
  padding: 2em;
  height: 37in;
```

```
    width: 88in;
    z-index: 40;
}

.couch-leather {
  background: saddlebrown;
}

.couch-fabric {
  background: linen;
}
```

This is a simple, but really useful principle to grasp. In some respects, it is better to use `@extend` to create new styles, rather than a mixin. Using a mixin will work perfectly well, but it requires care and attention to ensure that we're only using those attributes that we need, and not any that will lead to unnecessary bloat in our code.

> There is a great article at `http://alistapart.com/article/dry-ing-out-your-sass-mixins` that talks about how to *dry* out your mixins with the use of `@extend`. It is worth a read, although it does assume a certain amount of prior knowledge. The article on how to use the `@extend` keyword by Hugo Giraudel at `http://www.sitepoint.com/sass-mixin-placeholder/` is also worth a read.

Okay, time to move on. We've built a number of different styles of mixins and worked on how we can better structure our code through importing external mixin files. However, do we really need to build lots of different mixin libraries? Absolutely not. There is no sense in doing so if others have already created an alternative and made it available online. Let's take a moment to look at a few examples and see how we can incorporate them in our code.

Using prebuilt mixin libraries

I don't know about you, but I am all for not reinventing the wheel. One way we can achieve this in Sass is to use a prebuilt mixin library.

From something simple as a couple of mixins in a single file to a complex library spanning multiple files, there are dozens available online. The beauty of using a mixin library is that it is really easy to import. In fact, we've already explored how to import mixins with the help of demos throughout this chapter.

We've already seen that it is a simple matter of adding the relevant `@import` statement at the head of our style sheet and then calling the appropriate mixin using the `@include` keyword at the appropriate point in our code.
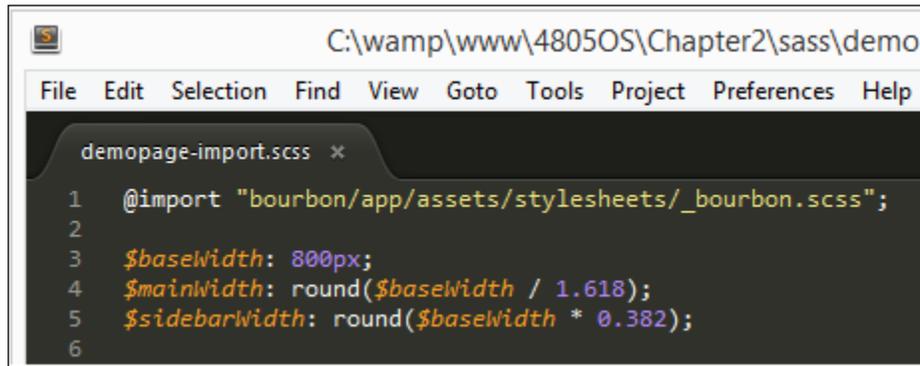
```scss
C:\wamp\www\4805OS\Chapter2\sass\demo
File   Edit   Selection   Find   View   Goto   Tools   Project   Preferences   Help

demopage-import.scss   ×

1    @import "bourbon/app/assets/stylesheets/_bourbon.scss";
2
3    $baseWidth: 800px;
4    $mainWidth: round($baseWidth / 1.618);
5    $sidebarWidth: round($baseWidth * 0.382);
6
```

To show you how easy it is to use the imported mixin library, let's take a look at the following example (which uses the Bourbon library that we've just imported) to render a `@font-face` style rule:

```scss
7    @media screen and (min-device-pixel-ratio: 0) {
8        @include font-face("fira_sansregular", "../fonts/firasans-regular-webfont");
9    }
```

See, how easy it was! We've already used the import directive to pull in mixins from an external file. It's a simple matter of downloading the relevant files from the Internet and referencing these instead.

The hard part though—if indeed it can be described as such—is the choice of library to use with lots of examples out there; here are a few examples to try out:

- **Compass**: http://compass-style.org/
- **Bourbon**: http://bourbon.io/
- **FontAwesome for Sass**: https://github.com/FortAwesome/font-awesome-sass
- **Scut**: http://davidtheclark.github.io/scut/
- **Breakpoint**: http://breakpoint-sass.com/
- **Buttons**: http://unicorn-ui.com/buttons/builder/
- **Saffron**: http://colindresj.github.io/saffron/
- **Andy.scss**: https://github.com/gillesbertaux/andy
- **Sassy Buttons**: http://jaredhardy.com/sassy-buttons/

Using a library will reduce the amount of work we need to do when building style sheets. Simply dropping it in won't be enough though; it does require a little work to incorporate the library. Here are a few tips that will help:

- Don't try to convert everything in one go; it is not worth the trouble. The great Pyramids weren't built in a day. Working with Sass, particularly in large sites, should be an iterative process over time (with changes made in chunks).

- The key to working with libraries is choice, choice, and choice. There are dozens of libraries available; don't feel obliged to stay with one if it doesn't work well for you.

- Sass will compile all the mixins from a library into valid CSS. If the bulk of the styles are not used, then consider either extracting those that are needed into a new custom library or move to using a different library where more styles will be used.

- We're not limited to just using mixin libraries. There are lots of people who have written single or multiple mixins that are not part of any library. It's worth researching online to see what is available. We will use some of these examples, such as those created by Sebastian Ekström, which are available at `http://zerosixthree.se/8-sass-mixins-you-must-have-in-your-toolbox/`.

- Don't forget to also look at the NPM (`http://www.npmjs.com`), Grunt, or Bower websites. Authors will sometimes make them available via these routes in addition to downloading them from their respective sites. Two good examples include the `sass-font-face` mixin at `https://www.npmjs.com/package/sass-font-face` and the `sass-rem` mixin at `https://www.npmjs.com/package/sass-rem`.

Enough chatting. I feel a couple of demos coming! It's time to put something of what we've discussed into practice; to get a feel of what is involved, let's first use the `Animate.scss` library (available at `https://github.com/geoffgraham/animate.scss`) to animate some simple buttons. It's not going to win any awards for sure, but it still shows the process involved in switching to using external libraries as our mixin source.

Animating content using Sass

When building sites, how often have you been asked to animate content in some form or other?

I guess it is fairly often. The mundane part of animating is making sure that we include any attributes that still require vendor prefixes. This isn't an issue when using Sass. We can build a mixin that includes them automatically.

It is worth noting though that the recommended practice is to use an autoprefixer option to provide vendor prefixes; this can be automated using a task runner, such as Node or Grunt.

> There is one built-in within the Grunt file that we have used in previous exercises in this chapter.

For this reason, mixins should concentrate more on the values required for an animation to work and less on vendor prefixes. An ideal use for mixins is to provide support for keyframe-based mixins, which require a few lines.

Let's dig in and take a look at how to animate content using Sass. We will use the Sass version of `Animate.scss` created by Geoff Graham at `https://github.com/geoffgraham/animate.scss`.

> If desired, this can be installed using Bower. Look inside `bower.json` for details of the name to use when installing.

Let's start. Perform the following steps:

1. Extract copies of `animation.html`, `animation.scss`, and the `js` folder from the code download that accompanies this book. Save the markup file to the root of our project folder and the Sass file to the `sass` subfolder.

2. We need a copy of `animate.scss`. So, go ahead and download the library from GitHub at `https://github.com/geoffgraham/animate.scss/archive/master.zip`.

3. Next, open up the library and extract copies of the following files:

Name of mixin file	Found in...
`_properties.scss`	This is found in the root of the archive file
`_fadeOut.scss`	This is found in the `_fading` exits folder
`_flipOutX.scss`	This is found in the `_flippers` folder
`_slideOutUp.scss`	This is found in the `_sliding` exits folder

4. Save all of these in the `_partials` subfolder that sits within the `sass` folder.

5. We need to switch to using the mixins from the library. So, go ahead and remove lines 1 to 59, replacing them with this:

```
@import "_partials/_properties.scss";

@import "_partials/_fadeOut.scss";
@import "_partials/_flipOutX.scss";
@import "_partials/_slideOutUp.scss";

.flipOutX {
    @include flipOutX();
}

.fadeOut {
    @include fadeOut();
}

.slideOutUp {
    @include slideOutUp();
}
```

6. Switch to Koala and recompile the `animation.scss` file. If all is well, we should see something akin to this screenshot, which shows a clicked button in motion:
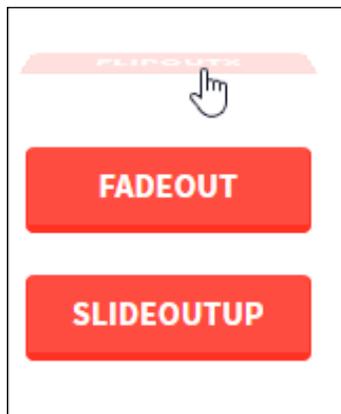
This opens up a world of opportunities. Although we've only referenced three mixins out of the library, we've only had to deal with calling in each of the animations; the mixin library takes care of everything else, such as vendor prefixes, and the individual settings for each animation.

We could easily just use any of the mixins in the library. While it is a perfectly usable library in its own right, it only touches on a small part of what is possible. To give a small flavor of what can be done, try visiting these three sites:

- `http://cubic-bezier.com`: This allows you to generate any number of different Bezier curve animations.

- `http://easings.net`: This site shows off each of the standard animations available in libraries, such as the jQuery UI. It even includes a link to the Compass Ceaser plugin, which allows you to use these same values as Sass variables within any transition animation that we create (we can also use the Sass Easings plugin for Node available at `https://www.npmjs.com/package/sass-easing` to achieve a similar effect).

- `http://cssanimate.com`: This is a relatively new site at the time of writing. It is a perfect work area to create our own animations. The code is automatically generated as standard CSS, but with a little work, it can be converted to Sass.

The key message here is that it is worth spending time when working with something such as animation mixins. First, it is key to get your head around how animation mixins work in Sass and then explore some of the options available in terms of animations that we can use.

Let's move on and take a look at a different library. We will use similar principles in the next demo, but this time, we will use the Compass library, which is frequently used with Sass.

Adapting existing mixins to use libraries

For our second demo, we will take the single page demo used earlier in the chapter and modify the Sass code to use the `font-face` mixin from the Bourbon library.

The mere mention of the word Bourbon may make you think of a fine whiskey, but in this case, I must disappoint—it is the name of one of the more well-known Sass libraries.

At first value, our change would seem a pretty straightforward change. Well, as it so happens, there is a sting in the tail. Don't worry, it's not catastrophic, but it highlights perfectly some of the considerations we need to be mindful of when making these changes.

> For this demo, we will use the Fira Sans font from `http://www.fontsquirrel.com/fonts/fira-sans`; versions suitable for use online have been created with the web font generator at `http://www.fontsquirrel.com/tools/webfont-generator`.

For now, let's get stuck in. We'll start by installing Bourbon; this demo assumes that Bower is already installed. Perform the following steps:

1. Let's start by making sure that the Koala application that we met back in *Chapter 1*, *Introducing Sass*, is installed. If you don't have it running, refer back to it for more details.

2. Next, bring up a Node prompt and change the current folder to our project area. At this prompt, enter the following command and then press *Enter*:

    ```
    bower install bourbon
    ```

3. If all is well, we should see this appear in the command prompt window:

```
Node.js command prompt                              _  □  ×

c:\wamp\www\4805OS\Chapter2>bower install bourbon
bower not-cached      git://github.com/thoughtbot/bourbon.git#*
bower resolve         git://github.com/thoughtbot/bourbon.git#*
bower download        https://github.com/thoughtbot/bourbon/archive/v4.2.2.tar.gz
bower extract         bourbon#* archive.tar.gz
bower resolved        git://github.com/thoughtbot/bourbon.git#4.2.2
bower install         bourbon#4.2.2

bourbon#4.2.2 bower_components\bourbon

c:\wamp\www\4805OS\Chapter2>_
```

> If you want to know the technical reason to use Koala, then it is due to an ongoing bug in `libsass` (used in `grunt-sass`) that throws an error when used with multiple `@each` statements. The latter is used in Bourbon.

4. Next, go ahead and extract `demopage-import.html`, `demopage-import.css`, `demopage-import.scss`, and the fonts folder from the code download that accompanies this book. Then, save them in the project area.

5. Open up a copy of `demopage-import.scss` and alter the code, as shown in the following code:

```
@import "bourbon/app/assets/stylesheets/_bourbon.scss";

$baseWidth: 800px;
$mainWidth: round($baseWidth / 1.618);
$sidebarWidth: round($baseWidth * 0.382);

@include font-face("fira_sansregular", "../fonts/firasans-
  regular-webfont");
```

6. We need to copy the `bourbon` folder from the `bower_components` folder to the sass folder. So, go ahead and do this now before continuing with the next step.

7. Switch back to Command Prompt. Then, enter this command and press *Enter*:

```
grunt
```

8. We should see something similar to this screenshot:


```
File css/styles.css created.
File css/styles.css.map created.
Warning: c:/wamp/www/48050S/Chapter2/sass/bourbon/app/assets/stylesheets/helpers
/font-source-declaration:29: expected 'in' keyword in @each directive
 Use --force to continue.

Aborted due to warnings.

c:\wamp\www\48050S\Chapter2>_
```

9. Hold on. This has thrown an error and is not compiled. To fix this, we need to switch to using Koala. We will explore more of this error at the end of the exercise.

10. Drag the `sass` folder on to the main window. This will add it as a project that Koala can compile. Right-click on `demopage-import.scss` to compile, as shown in this screenshot:
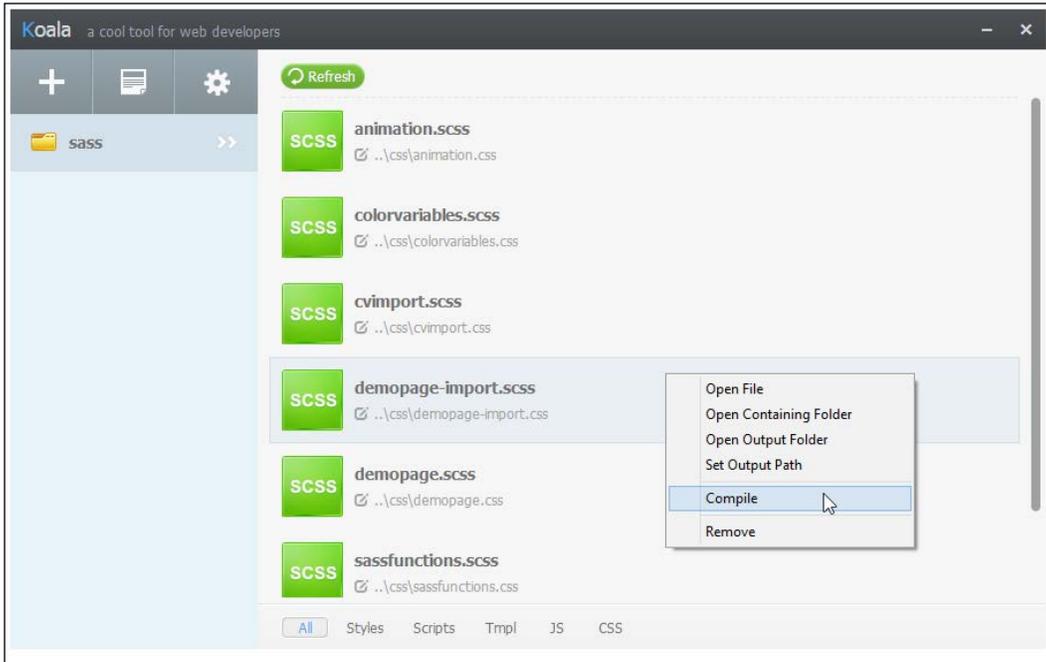
11. If all is well, we should see something akin to this screenshot when previewing the results in a browser:

At this point, we now have a working page that uses the `@font-face` mixin… except…hold on…hasn't the font changed from the previous demo?

Absolutely! With a good reason: the font we used in the original demo is provided using Google Fonts. Licensing issues mean that we can't download the font source files and convert it to use the `@font-face` mixin. This means that we must use an alternative font, such as Fira Sans, which we can download and import using the `@font-face` mixin.

You may also spot that the quality of the font rendered on screen could look better. This is not because I've chosen a bad font; some fonts are just not rendered well on screen in a browser. To get around this, we can use the `min-device-pixel-ratio` attribute to change the number of device pixels shown per CSS pixel. Alter the Sass code as follows:

```
@media screen and (min-device-pixel-ratio: 0) {
  @include font-face("fira_sansregular", "../fonts/firasans-
    regular-webfont");
}
```

Then, recompile it, and all should be well. Hopefully, it will look a lot more like the screenshot shown at the end of the exercise.

Summary

Mixins and variables are two of the most important parts of Sass; without them, our options become somewhat limited. Let's take a moment to consolidate what you have learned throughout this chapter.

We kicked off our journey with a look at how to use variables. Then, we looked at how to assign numeric or alpha values, followed by how to store color values.

We then moved on to look at how Sass allows you to split your style sheet across several files; we can use this to better structure our code and make each constituent file more manageable.

Next, we discussed how to create mixins. We first touched on standard mixins before altering them to become more dynamic and useful by passing values to each mixin. We also covered how to use `@extend` with a pointer and how this can help *dry* out our mixins and remove unnecessary code bloat.

We then rounded out our journey with a look at how to use external mixins from prebuilt libraries, such as Compass or Bourbon, before diving into two examples that used completely different libraries, yet demonstrated that the principals involved are still the same.

Onwards we go! How many times have you had to write style rules that duplicate (in whole or in part) selector names already in the style sheet? It's a real pain—not any more with Sass! Sass has a great facility that allows you to nest styles. I'll explain more in the next chapter.

3
Building Functions, Operations, and Nested Styles

So far, we've covered creating variables and mixins; we've made a start towards simplifying our code. But we can do much more—a typical scenario would be to use operators or functions to construct a whole site theme from just a handful of colors, or defining font sizes for the entire site from a single value. You will learn how to do all these things in this chapter.

We will also touch on nesting styles, to help reduce the amount of code that we need to write; this includes using the & placeholder to reference parent selectors. So, in this chapter, we will be covering the following topics:

- Creating values using functions and operators
- Building site themes using functions
- Reducing repetition of code with nesting
- Constructing layouts using functions and operators
- Using interpolation

Okay, so let's get started!

Creating values using functions and operators

Imagine a scenario where you're creating a masterpiece that has taken days to put together, with a stunning choice of colors that has taken almost as long as the building of the project and yet, the client isn't happy with the color choice. What to do? At this point, I'm sure that while you're all smiles to the customer, you'd be quietly cursing the amount of work they've just landed you with, this late on a Friday. Sound familiar? I'll bet you scrap the colors and go back to poring over lots of color combinations, right? It'll work, but it will surely take a lot more time and effort.

There's a better way to achieve this; instead of creating or choosing lots of different colors, we only need to choose one and create all of the others automatically. How? Easy! When working with Sass, we can use a little bit of simple math to build our color palette.

One of the key tenets of Sass is its ability to work out values dynamically, using nothing more than a little simple math; we could define font sizes from `H1` to `H6` automatically, create new shades of colors, or even work out the right percentages to use when creating responsive sites! We will take a look at each of these examples throughout the chapter, but for now, let's focus on the principles of creating our colors using Sass.

Creating colors using functions

We can use simple math and functions to create just about any type of value, but colors are where these two really come into their own.

The great thing about Sass is that we can work out the hex value for just about any color we want to, from a limited range of colors. This can easily be done using techniques such as adding two values together, or subtracting one value from another.

> To get a feel of how the color operators work, head over to the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#color_operations`—it is worth reading!

Nothing wrong with adding or subtracting values—it's a perfectly valid option, and will result in a valid hex code when compiled. But would you know that both values are actually deep shades of blue? Therein lies the benefit of using functions; instead of using math operators, we can simply say this:

```
p {
  color: darken(#010203, 10%);
}
```

This, I am sure you will agree, is easier to understand as well as being infinitely more readable!

The use of functions opens up a world of opportunities for us. We can use any one of the array of functions such as `lighten()`, `darken()`, `mix()`, or `adjust-hue()` to get a feel of how easy it is to get the values. If we head over to `http://jackiebalzer. com/color`, we can see that the author has exploded a number of Sass (and Compass—we will use this later) functions, so we can see what colors are displayed, along with their numerical values, as soon as we change the initial two values.

Okay, we could play with the site ad infinitum, but I feel a demo coming on—to explore the effects of using the color functions to generate new colors. Let's construct a simple demo. For this exercise, we will dig up a copy of the `colorvariables` demo from *Chapter 2*, *Creating Variables and Mixins*, and modify it so that we're only assigning one color variable, not six.

> For this exercise, I will assume you are using Koala to compile the code, which we touched on back in *Chapter 1*, *Introducing Sass.*

Okay, let's make a start:

1. We'll start with opening up a copy of `colorvariables.scss` in your favorite text editor and removing lines 1 to 15 from the start of the file.

2. Next, add the following lines, so that we should be left with this at the start of the file:
   ```
   $darkRed: #a43;
   $white: #fff;
   $black: #000;

   $colorBox1: $darkRed;
   $colorBox2: lighten($darkRed, 30%);
   $colorBox3: adjust-hue($darkRed, 35%);
   $colorBox4: complement($darkRed);
   $colorBox5: saturate($darkRed, 30%);
   $colorBox6: adjust-color($darkRed, $green: 25);
   ```

3. Save the file as `colorfunctions.scss`. We need a copy of the markup file to go with this code, so go ahead and extract a copy of `colorvariables.html` from the code download, saving it as `colorfunctions.html` in the root of our project area. Don't forget to change the link for the CSS file within to `colorfunctions.css`!

4. Fire up Koala, then drag and drop `colorfunctions.scss` from our project area over the main part of the application window to add it to the list:



5. Right-click on the file name and select **Compile**, and then wait for it to show **Success** in a green information box.

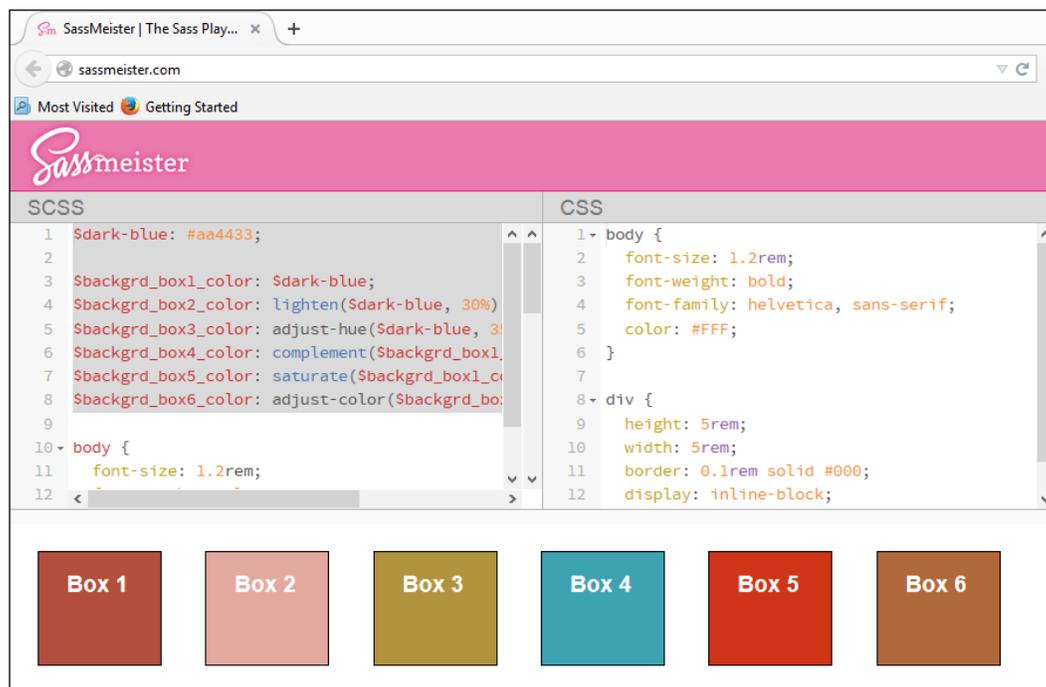6. If we preview the results of our work in a browser, we should see the following boxes appear:
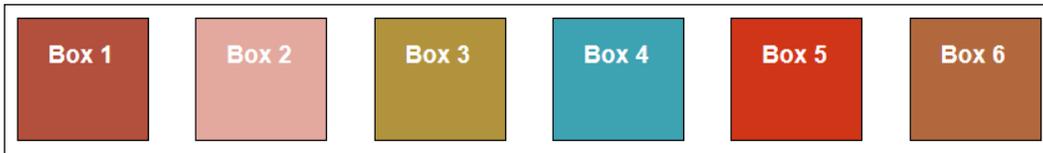


At this point, we have a working set of colors—granted, we might have to work a little on making sure that they all work together. But the key point here is that we have only specified **one** color, and that the others are all calculated automatically through Sass.

Now that we are only defining one color by default, how easy is it to change the colors in our code? Well, it is a cinch to do so. Let's try it out using the help of the SassMeister playground.

# Changing the colors in use

We can easily change the values used in the code, and continue to refresh the browser after each change. However, this isn't a quick way to figure out which colors work; to get a quicker response, there is an easier way: use the online Sass playground at `http://www.sassmeister.com`.

This is the perfect way to try out different colors—the site automatically recompiles the code and updates the result as soon as we make a change. Try copying the HTML and SCSS code into the play area to view the result. The following screenshot shows the same code used in our demo, ready for us to try using different calculations:
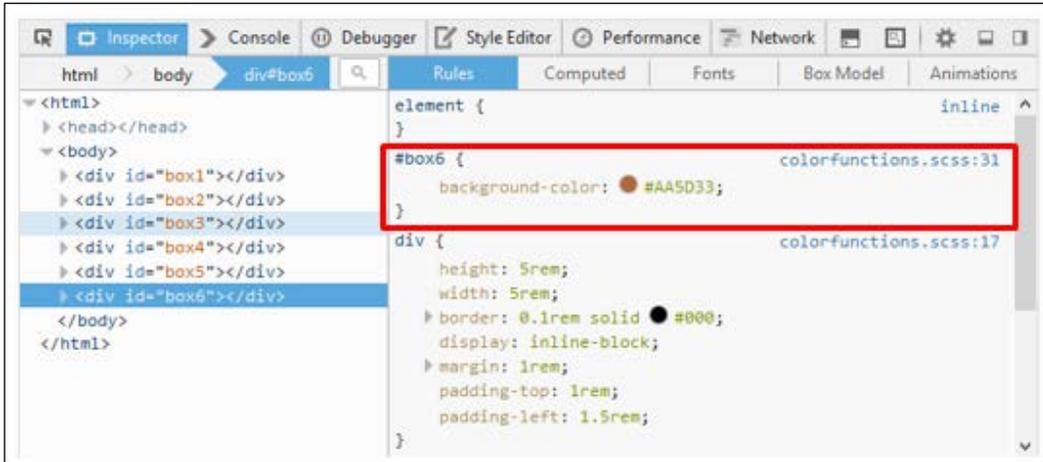
All images work on the principle that we take a base color (in this case, `$dark-blue`, or `#a43),` then adjust the color either by a percentage or a numeric value. When compiled, Sass calculates what the new value should be and uses this in the CSS. Take, for example, the color used for `#box6`, which is a dark orange with a brown tone, as shown in this screenshot:



To get a feel of some of the functions that we can use to create new colors (or shades of existing colors), take a look at the main documentation at `http://sass-lang.com/documentation/Sass/Script/Functions.html`, or `https://www.makerscabin.com/web/sass/learn/colors`. These sites list a variety of different functions that we can use to create our masterpiece.

> We can also extend the functions that we have in Sass with the help of custom functions, such as the toolbox available at `https://github.com/at-import/color-schemer` — this may be worth a look.

In our demo, we used a dark red color as our base. If we're ever stuck for ideas on colors, or want to get the right HEX, RGB(A), or even HSL(A) codes, then there are dozens of sites online that will give us these values. Here are a couple of them that you can try:

- HSLa Explorer, by Chris Coyier — this is available at `https://css-tricks.com/examples/HSLaExplorer/`.
- HSL Color Picker by Brandon Mathis — this is available at `http://hslpicker.com/`.

- If we know the name, but want to get a Sass value, then we can always try the list of 1,500+ colors at `https://github.com/FearMediocrity/sass-color-palettes/blob/master/colors.scss`. What's more, the list can easily be imported into our CSS, although it would make better sense to simply copy the chosen values into our Sass file, and compile from there instead.

# Mixing colors

The one thing that we've not discussed, but is equally useful is that we are not limited to using functions on their own; we can mix and match any number of functions to produce our colors.

A great way to choose colors, and get the appropriate blend of functions to use, is at `http://sassme.arc90.com/`. Using the available sliders, we can choose our color, and get the appropriate functions to use in our Sass code. The following image shows how:
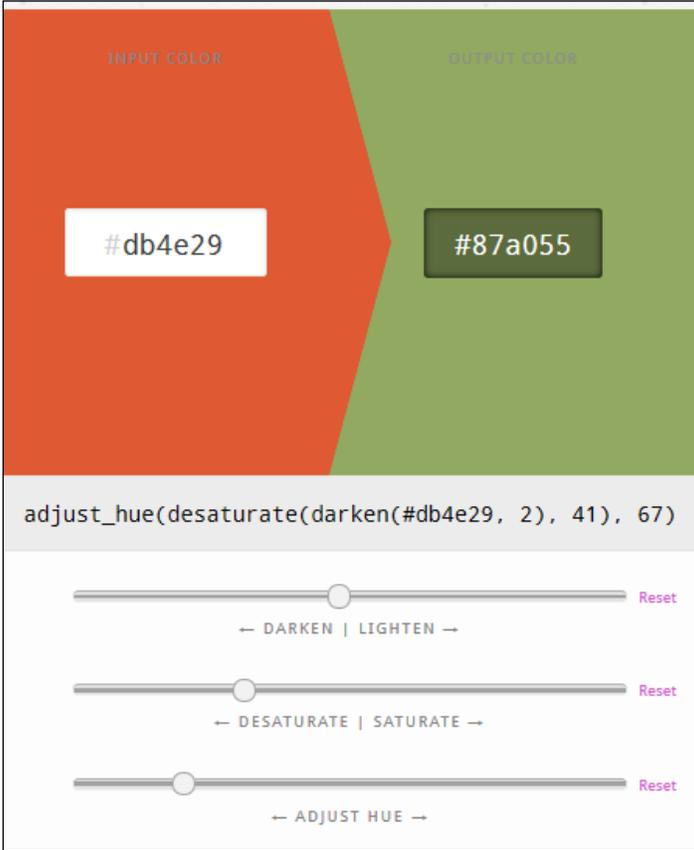
In most cases, we will likely only need to use two functions (a mix of darken and adjust hue, for example); if we are using more than two–three functions, then we should perhaps rethink our approach! In this case, a better alternative is to use Sass's `mix()` function, as follows:

```
$white: #fff;
$berry: hsl(267, 100%, 35%);
p { mix($white, $berry, 0.7) }
```

…which will give the following valid CSS:

```
p {
  color: #5101b3;
}
```

This is a useful alternative to use in place of the command we've just touched on; after all, would you understand what `adjust_hue(desaturate(darken(#db4e29, 2), 41), 67)` would give as a color? Granted, it is something of an extreme calculation, nonetheless, it is technically valid. If we use `mix()` instead, it matches more closely to what we might do, for example, when mixing paint. After all, how else would we lighten its color, if not by adding a light-colored paint?

Okay, let's move on. What's next? I hear you ask. Well, so far we've used core Sass for all our functions, but it's time to go a little further afield. Let's take a look at how you can use external libraries to add extra functionality. In our next demo, we're going to introduce using Compass, which you will often see being used with Sass.

Using an external library

So far, we've looked at using core Sass functions to produce our colors—nothing wrong with this; the question is, can we take things a step further?

Absolutely, once we've gained some experience with using these functions, we can introduce custom functions (or helpers) that expand what we can do. A great library for this purpose is Compass, available at `http://www.compass-style.org`; we'll make use of this to change the colors which we created from our earlier boxes demo, in the section, *Creating colors using functions*.
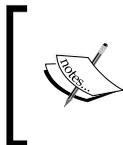
> Compass is a CSS authoring framework, which provides extra mixins and reusable patterns to add extra functionality to Sass. In our demo, we're using `shade()`, which is one of the several color helpers provided by the Compass library.

Let's make a start:

1. We're using Compass in this demo, so we'll begin with installing the library. To do this, fire up Command Prompt, then navigate to our project area.

2. We need to make sure that our installation RubyGems system software is up to date, so at Command Prompt, enter the following, and then press *Enter*:

   ```
   gem update --system
   ```

3. Next, we're installing Compass itself—at the prompt, enter this command, and then press *Enter*:

   ```
   gem install compass
   ```

4. Compass works best when we get it to create a project shell (or template) for us. To do this, first browse to `http://www.compass-style.org/install`, and then enter the following in the **Tell us about your project...** area:

```
I would like to set up my  new      ∨  compass ∨  project named  colorlibrary       with
compass's ∨  starter stylesheets.
I prefer the  CSS based (SCSS) ∨  syntax and would like to  customize my project's      ∨
directory structure using  sass        for the sass source directory,  css        for
the stylesheets output directory,  javascripts    for the javascripts directory, and
images        for the images directory.
```

> Leave anything in grey text as blank.
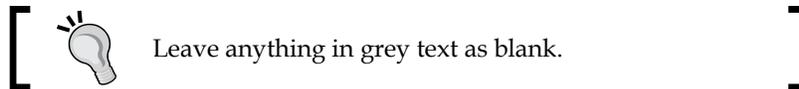
5. This produces the following commands—enter each at Command Prompt, pressing *Enter* each time:

```
$ gem install compass
$ compass create colorlibrary --sass-dir "sass" --css-dir "css"
```

6. Navigate back to Command Prompt. We need to compile our SCSS code, so go ahead and enter this command at the prompt (or copy and paste it), then press *Enter*:

   ```
   compass watch –sourcemap
   ```

7. Next, extract a copy of the `colorlibrary` folder from the code download that accompanies this book, and save it to the project area.

8. In `colorlibrary.scss`, comment out the existing line for `$backgrd_box6_color`, and add the following immediately below it:

    ```
    $backgrd_box6_color: shade($backgrd_box5_color, 25%);
    ```

9. Save the changes to `colorlibrary.scss`. If all is well, Compass's watch facility should kick in and recompile the code automatically. To verify that this has been done, look in the `css` subfolder of the `colorlibrary` folder, and you should see both the compiled CSS and the source map files present.

> If you find Compass compiles files in unexpected folders, then try using the following command to specify the source and destination folders when compiling:
>
> ```
> compass watch --sass-dir sass --css-dir css
> ```

10. If all is well, we will see the boxes, when previewing the results in a browser window, as in the following image. Notice how Box 6 has gone a nice shade of deep red (if not almost brown)?

| Box 1 | Box 2 | Box 3 | Box 4 | Box 5 | Box 6 |
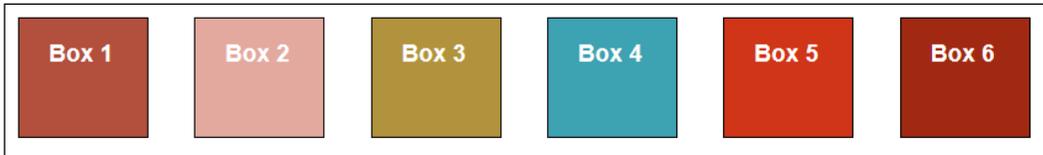
11. To really confirm that all the changes have taken place as required, we can fire up a DOM inspector such as Firebug; a quick check confirms that the color has indeed changed:

```
#box6 {                                    colorlibrary.scss:32
    background-color: ● #98210D;
}
```

12. If we explore even further, we can see that the compiled code shows that the original line for Box 6 has been commented out, and that we're using the new function from the Compass helper library:

```
$dark-blue: #aa4433;

$backgrd_box1_color: $dark-blue;
$backgrd_box2_color: lighten($dark-blue, 30%);
$backgrd_box3_color: adjust-hue($dark-blue, 35%);
$backgrd_box4_color: complement($backgrd_box1_color);
$backgrd_box5_color: saturate($backgrd_box1_color, 30%);
//$backgrd_box6_color: adjust-color($backgrd_box1_color, $green: 25);
$backgrd_box6_color: shade($backgrd_box5_color, 25%);
```

This is a great way to push the boundaries of what we can do when creating colors. To learn more about using the Compass helper functions, it's worth exploring the official documentation at `http://compass-style.org/reference/compass/helpers/colors/`. We used the `shade()` function in our code, which darkens the color used. There is a key difference to using something such as `darken()` to perform the same change. To get a feel of the difference, take a look at the article on the CreativeBloq website at `http://www.creativebloq.com/css3/colour-theming-sass-and-compass-6135593`, which explains the difference very well.

The documentation is a little lacking in terms of how to use the color helpers; the key is not to treat them as if they were normal mixins or functions, but to simply reference them in our code. To explore more on how to use these functions, take a look at the article by Antti Hiljá at `http://clubmate.fi/how-to-use-the-compass-helper-functions/`.

> We can, of course, create mixins to create palettes—for a more complex example, take a look at `http://www.zingdesign.com/how-to-generate-a-colour-palette-with-compass/` to understand how such a mixin can be created using Compass.

Okay, let's move on. So far, we've talked about using functions to manipulate colors; the flip side is that we are likely to use operators to manipulate values such as font sizes.

We will explore the creation of color palettes in more detail, using some of the functions we've discussed, later in this chapter. For now, let's change tack and take a look at creating new values for changing font sizes.

Changing font sizes using operators

Earlier in the book, we talked about using functions to create practically any value.

Well, we've seen how to do it with colors; we can apply similar principles to creating font sizes too. In this case, we set a base font size (in the same way that we set a base color), and then simply increase or decrease font sizes as desired.

In this instance, we won't use functions, but instead, use standard math operators, such as add, subtract, or divide. When working with these operators, there are a couple of points to remember:

- Sass math functions preserve units—this means we can't work on numbers with different units, such as adding a `px` value to a `rem` value, but can work with numbers that can be converted to the same format, such as inches to centimeters

- If we multiply two values with the same units, then this will produce square units (that is, `10px * 10px == 100px * px`). At the same time, `px * px` will throw an error as it is an invalid unit in CSS.

- There are some quirks when working with `/` as a division operator —in most instances, it is normally used to separate two values, such as defining a pair of font size values. However, if the value is surrounded in parentheses, used as a part of another arithmetic expression, or is stored in a variable, then this will be treated as a division operator.

> For full details, it is worth reading the relevant section in the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#division-and-slash`.

With these in mind, let's create a simple demo—a perfect use for Sass is to automatically work out sizes from `H1` through to `H6`. We could just do this in a simple text editor, but this time, let's break with tradition and build our demo directly into a session on `http://www.sassmeister.com`. We can then play around with the values set, and see the effects of the changes immediately. If we're happy with the results of our work, we can copy the final version into a text editor and save them as standard SCSS (or CSS) files.

1. Let's begin by browsing to `http://www.sassmeister.com`, and adding the following HTML markup window:

```html
<html>
  <head>
    <meta charset="utf-8" />
    <title>Demo: Assigning colors using variables</title>
```

```
    <link rel="stylesheet" type="text/css" href="css/
      colorvariables.css">
  </head>
  <body>
    <h1>The cat sat on the mat</h1>
    <h2>The cat sat on the mat</h2>
    <h3>The cat sat on the mat</h3>
    <h4>The cat sat on the mat</h4>
    <h5>The cat sat on the mat</h5>
    <h6>The cat sat on the mat</h6>
  </body>
</html>
```

2. Next, add the following to the SCSS window — we first set a base value of 3.0, followed by a starting color of #b26d61, or a dark, moderate red:

```
$baseSize: 3.0;
$baseColor: #b26d61;
```

> We need to add our H1 to H6 styles. The rem mixin was created by Chris Coyier, at https://css-tricks.com/snippets/css/less-mixin-for-rem-font-sizing/.

3. We first set the font size, followed by setting the font color, using either the base color set earlier, or a function to produce a different shade:

```
h1 {
  font-size: $baseSize;
  color: $baseColor;
}

h2 {
  font-size: ($baseSize - 0.2);
  color: darken($baseColor, 20%);
}

h3 {
  font-size: ($baseSize - 0.4);
  color: lighten($baseColor, 10%);
}

h4 {
  font-size: ($baseSize - 0.6);
  color: saturate($baseColor, 20%);
```
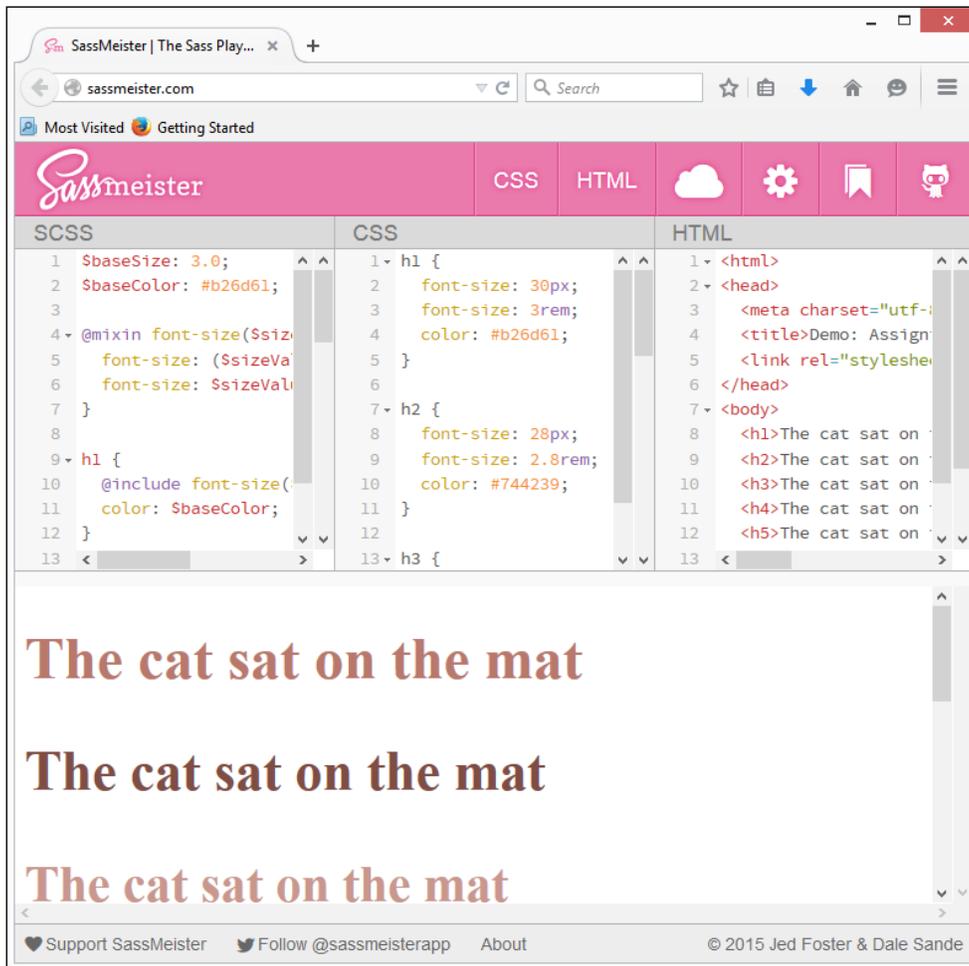
```
}

h5 {
    font-size: ($baseSize - 0.8);
    color: $baseColor - 111;
}

h6 {
    font-size: ($baseSize - 1.0);
    color: rgb(red($baseColor) + 10, 23, 145);
}
```

4. SassMeister will automatically compile the code to produce a valid CSS, as shown in this screenshot:

Try changing the base size of `3.0` to a different value—using `http://www.sassmeister.com`, we can instantly see how this affects the overall size of each H value. Note how we're multiplying the base variable by `10` to set the pixel value, or simply using the value passed to render each heading.

In each instance, we can concatenate the appropriate unit using a plus (+) symbol. We then subtract an increasing value from `$baseSize`, before using this value as the font size for the relevant H value.

> You can see a similar example of this by Andy Baudoin as a CodePen, at `http://codepen.io/baudoin/pen/HdliD/`. He makes good use of nesting to display the color and strength of shade; we will cover nesting later in this chapter. Note that it uses a little JavaScript to add the text of the color that each line represents, and can be ignored; it does not affect the Sass used in the demo.

The great thing about using a site such SassMeister is that we can play around with values and immediately see the results.

> For more details on using number operations in Sass, browse to the official documentation, which is at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#number_operations`.

Okay, onwards we go. Let's turn our attention to creating something a little more substantial; we're going to create a complete site theme using the power of Sass and a few simple calculations.

Creating site themes using functions

So far, we've worked with individual colors, or a group of them—but what about ensuring that they work well together?

It's time to change that, and to look at generating color palettes where care has to be taken over which colors we want to use. Thankfully, we don't actually have to do too much work; with the right math, Sass will take care of creating colors that work well together, automatically.

The key to creating color palettes that work lies in the various types available—each is based on a principle, such as maintaining a constant hue, whilst varying the saturation or value levels (monochromatic). There are a number of palette types available, of which any one may suit your projects. Let's take a moment to consider some examples.

- **Monochromatic** color harmony maintains a constant value in the hue and varies the saturation and value levels
- **Complementary** color harmony uses hue values that are about 200 degrees apart (with variations in saturation and value levels, as desired)
- **Split Complementary** is a variation on complementary colors—it uses the two colors adjacently, in addition to the complementary color
- **Triadic** color harmony uses three hue values, roughly 130 degrees apart (with varying saturation and value levels to taste)
- **Analogous** color harmony uses three or more hue values in a range of about 30 degrees, each side of our chosen color

> For a visual representation, browse the Internet—there is plenty of information available. You can go to `http://www.tigercolor.com/color-lab/color-theory/color-harmonies.htm` for a visual example of some of the palettes available.

Alright, let's get down and dirty with some code! We will build to create a full-sized theme for a mocked-up web page, but before doing so, let's start with something simple such as creating a palette swatch using Sass.

Creating palettes using functions

Colors, colors—where does one start with choosing the right color, I wonder?

Well, there are plenty of sources available; it's worth having a look online at sites such as COLOURlovers, at `http://www.colourlovers.com`. Another great resource is the extensive list of color values by Richard Bray, at `http://richbray.me/cms/`—it even comes with suitable Sass color names to boot!

Keeping this aside, let's assume that we've chosen a suitable base color—for our next example, we will be using a nice strong shade of red, or `#c24`. Let's get started:

1. We'll begin by extracting a copy of the HTML markup required for this demo, in the form of `colorpalette.html`; save a copy of this file to the project area.

2. Next, go ahead and add the following Sass styles to a new file, saving it as `colorpalette.scss`. We begin with some standard CSS styling to handle the fonts used in the demo:

```
body { font-family: sans-serif, Times New Roman; font-size:
  1.6rem; }
```

3. The magic of our demo comes in the form of the style rule for the `.container` div. We start by setting the size and positioning on the screen:

```
.container div {
  width: 5em;
  height: 5em;
  display: inline-block;
  padding: 0.5rem;
```

4. Immediately below come the variables that control the number of colors to render on screen, the starting base color, range, and the offset of colors to use are:

```
$num-colors: 7;
$base-color: #c24;
$spectrum: 180deg;

$offset: 30deg;
```

5. We round off our SCSS code with a brief introduction to using the `for…each` directive; this simply counts through a specified count, adjusting the hue of each color in turn using the formula given in the code:

```
@for $i from 0 to $num-colors {
  &:nth-child(#{$i}) {
    background: adjust-hue($base-color, $offset +
      $spectrum / $num-colors * $i);
  }
}
}
```

6. If all is well, we should see this screenshot of a color palette when previewing the results in a browser:

The beauty of an example such as this one is that it is perfectly suited for playing with in a session at `http://www.sassmeister.com`. All we need do is change one base color, and let Sass recompile the code to automatically produce new colors for us. Neat, huh?

> In complete contrast, take a look at JSFiddle at `http://jsfiddle.net/i_heart_php/6yaLLjd5/`. It creates a perfectly valid palette, but does it the hard way. If you can get your head around using the function, then it will make working with Sass color palettes much easier!

So far, we've created palettes from scratch, using the power of the Sass functions to do the hard work for us. This assumes though that we already know our base color, but have yet to work out what we want to use as our remaining palette colors. In some instances, we may already use a tool such as Adobe's Kuler (or Color CC, as it is now called). What if we're using such a tool, online sites such as `http://www.paletton.com` or an example theme, such as the one at `http://paletton.com/#uid=13g0u0k7A915q-reIOtmCy8O2n`?

This puts a different spin on how we arrive at our Sass code. Let's take some time to explore what this means for our workflow process.

Using tools to create palettes

When creating palettes, it is easy to assume that we might be starting from scratch, with no preconceived ideas or plans in place. However, this isn't always going to be the case. What if we already use a tool such as the Adobe Color CC to create our palettes for us?

No problem, we can still make use of these palettes; it requires a little more work, and will no doubt need fine-tuning. This might initially put some people off, but Sass is all about preparation. If you decide to change your palette's base color, then Sass will automatically calculate the new values for you, which reduces the time and effort required to update your code. The time spent upfront in setting up the values and fine-tuning them now will be repaid at a later date!

When working with the Adobe Color CC tool, you may find that some colors appear slightly different, as a margin of error is introduced—the format allows us to be very precise (even to a decimal place). Color CC and packages such as Photoshop will round off the values to the nearest integer, so when these values are converted to their HEX equivalents, they will return slightly different results. We can get around this by using a playground such as SassMeister, which compiles code on the fly for us; we can tweak the values if required.

In our next demo, we're going to convert two palettes from the Adobe Color CC tool—one will use manual values, and the second will work out all the colors from one given base color.

Let's make a start by setting up our palette:

1. We'll start by choosing our palette colors from the Adobe Color CC site; for the purposes of this demo, I've already set up an example palette, which you can see at `http://adobe.ly/1ejWj3e`.

2. The URL produced by the Adobe tool is ugly to say the least! Let's convert this horrible URL to a set of Sass variables that we can use. For this purpose, head over to a Pen by Toto at `http://codepen.io/teetteet/pen/uacdJ`, and enter the URL to be converted, then press *Enter*:
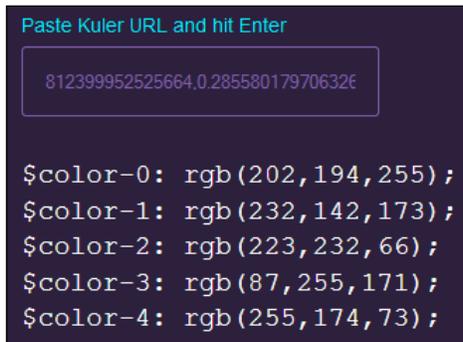
We can, of course, stop there and just use these variables as they are. Instead, we're going to convert them into something that many designers will likely find more intuitive: HSB, or Hue, Saturation, and Brightness values. To do this, we will use a small function to convert them from RGB to HSB values:

1. In a SassMeister session, add the following code to the SCSS panel. We start by setting the `<body>` sizes, followed by the basic sizing for each `<div>`:

```scss
body {
  font-family: sans-serif, Times New Roman;
  font-size: 1.6rem;
}

.container div {
  width: 5em;
  height: 5em;
  display: inline-block;
  padding: 0.5rem;
  border: 0.1rem solid #000;
}
```

2. This is where the magic happens—this function converts a given color to the HSB format (this is a cousin of HSL, or Hue, Saturation, and Lightness and works in a similar manner):

```
@function hsb($h-hsb, $s-hsb, $b-hsb, $a: 1) {
  @if $b-hsb == 0 {
    @return hsla(0, 0, 0, $a)
  } @else {
    $l-hsl: ($b-hsb/2) * (2 - ($s-hsb/100));
    $s-hsl: ($b-hsb * $s-hsb) / if($l-hsl < 50, $l-hsl * 2,
      200 - $l-hsl * 2);
    @return hsla($h-hsb, $s-hsl, $l-hsl, $a);
  }
}
```

3. We can't, obviously make use of the function unless we call it in our code; go ahead and add this at the end of the SCSS window:

```
#box1 { backgroundColor: hsb(248, 24, 100); }
#box2 { backgroundColor: hsb(339, 39, 91); }
#box3 { backgroundColor: hsb(63, 71, 91); }
#box4 { backgroundColor: hsb(150, 66, 100); }
#box5 { backgroundColor: hsb(33, 71, 100); }
```

4. We now need to add our markup. In the HTML window, add the following code:

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Demo: Creating color palettes using Sass</title>
  </head>
  <body>
    <div class="container">
      <div id="box1">Box 1</div>
      <div id="box2">Box 2</div>
      <div id="box3">Box 3</div>
      <div id="box4">Box 4</div>
      <div id="box5">Box 5</div>
    </div>
  </body>
</html>
```

5. If all is well, we should see this, when we preview the results in the
    SassMeister output window:

| Box 1 | Box 2 | Box 3 | Box 4 | Box 5 |
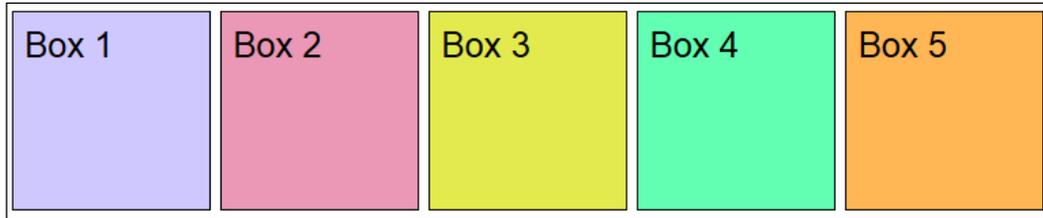
A small point of note—Sass will always compile the color value to the equivalent
HEX codes to ensure compatibility across the widest range of browsers possible.

One might ask: why we would, therefore, use HSB? The simple answer is that it is
easier to understand what effect changing one of the constituent values would have
on the result. For example, changing the third value (or Brightness), will clearly
change how bright the color is; changing the middle value (Saturation) will make the
color more or less saturated. It's one of the several ways to work with color in Sass,
rather than relying on RGB all the time.

Trying out an alternative palette

Now that we've seen the basics of creating a Sass palette from the Adobe Color CC
tool, let's work on something a little more complex. We don't want to specify the
color variables manually each time, do we?

Absolutely not! Instead, we can apply some basic principles to work out the
remaining four colors, based on a given base color. This time though, I'm not simply
going to give you the steps. Let's see if you can work through the principles needed
to produce the answer—the steps needed are not new; we've covered them at
various points throughout the chapter.

Let's make a start—I will assume continued use of the SassMeister session, from the previous exercise:

1. For this exercise, we will use the palette URL at `http://adobe.ly/1HBKN09`, which produces the following range of colors:
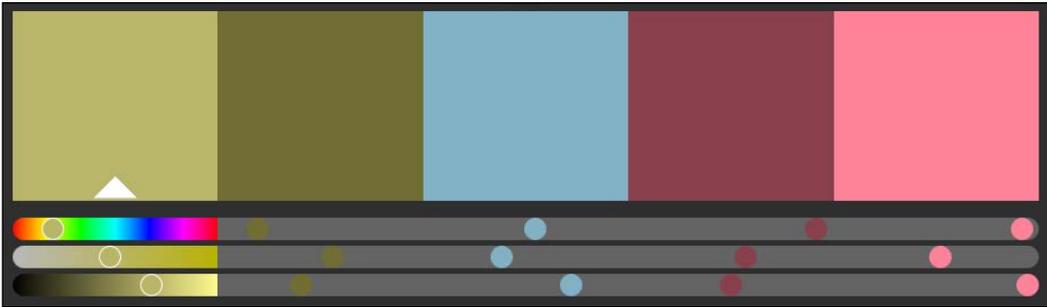
2. Remove the `hsb()` function from the previous demo, and replace it with the following code. We start by setting some variables: the base color in use is the HSL equivalent of the light brown color highlighted to the left of the palette:

```
$mainColor: hsl(57, 35%, 53%);
$mainColorHue: hue($main-color);
$mainColorSaturation: saturation($main-color);
$mainColorLightness: lightness($main-color);
```

3. Next, replace the block from step 5 of the previous exercise with the following lines:

```
@function theme-adjust($h: 0, $s: 0, $l: 0) {
  @return hsl($mainColoHue + $h, $mainColorSaturation + $s,
    $mainColorLightness + $l);
}

#box1 { background-color: $mainColor; }
#box2 { background-color: theme-adjust(0, 6, -25); }
#box3 { background-color: theme-adjust(141, 1%, 8); }
#box4 { background-color: theme-adjust(293, 5%, -18); }
#box5 { background-color: theme-adjust(293, 65%, 21); }
#box5 { background-color: hsl($main-color-hue + 293, $main-
  color-saturation + 65%, $main-color-lightness + 21) }
```

4. We need to tweak the basic styling of each box slightly; make the changes as highlighted:

```
body { font-family: sans-serif, Times New Roman; font-size:
  1.6rem; font-weight: bold; }
.container div { width: 5em; height: 5em; display: inline-
  block; padding: 0.5rem; border: 0.1rem solid #000; color:
  #fff; }
```

5. If all is well, we should see this in SassMeister output window, when previewing the results:
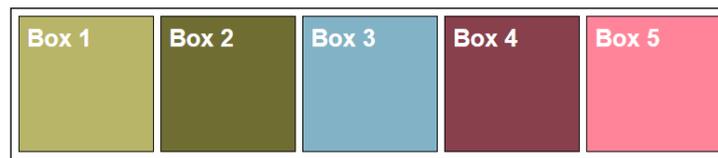
This demo has produced a reasonably accurate copy of the colors that we created in the Color CC palette, but you *may* notice that not every color is exactly the same when using this process.

The key to using this method is to treat it as a good start point; it produces results, but is a little cumbersome as we have to work out the HSL values in code. For help with this method, it is worth using a site such as `http://hslpicker.com` to get the correct HSL values for the chosen colors. It returned the following HSL values for the colors used in our demo:

```
#B2AE5E = hsl(57, 35%, 53%)
#66632B = hsl(57, 41%, 28%)
#78AABF = hsl(198, 36%, 61%)
#7F3642 = hsl(350, 40%, 35%)
#FF7990 = hsl(350, 100%, 74%)
```

At this point, you may ask the question: how is this useful to me? It's a good question; the two demos highlight a few key points, so let's cover those now:

- If you want to use Sass, you will get better engagement when it becomes a core part of your process, and is not seen as yet another tool bolted onto your workflow. If we're already using an existing color palette tool or process, why not use Sass to automatically calculate the values for you? If you need to change the base value in the future, then all the other colors will be automatically updated and still work together.

- If Color CC is your tool of choice, then this method shows a perfect way to not only export your palette, but actually retrieve and use the colors directly into Sass.

Let's move on now. We've created our new colors using the standard functions in Sass. This is a perfectly acceptable route to take, but limits the extent of what we can achieve. We can broaden our horizons; in much the same way as we used Compass to add some additional color functions, we can use external libraries to help create complete palettes for us.

A perfect example of this comes in the form of SassyColors. Its creator, Chris Asher, has created a demo which shows off how you can specify a base color and the name of a palette type to generate a number of different colors from that one base color:

Split Complementary: sassycolours(#7EA7D8, 'split')

It's worth spending a little time exploring this library; it takes what we've already done to the next level, while at the same time avoiding the need to spend lots of time creating code.

> The main demo can be found at `http://chrisasher.com/sassycolours/`. Chris has produced a useful tutorial on using the library, which is at `http://chrisasher.com/sassycolours-a-colour-palette-generator-for-sass/`.

Okay, let's move on. Time for some light relief, methinks; we're going to touch on the use of color filters using Sass, as another way to change colors within our projects. Before we do so, we need to touch on a different concept—that of using color maps within Sass.

Using maps to reference colors

Once we've spent time working with color variables, we'll begin to build up what could become a long list of color variables, which are likely to have long names, and don't always match what they were intended to be used for! There is a better way, and something to work towards, once you have become more familiar with using the color variables, that is, using **color maps**. For now, we won't go into too much detail, but let's take a whistle-stop tour of using one, so you get an idea about what they are and how they work.

Color maps take the form of key/value pairs, which take this format:

```
$map: (key1: value1, key2: value2, key3: value3);
```

In a real-world context, we might see something like this:

```
$brandCOlors: (  primary: #333, secondary: #666, accent: #999);
```

Why would we use a map? I hear you ask. Easy — when creating variables, we naturally have to give a name to each. In some instances, this might require a long, verbose name which is not an ideal thing to do.

Instead, we can set up a map. This may require a few more lines of code, but that is a small price to pay when it is easier for us to understand which colors are being used. Let's work through a quick example using SassMeister. Here, we have defined various shades of purple and grey, using different values or functions:

```
$colorBase-Grey: rgb(229,231,234);

$palettes: (
  purple: (
    base:   rgb(42,40,80),
    light:  rgb(51,46,140),
    dark:   rgb(40,38,65)
  ),
  grey: (
    base:  $colorBase-Grey,
    light: lighten($colorBase-Grey, 10%),
    dark: darken($colorBase-Grey, 10%)
  )
);
```

To access the colors, we can use a simple function where we define both the desired color and the shade or tone:

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

Let's put that function into an example, using a link as our basis:

```
a {
  color: palette(purple);
  &:hover {
    color: palette(purple, light);
  }
}
```

A quick check in a SassMeister session produces the following CSS, where `#2a2850` gives a very dark desaturated blue (almost purple), and `#332e8c` produces a shade of dark blue:

```
CSS
1 ▾ a {
2     color: #2a2850;
3   }
4 ▾ a:hover {
5     color: #332e8c;
6   }
7   |
```

You can use the following HTML markup as a basis to see the effect when working in a SassMeister session:

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Demo: Working with SASS maps</title>
  </head>
  <body>
    <div class="container">
      <a href="#">Link</div>
    </div>
  </body>
</html>
```

If you would like to learn more about using color maps, then these two articles are excellent starting points: `http://www.sitepoint.com/using-sass-build-color-palettes/` and `https://scotch.io/tutorials/aesthetic-sass-2-colors`. It's also worth taking a look at a CodePen example, of which there will be a few available online, for example, `http://codepen.io/davidkpiano/pen/0d74b9a54fb2ebc3c3be270284ed618d`.

Okay, onwards we go. Before we dive into building layouts, let's take a breather, and look at one of the more intriguing uses of Sass: working with filters. We can always apply these directly, but sometimes it's nice to combine various effects into a single filter that we can call on at any time. The Sassmatic library does a great job of this—let's take a look at how it works.

Applying filters using Sassmatic

Throughout this chapter, we've used the Sass core functions to create our colors. Before we launch into building something a little more complex, let's take a moment and have a little fun with filters.

Filters? I hear you ask. Yes, this does digress a little from what we have covered so far, but with good reason: in a sense, we're still manipulating the colors of content, this time using CSS filters. A perfect example of applying filters (albeit a couple of years old), is to use the Sassmatic library by Darby Brown, available at `http://sassmatic.com/`. Let's take a quick look at a demo:

> Please note that this demo works only in Webkit or Blink-based browsers; you will need to use Chrome to get it to work correctly.

1. From the code download that accompanies this book, dig out a copy of `colorfilters.html` and `colorfilters.scss` each, then store them in the right location in the project area. We also need the `shelby.png` image, so get that at the same time, and store it in our project area as well.

2. Go ahead and compile `colorfilters.scss`. To do so, fire up Command Prompt, then change to the project folder, and enter this command:

   compass watch --sourcemap

3. If we preview `colorfilters.html` in a browser, we should see the image appear, with the filter applied:
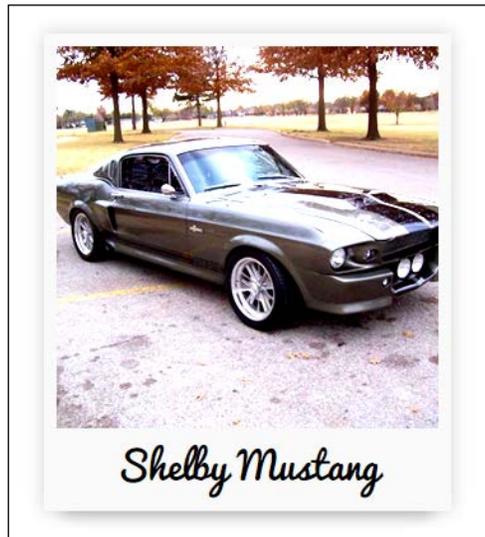
The Sassmatic site has a number of different filters that we can apply, or we can apply our own custom versions. For standard filters, all we need to do is import both Compass and the `_sassmatic` partial library, then use the Sass `extend` option to apply the filter to the image:

```
@import "compass";
@import "_sassmatic.scss";

...

div { @extend %filter-lomo; }
```

To get a feel of how the code works, have a look at the source code; the Sassmatic mixin library file source is available at `https://github.com/DarbyBrown/sassmatic/blob/master/sass/_sassmatic.scss`.

At this point, you may have noticed that we've used both mixins and functions at various places. I will bet that this is likely to be a little confusing, as it isn't always clear as to why you would use either functions or mixins (or even both).

There's a key difference that you should bear in mind to determine when you should use one over the other, or both. Let's take a moment to consider what this means for our code development.

Recognizing the difference between functions and mixins

A question: when is a mixin not a mixin, but a function?

It's an interesting point. After all, it would seem there is little difference between the two; they can both be used to a similar effect. However, there is a key difference; if we're writing a lot of styling code that can be reused, then we should use a mixin, such as in this example:

```
@mixin fontSize($size) {
  font-size: $size;
  font-size: calculateRem($size);
}

h1 {
  @include fontSize(32);
}
```

If, however, we wanted to apply a calculation to multiple elements of different types that can be stored as a variable, then a mixin won't work; this is where we need to use a function instead:

```
@function calculateRem($size) {
  $remSize: $size / 16px;
  @return $remSize * 1rem;
}

h1 {
  font-size: calculateRem($size);
}
```

> It is true that mixins can be used to perform calculations, but they will be tied to calculating a value for a particular style attribute, such as `font-size`.

This is an important concept to understand. When used correctly, it will help reduce the code you write; if you're not careful, it will lead to creating additional mixins that should be grouped into a smaller number.

Right, onwards we go. Time for something a little more complex, methinks! Constructing a layout might easily lead one to thinking that it requires a lot of effort, horrendous math, and the like. I'll show you that, with a little simple math, this cannot be further from the truth, at least for simple layouts.

Constructing layouts using functions and operators

If we look back at the code that we've explored so far, most of it has been relatively simple, and used to manipulate individual elements, such as single `<div>` boxes.

What if we wanted to do something larger, say construct a website? Can we take things further and use the power of Sass to build an entire layout for a site? Absolutely! The beauty is that, to create a simple site, we don't need a great deal of complicated math to do it either! Let's explore what this means for our layout.

> For this demo, we're using a header from `FreeWordPressHeaders.com`, at `http://www.freewordpressheaders.com`; if you would like to use something else, please feel free to change the code accordingly.

Let's make a start on our demo:

1. We'll start by extracting copies of `layout.html`, `styles.scss`, and `space.jpg`. Go ahead and save all of these into the relevant folders of our project area.

2. Try running the demo in a browser; if all is well, we should see something akin to this screenshot:

If we run the demo in a browser, we can see a simple, two-column layout—nothing outrageous, but still a perfect example of how we can get Sass to do our heavy lifting for us:

1. Open up a copy of `styles.scss`. At the head of the file, we should see this:

    ```
    $container: 100%;
    ```

2. A little further down, at around lines 3 to 5, and 7 to 9, we should see the following two functions—the first function uses the Golden Rule that we touched on earlier to work out the size of the main content area; the second calculates the width of the navigation area:

    ```
    @function calcContentWidth($container) {
      $contentWidth: round($container / 1.618);
      @return #{$contentWidth};
    }

    @function calcNavWidth($container) {
      $navWidth: round($container * 0.382);
      @return #{$navWidth};
    }
    ```

3. Further down, at around line `13`, we set the styles for our `<section>` area. Here, I've highlighted two rules of importance:

```
section {
  box-sizing: border-box;
  ...
  width: calcContentWidth($container);
  ...
}
```

4. A little further down, at lines `61` to `71`, we set the styles for our margin. Again, there are two rules to note — we use box-sizing: `border-box` to use the full height of the container, and the width is set using a Sass mixin:

```
nav {
  box-sizing: border-box;
  ...
  width: calcNavWidth($container);
  ...
}
```

So what is the significance of this code? Is this really all the code that we need to create our two-column layout?

Well, yes, this is the code we need to create our two columns. True, we need to apply some additional styles such as `box-sizing: border-box` to ensure that we have consistently sized content areas (and, of course, call our functions!), but mainly, these two functions and a single variable will do most of the work for us.

The work begins by setting the size of the main container in our markup; without this, the code will collapse into a big heap and not compile. In the two functions, we do a simple calculation to work out the size of each content area. This math is based on the Auron Ratio (otherwise known as the Golden Ratio), which forms the basis of a theory first popularized by Ethan Marcotte, who created what we now know as responsive web design.

It's a simple principle to grasp, but one that can prove incredibly useful. It's worth spending time getting familiar with how this principle works, so that you can build up a library of templates that use this principle, and reuse them for future projects.

Our demo covered two important concepts that we should explore in more detail. If you take a look at the code for the navigation, it looks OK, but we can do better. Enter the world of nesting — this helps us reduce the amount of code we have to write, when working with style rules that have multiple descendants. Let's explore how we can improve our code using this technique.

Reducing code repetition with nesting

In the layout demo, we added a simple navigation menu to the left side of the screen. It's not going to win any style awards, but serves to highlight one of the two key concepts that we used in our demo. The second is interpolation, which we will cover later in this chapter, but for now, let's look at the first, which is nesting.

Leaving aside any puns on what our feathered friends might do during the year (okay, terrible joke, I know), nesting is a useful feature to help reduce the amount of code that we need to write in our Sass style sheets.
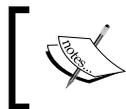
> For more information on nesting and use of the & as a placeholder, take a look at the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#nested_rules`.

If we take a look at the previous layout demo, we can see the code for our navigation, as follows:

```
#leftmargin {
  box-sizing: border-box;
  border-right: 1px solid #ccc;
  float: left;
  height: 575px;
  width: #{calc-sidebar(0.382, 800)}px;
  padding: 10px
}

#leftmargin li { list-style: none outside none; padding: 3px; }
#leftmargin a { text-decoration: none }
#leftmargin a:hover { background-color: #D78742; padding: 3px; }
```

It's a perfectly valid set of CSS rules, which will work fine. Notice how often we repeat the word `#leftmargin`, though?

We can easily rewrite our example to remove some of the perceived duplication, and give it a proper HTML5 semantic tag. If we use nesting, our code will look like this:

```
nav {
  box-sizing: border-box;
  border-right: 0.0625rem solid #ccc;
  float: left;
  height: 35.94rem;
  width: calcNavWidth($container);
  padding: 0.625rem;
  li { list-style: none outside none; padding: 0.1875rem; }
```

```
    a { text-decoration: none }
    &:hover { background-color: #d78742; padding: 0.1875rem; }
}
```

Although this may not reduce the number of lines required, it makes it easier to read, as we group all the descendant rules together. One small point to note, though, is that we can't begin a descendant rule with a semi-colon, which we would have for the `:hover` pseudo-selector. Instead, we have to use an ampersand; once compiled, Sass will replace the ampersand with the appropriate class or selector, as shown in the following image:

```
48  nav {
49    box-sizing: border-box;
50    border-right: 1px solid #ccc;
51    float: left;
52    height: 575px;
53    width: 29%;
54    padding: 10px;
55  }
56  nav li {
57    list-style: none outside none;
58    padding: 3px;
59  }
60  nav a {
61    text-decoration: none;
62  }
63  nav:hover {
64    background-color: #D78742;
65    padding: 3px;
66  }
67
```

Even though nesting is a simple concept, its simplicity belies its power. For a good example of how it can be used, take a look at the Pen by the Treehouse team, which is available at `http://codepen.io/Treehouse/pen/vEkit`.

A great example of the practical use of nesting is for creating buttons; a good button will change appearance when we hover over it, so there's a perfect use of `:hover`, `:active`, `:disabled`, and so on. To see what I mean, take a look at the demo by the team at Treehouse, at `http://blog.teamtreehouse.com/create-a-themable-button-set-with-sass`. It ties together a number of concepts that we've explored throughout this chapter.

Okay, time now for the second concept from our layout demo: let's change tack and look at using interpolation as a second form of placeholder when working with Sass.

Using interpolation

Cast your mind back to the layout demo that we created, where we used functions to determine the widths of our columns, based on a given size for the container of our content.

Okay, I know it wasn't too far back, but the demo used two important concepts. The first was nesting, and the second was interpolation. Although this is a simple concept to understand, it is nevertheless one that can be very powerful. Let's go through what this means.

Interpolation may seem like a scary topic, but it is nothing more than a simple way of inserting Sass variable values into a statement. Think of it as a placeholder; we used it to great effect within the demo, thus:

```
$container: 75%;

@function calcContentWidth($container) {
  $contentWidth: round($container / 1.618);
  @return #{$contentWidth};
}

section {
  box-sizing: border-box;
  float: left;
  height: 35.94rem;
  width: calcContentWidth($container);
  padding: 0.625rem;
}
```

When compiled, this translates to this CSS (as shown in a SassMeister session):
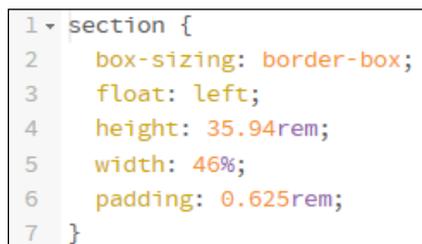
```
1 ▾ section {
2     box-sizing: border-box;
3     float: left;
4     height: 35.94rem;
5     width: 46%;
6     padding: 0.625rem;
7 }
```

The key concept is the use of #{...}, where the ... represents the content within the brackets. Anything within will be compiled, and the results inserted into the placeholder, as shown in our screenshot.

> To learn more about interpolation, take a look at the main documentation at `http://sass-lang.com/documentation/file.SASS_ REFERENCE.html#interpolation`; Tutsplus has a useful article on using Sass interpolation, which is available at `http://webdesign. tutsplus.com/tutorials/all-you-ever-need-to-know- about-sass-interpolation--cms-21375`.

Summary

Phew! What a tour! One of the key concepts of Sass is the use of functions and operators to create values, so let's take a moment to recap what we have covered throughout this chapter.

We kicked off with a look at creating color values using functions, before discovering how we can mix and match different functions to create different shades, or using external libraries to add extra functionality to Sass.

We then moved on to take a look at another key use of functions, with a look at defining different font sizes, using standard math operators.

Next up, we covered the basics of different palette types, as part of creating different palettes using Sass either through code, or by use of tools such as Adobe's Color CC tool. We dived off to cover the use of color maps as an introduction to better handling of colors, before pausing to take a look at using filters in Sass.

We then moved on to creating website layouts using some simple Sass calculations, which can be used as a basis for constructing any site layout. We then rounded out the chapter with a look at the two key topics used in the demo, that is, nesting and interpolation within Sass. Fancy taking a go at directing? No, I don't mean working at a theatre, but controlling the outcome of the code that Sass produces, using directives; this is the subject of the next chapter in our journey through the essentials of using Sass.

4
Directing Sass

One may be forgiven for thinking that we're about to direct a play. This chapter is not about plays, but how we can use any one out of a number of directives to perform an operation on our code. These range from extending styles to catering to different media or even choosing which one of several values to use based on the outcome of a calculation.

In this chapter, we will dive in and get accustomed to some of the more popular directives and see how, with some care and planning, they can prove to be a really powerful tool to use in Sass.

Throughout this chapter, we will cover the following topics:

- Working with the `@extend` directive
- Using and abusing multiple inheritance
- Making your site responsive
- Determining values based on the outcome of a condition check
- Avoiding repetition with the use of `@each`

Okay, let's make a start…

Working with the @extend directive

A question—how many times have you had to style an element, such as buttons, where you're to write the same styles several times?

I know that it's a real pain; although it gets the job done, it still leads to bloated CSS style sheets. What if we could avoid the need to add all this extra CSS? Well, we can. The trick to this is to use `@extend` within Sass.

Extends is a fantastic tool. At a basic level, we can pull out the common styles and then use them as a basis to create new rules that inherit this base group of attributes. Let's take a few minutes to explore how these work in more detail.

Consider this example:

```
.first .descendant { color: green; }
.second { @extend .first; }
```

When compiled to CSS, we get this:

```
CSS
1▾ .first .descendant, .second .descendant {
2    color: green;
3 }
```

In this case, we have a simple descendant style, which has green set as the font color. Instead of writing `.second .descendant { color: green }` as separate rules, we use `@extend` to effectively share the declarations from `.first .decendant` to what becomes `.second descendant`.
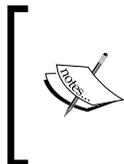
> A key part of using `@extend` is the `%` placeholder. This allows you to create styles that are *not* themselves compiled into CSS, but compiled when used to extend an existing style. For more information, take a look at the article by Guil Hernandez at `http://blog.teamtreehouse.com/extending-placeholder-selectors-with-sass`.

Using and abusing multiple inheritance

When creating CSS styles, how often do you find yourself creating rules that inherit from a parent tag? It doesn't matter if this is just one level up or several; multiple inheritance makes for bloated, unstable code that is difficult to debug.

If we change a style, it is likely to break others. Then, we will probably end up not being able to make a change because of the impact it has on other rules in the code. Sound familiar? To see what I mean, take a look at this screenshot. This is an example where code is being repressed at several different levels for this reason:

Instead, we can use the `@extend` function to move all of these styles into mixins; the preference is to use silent extends (such as `%primary-header`) so that we can keep unused styles out of our style sheet.

You may think that this would lead to code bloat — this is absolutely true — the key here though is that using extends will increase the longevity of our code; the benefits of this will far outweigh a little extra code in our Sass style sheets. Let's dive in and explore how to use the extend function and realize these benefits in our code.

> If you want to get into the details of the risks of abusing inheritance, then take a look at the great article by Micah Godbolt at `http://www.phase2technology.com/blog/used-and-abused-css-inheritance-and-our-misuse-of-the-cascade/`.

Exploring the benefits of using @extend

A key point when using `@extend` is that it is not the same as using standard mixins. When using a mixin, the rules that are created can be used against multiple different selectors with the option to customize the parameters that are passed to the mixin. With `@extend`, there are three key uses:

- **Inheritance** of rulesets between selectors that we can override with an extending selector
- **Traits** or shareable rulesets
- **Relationships** between selectors that are maintained when a selector with a defined relationship is extended via combinators (for example `>`, `+`, and `~`)

Okay, enough chit-chat. Let's move on and take a look at a simple demo based on how to create dialog boxes for use on a site.

Creating dialog boxes using @extend

When using `@extend`, the key to success is using it for styles that can be repeated; a perfect example is using it for buttons (such as the CodePen example at `http://codepen.io/Treehouse/pen/vEkit`) or dialog boxes to display messages to your visitors. We can of course spend lots of time extolling their virtues, but instead, let's cut to the chase and work through a simple example.

In our example, we will put together a number of simple dialog boxes; they will need a little more work in terms of styling, but for now, it's the basic principles of using `@extend` that is of interest to us. Okay, let's make a start:

1. We'll start by extracting a copy of extend.html and the three icons we need for our demo; they are `warning.png`, `info.png`, and `error.png`. Save the former HTML markup file in the root of our project area and the three icons in a subfolder called `img`.

2. Next, in a new file, add the following styles; we will start with the page's font styling:

```
body {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 13px;
}
```

3. This should be followed by a base `.box` style:

```
box {
  border: 1px solid #000; margin: 10px 0px; padding: 40px
    10px 5px 70px; background-repeat: no-repeat;
    background-position: 10px center;   width: 250px;
    padding-top: 30px;
border-radius: 3px; box-shadow: 3px 3px 4px 0px
    rgba(0,0,0,0.65); height: 80px; }
```

4. We then finish with the individual styles for the three dialog boxes:

```
.info {
  @extend .box;
  color: #00529B; background-color: #BDE5F8;
  background-image: url('../img/info.png');
}

.warning {
  @extend .box;
  color: #9F6000; background-color: #FEEFB3;
  background-image: url('../img/warning.png');
}

.error {
  @extend .box;
  color: #D8000C; background-color: #FFBABA;
  background-image: url('../img/error.png');
}
```

5. Save the file as `extend.scss` in the `sass` subfolder of our project area.

6. We now need to compile the file, so go ahead and open up Koala, then drag the `sass` subfolder on the left-hand side of the window. Koala should refresh the screen, displaying the names of each of the files in the folder; for now, it will only show one file:

7. Right-click on the filename and select Compile. If all is well, we should see the **Success** message appear; if we then preview the results of `extend.html` in a browser, we should see the following result:

This is a great technique to reduce the code we need to write if we're creating similar styles for elements (such as buttons or dialog boxes). There are other uses as well; how about applying animation? The key here is working on what is common to each of the chosen elements and then grouping these into a common rule before adding the extras at the end.

To really get a feel of what is happening, open up your browser and then take a look at the code via a DOM Inspector, as shown in the following screenshot:

Notice how most of the styles we've used for each box are grouped together? The styles that are specific to each box (for example, the background color or icon) are then kept in individual styles that complement our basic style.

> Miguel Camba, the developer, goes through the differences between mixins and the `%` placeholder at length. His article is available at `http://miguelcamba.com/blog/2013/07/11/sass-placeholders-versus-mixins-and-extends/`. He presents a good case for why using `@extend` isn't always the right thing to do.

This scratches the surface of what is possible when using `@extend`; to really understand what is possible, take a look at the main documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#extend`. It's also worth reading the article by David Khoursid at `http://www.smashingmagazine.com/2015/05/04/extending-in-sass-without-mess/`. It will take you through how it works and give some tips on best practice.

In the meantime, let's move on. Sass supports the use of a number of directives; we've covered two in the form of `@import` and `@extend`. Before we move on and cover another in the form of `@media`, there is something we should consider for a moment, that is, we should not abuse the use of `@extend` when inheriting styles; let's see what this means in terms of performance, and how we can reduce the impact using Sass.

Making a site responsive with @media

How many times have you accessed the Internet from a mobile device? I'll bet that asking a random selection of people this question will show that a good number of individuals or groups will have done just that—it doesn't matter if it is a laptop, iPhone, or iPad—the user experience on a mobile device will still be different to that of a normal desktop.

Some sites manage to create a user experience that has worked very well across a number of devices. For inspiration, take a look at `http://www.mediaqueri.es`. This showcase illustrates why it is crucial that responsive design is done well.

For those (Sass-based) sites that have yet to embrace it, the good news is that it is easy to fix; we can add media queries to control what is displayed at each breakpoint for different viewport sizes. The not so good news is that while it is a cinch to add the breakpoints, it's up to us to decide what should be displayed; this can be a real minefield.

As a test, try previewing the sample layout page we constructed back in *Chapter 3,
Building Functions, Operations, and Nested Styles*. In most recent browsers, if we enable
the Responsive Design mode, we can clearly see that the site doesn't resize well
when viewing from a smaller device. It will appear cut in half; images too won't
resize well:
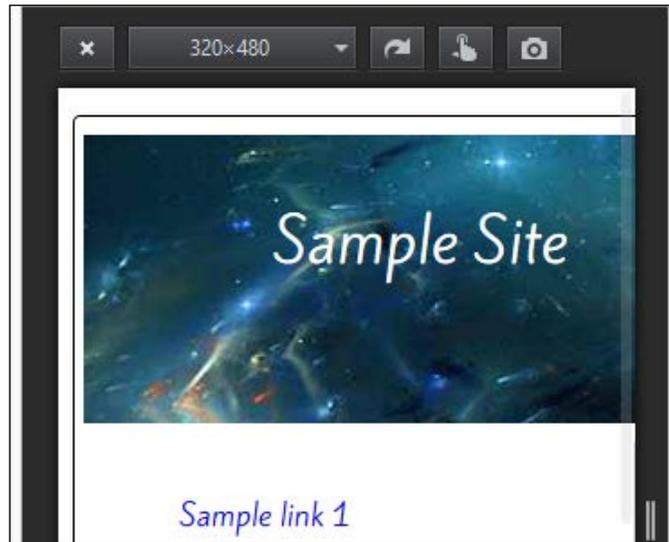
See what I mean? Both images and content play an equally important role; we can't
cater to one without the other if we want our site to be successful.

> Dealing with media (such as images) can be more complex when
> working responsively; to get a feel of what is involved, take a look
> at my book, *Responsive Media in HTML5,* which is available at Packt
> Publishing. Many of the techniques in that book can equally be used
> when working with Sass.

The great thing about Sass is that we can use it to create media queries in various
ways. In the next few pages, we will concentrate on how to work with pages; we
will cover the use of Sass in CSS grids in *Chapter 5, Incorporating Sass into Projects*.

Historically, many people may simply create variables, which are used to build
media queries. As a start, we can create media queries in Sass using something
akin to this:

```
@media screen {
  .sidebar {
    @media (orientation: landscape) {
```

```
        width: 500px;
      }
    }
  }
```

This will compile to the following CSS:

```
@media screen and (orientation: landscape) {
  .sidebar {
    width: 500px; }
}
```

This will work perfectly well, but it seems a little overkill, particularly if we have to specify two @media statements to produce one @media rule.

> For more details on how Sass treats @media queries, take a look at the main documentation at http://sass-lang.com/documentation/file.SASS_REFERENCE.html#media.

We can take things further using variables; this allows you to remove some of the values from within the code and store them in a separate partial file, which we can import into our code:

```
$desktop: "(min-width: 600px) and (max-width: 800px)";

@media #{$desktop} {
  .foo {
    background: tomato;
  }
}
```

This is a perfectly valid option to create mixins, but it still seems a lot of work. Of course, we can build a library of media query variables that we can reuse at a later date. A cleaner option is to use a mixin that can handle the heavy lifting for us, leaving us to simply specify the media format (such as all) and the breakpoint value, which can be set as a variable. There are examples already available online, but a popular choice is the Breakpoint Sass mixin at http://www.breakpoint-sass.com. Let's take a look at how to install it and make it ready for use.

Installing the Breakpoint mixin for Sass

The Breakpoint mixin available at `http://breakpoint-sass.com/` can be installed either as a Ruby gem or using Bower; for the purpose of this exercise, we will use the latter method:

1. Bring up the Node.js command prompt and change the working directory of our project area.

2. At the prompt, enter this command and then press *Enter*:

   bower install breakpoint-sass

3. If all is well, we should see this appear in the Node.js command prompt:

At this stage, we are now ready to add media queries to our Sass code. Now, it's time for a demo. For our next demo, we will retrofit responsive capabilities to a reworked version of the simple layout page we created back in *Chapter 3*, *Building Functions, Operations, and Nested Styles*; let's take a look at what is involved.

Retrofitting responsive capabilities to a page

To succeed in this age of mobile devices, it is critical that a site be responsive; for some sites, we have the luxury of building this in the design from the ground up. However, for many, it is more likely to be an afterthought and has to be incorporated post construction.

For our demo, we will follow the latter route because this matches reality. We will use a reworked version of the simple layout page we created back in *Chapter 3, Building Functions, Operations, and Nested Styles*, as shown in this screenshot:

As a precursor to adding responsive capabilities, try running the `layout.html` file in the `layout - non responsive` folder from the code download that accompanies this file; see, how it doesn't resize well?

Let's make a start on fixing it by adding some media queries to handle smaller viewports:

1. We'll begin by extracting a copy of the `layout - SASS responsive` folder from the code download that accompanies this book and then save it in our project area.

2. We need to import the Breakpoint mixin and define two variables to support it, so go ahead and open up a copy of `layout.scss`; at the head of the file, add these three lines, as shown in the following code:

```
@import "bower_components/breakpoint-sass/stylesheets/
  _breakpoint.scss";

$phone: 'screen' 320px;
$tablet: 'screen' 480px;

@font-face {
```

3. Let's now add the first of our breakpoints; in the #container rule, immediately after min-height: 100px and before the closing bracket, leave a blank line and then add these rules, as shown in the following code:

```
min-height: 100px;

#container
  //media query - screen and 320px
  @include breakpoint($phone) {
    #container { max-width: 800px; background: #ccc; box-
      shadow: none; margin: 0 auto; }
  }
```

4. Further down, we need to add a similar rule for section. This time, leave a blank line underneath box-sizing: border-box; and then add this:

```
//media query - screen and 320px
@include breakpoint($phone) {
  section { float: left; }
}
```

5. Our penultimate stop through the code is in the aside section; go ahead and modify the code as follows:

```
  &:hover { background-color: #D78742; display: block; }
}

//media query - screen and 320px
@include breakpoint($phone) {
  aside {
    float: left; clear: both;
    li { width: 103%; }
  }
}
```

6. At the end of the file, we need to add another media query; to show how versatile the Breakpoint mixin is, we can simply add the second breakpoint at the end of the code as follows:

```
//media query - screen and 480px
@include breakpoint($tablet) {
  #container { max-width: 800px; background: #ccc; margin:
    5% auto; overflow: auto; box-shadow: 4px 4px 5px 0px
    rgba(0,0,0,0.60)
  }

  section { width: 72.5%; float: left; height: 575px;  }
```

```
aside {
  float: left; width: 27.5%; height: 575px;
  li { list-style: none outside none; padding: 3px; }
  a {
    text-decoration: none;
    &:hover { background-color: #D78742; display: block; }
  }
}
}
```

7. Save the file and then compile it using Koala. If all is well, we should see this when we preview the results in a browser; this screenshot shows the browser window resized to the smallest size:

Good. Our site is now responsive; of course, it needs more work to define additional styles, but at least we have a basis we can improve on.

> Talking of improve on; that image looks rubbish on my iPhone 6; can we do any better? Absolutely! We can use @media to add hi-res image support to our page. This is one of many ways we can customize our site using Breakpoint. I'll leave you to work out how.

Okay, let's change focus. A key feature of authoring media queries in Sass is its ability to bubble the media query out of the nested statement and back up to the root of the style sheet. Let's take a few minutes to explore this in more detail and see how Sass can help us group related styles and media queries together when writing code.

Exploring media bubbling in Sass in more detail

Take another look at the `layout.scss` file that we created when we added media queries to our responsive page. Note that we've only added two queries to our code; notice something about *how* the queries have been added?

The eagle-eyed among you should spot that we've put one media query in line with existing styles, as shown in this screenshot:

```
29   #container {
30     border: 1px solid #000;
31     border-radius: 4px;
32     background-clip: padding-box;
33     margin: 5% auto;
34     overflow: hidden;
35     white-space: normal;
36     min-height: 100px;
37
38     //media query - screen and 320px
39     @include breakpoint($phone) {
40       #container { max-width: 800px; margin: 0 auto; background: #ccc; box-shadow: none; }
41     }
42   }
```

This is a perfect example of **media query bubbling**; if we add a media query inside a nested selector, Sass is intelligent enough to bubble (or lift) that media query out of the nested statement and place it at the root of our style sheet. It's a great technique to keep media queries next to the original rules. Here's how our Sass example compiles to valid CSS:

```
33   #container {
34     border: 1px solid #000;
35     border-radius: 4px;
36     background-clip: padding-box;
37     margin: 5% auto;
38     overflow: hidden;
39     white-space: normal;
40     min-height: 100px;
41   }
42   @media screen and (min-width: 320px) {
43     #container #container {
44       max-width: 800px;
45       margin: 0 auto;
46       background: #ccc;
47       box-shadow: none;
48     }
49   }
```

It does raise a question of which one should be used. Some developers consider it more practical to incorporate styles inline so that we reduce the number of code we need to write.

Others will argue that media queries are better kept separate, although this is at the expense of "doubling up" styles (in this case, `#container` would feature twice in the code). Is there a right or wrong answer? It all boils down to personal preference; with Sass' ability to compile code from multiple files, we could even hive off the code elsewhere.

> We've used Breakpoint for Sass for our demo. There are many more ways we can use the library to great effect; to get a taste of it, take a look at the presentation by Sam Richard at `http://snugug.github.io/RWD-with-Sass-Compass/`.

This is one of the many ways of adding responsive support to our sites; there are literally dozens of alternatives for us to try. Bear in mind that not all will cover every aspect of a site, so we will probably need to use multiple methods. Take a look at these as a starting point:

- Guil Hernandez has written an intriguing article on the use of the new viewport-relative units (such as `vw`, `vh`, `vmin`, and `vmax`) when working with Sass. These are worth looking at once you're more familiar with how to create responsive media queries. Take a look at `http://sassbreak.com/viewport-relative-headings-with-sass/` for more details.

- David Walsh put together an article (available at `http://davidwalsh.name/write-media-queries-sass`) that outlines similar principles to Mason's online post. David uses the principles of variables; it's a good basis to develop our own custom mixins once we become more familiar with creating media queries in Sass.

- Navigation is another key area that will benefit from the addition of responsive capabilities. Jason Kinney, author of a comprehensive mixin library at `http://responsive-sass-nav.com/` takes you through how to rework your navigation to automatically resize at various different breakpoints.

- We cannot forget the well-known Sassaparilla responsive library at `http://sass.fffunction.co/`. This library is worth exploring once you are more familiar with how to use Sass to manage your responsive design needs.

Okay, let's move on. I'm a great fan of keeping things simple. Sometimes though, there may be occasions when we can't always produce a nice set of styles from the outset, but have to let Sass compile the code based on satisfying one or more conditions. There is scope to really go to town when using this method, so let's dive in and see what this means to compile Sass.

Controlling the outcome of our code

Controlling the outcome of our code when using Sass can be a double-edged sword. It's an immensely powerful technique to use to determine what should happen if one or more values are used; its power means that it is not recommended for use, except in constructing mixins.

Although control directives are a more advanced feature of Sass, it is worth familiarizing ourselves with the basics; we can then develop our use of them once we are more familiar with how they work.
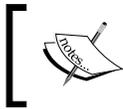
> For more details on how to use control directives, take a look at the main documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#control_directives__expressions`.

Sass contains a number of control directives, of which these are the most likely to be used:

- `@if`
- `@for`
- `@each`
- `@while`

The best way to illustrate how these directives work is in the form of a demo; we will work our way through two examples a little later in the chapter. For now, let's take a few minutes to explore each of these directives in more detail so that we can see how they work.

Using the @if directive

The first of our directives is based on the classic if *X* equals *Y*. Then, we'll do this or we can do something else instead. This is perfect to create scenarios where we want to have a single block of code that can be reused multiple times, but give a different answer based on matching a criteria.

So, what does this mean? Take for example this example of two buttons; we have one mixin, in which we can pass suitable values to produce different styles for each button based on the `lightness()` value for the given colors:

```scss
@mixin button-gradient($hex, $color: #fff) {
  color: $color;

  @if (lightness($hex) - lightness($color)) > 30 {
    text-shadow: -1px -1px 1px rgba(255, 255, 255, 0.5);
  } @else if (lightness($hex) - lightness($color)) < -30 {
    text-shadow: 2px 2px 2px rgba(0, 0, 0, 0.5);
  }

  @include background-image(
    linear-gradient(
      lighten($hex, 15%), lighten($hex, 5%), $hex, darken($hex, 3%)
    )
  );

  &:hover {
    color: #ff3;
    background-color: lighten($hex, 0%);
  }

  &:active {
    background-color: darken($hex, 5%);
  }
}
```

In this case, we will work out the difference between the `lightness()` values for two colors and then apply a `text-shadow()` style based on the results of our calculation. To see the results of our `@if()` statements working, copy the contents of `buttons. html` and `buttons.scss` to a SassMeister session.

> This should not be confused with the `if()` function; this is an inbuilt function that can only return one of two possible values.

Working with the @for directive

If we know that we need to apply the same styles to multiple items throughout, then using a @for statement is perfect. We can set up a count and use it to apply the same style for each of our items. In this example, we will create CSS styles to set a varying width for three boxes:
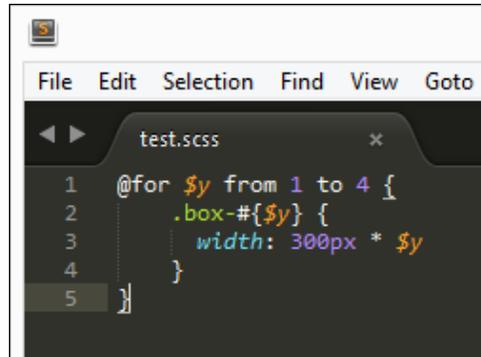
When compiled, it produces the following CSS rules:

Notice anything? Surely there should be *four* boxes, right?

In this case, no; a quirk of Sass is that we can use both from...through... and from...to... when counting through each style. The former includes the starting and finishing values (that is 1 to 4), whereas the latter does *not* include the end value, so in this case, it would be 1 to 3.

Exploring the @each directive

The @each directive takes a slightly different approach when counting through items; it will count through each item (similar to @for), but these items are normally stored in a comma-delimited list or Sass map.

> Sass maps are a relatively new function. For more details, take a look at the official documentation at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#maps`.

A perfect example of this is applying the same styles to a series of images, such as social media icons that you may see on a website, as shown in this screenshot:

```scss
@each $social in linkedin, facebook, twitter {
  .#{$social}-icon {
    background-image: url('img/#{$social}.png');
  }
}
```

```css
.linkedin-icon {
  background-image: url("img/linkedin.png");
}

.facebook-icon {
  background-image: url("img/facebook.png");
}

.twitter-icon {
  background-image: url("img/twitter.png");
}
```

Here, we will apply the same styles to three icons, namely, `.linkedin-icon`, `.facebook-icon`, and `.twitter-icon`, which will represent social media icons of the same name. We've used string interpolation to great effect, replacing `$social` with the name of the social media service when applying the background image.

Enough with the chit-chat; I feel a demo coming on! Let's make use of some of these directives; what better than to create some simple social media images?

Getting social in Sass

How many times have you visited a site only to see at least one button or image that depicts a link to a social media site…? I'll bet the answer is a fair few, right?

The explosion of social media services, such as the likes of Facebook, Twitter, or Instagram makes it hard to escape. If we can't beat them, we may as well join them!

Creating a social media icon is a cinch when working with Sass. We can create a template and use this as a basis to create each button. Using a map and string interpolation, we can generate different styles for each button from the same template. Let's dive in and take a look at what is involved:

1. We'll begin by extracting a copy of `social.html` from the code download that accompanies this book, so go ahead and save this in the root of our project area.

2. We now need to set up the Sass rules that will become our CSS. In a new file, let's add the styles that we need. There is a good chunk of code involved, so we'll go through it in sections, beginning with the variables that define our button colors:

```
// Variables for 'social network' colors
$Twitter: #41b7d8;
$FB: #3b5997;
$GPlus: #d64937;
$LinkedIn: #0073b2;
```

3. Next, we will add some simple styling for the unordered list that we will use to form the basis of our buttons:

```
ul { padding: 0; }
li { float: left; list-style: none; }
```

4. We're almost at the magic point that makes it all work. Then, comes the base style for each button. Here, we will use nesting to great effect to provide the background for each button's icon, turn each into a link, and adjust the position for the first button:

```
.social-link {
  font-family: verdana, arial, helvectica, sans-serif;
    width: 22.5%;
  height: 42px; letter-spacing: 1px; line-height: 42px;
    margin-left: 0.3%; position: relative;

  &:first-child { margin-left: 0; }
  &:before { background-color: white; content: ''; height:
    34px; left: 4px; position: absolute;
    top: 4px; width: 34px; }
  a {
    color: #FFF; display: block; padding-left: 55px; text-
      shadow: 1px 1px 0 rgba(black, 0.4);
    text-transform: uppercase; text-decoration: none; font-
      size: 1rem;
  }
}
```

5. Now, the real magic begins. We first define a Sass map to store the names of each button before working our way through each to set the background color, apply a small shadow effect, and add an image to each button:

```scss
$social: (twitter: $Twitter, facebook: $FB, linkedin: $LinkedIn,
  googleplus: $GPlus);

@each $social, $color in $social {
  .social-link--#{$social} {
    background-color: $color; transition: background-color,
     .5s, ease-in; margin: 5px; box-shadow: 3px 3px 5px 0px
      rgba(0,0,0,0.75);
    &:focus, &:hover {
      background-color: darken($color,10%);
    }
    &:before {
      background-image: url('../img/#{$social}.png');
    }
  }
}
```

6. Save the file as `social.scss` and compile it using Koala. If all is well, we should see these buttons appear when previewing the results of our work in a browser window:

Perfect! We've created a relatively small block of code that can be reused to create simple buttons of different types for different occasions. The key to making this work is twofold; it's the use of nesting to create `:hover` and `:focus` selectors and the use of the Sass map and `@each`. All this helps to bring it all together.

To see how effective these techniques are, we can see the CSS styles that have been generated with a DOM inspector, such as Firebug:

```
element {                                                    inline
}
.social-link--linkedin {                              social.scss:27
    background-color: ● #0073B2;
    ▸ transition: background-color 0s ⟳ ease 0s, all 0.5s ⟳
ease 0s, all 0s ⟳ ease-in 0s;
    ▸ margin: 5px;
    box-shadow: 3px 3px 5px 0px ● rgba(0, 0, 0, 0.75);
}
.social-link {                                        social.scss:11
    font-family: verdana,arial,helvectica,sans-serif;
    width: 22.5%;
    height: 42px;
    letter-spacing: 1px;
    line-height: 42px;
    margin-left: 0.3%;
    position: relative;
}
li {                                                   social.scss:9
    float: left;
    ▸ list-style: outside none none;
}
```

Alex Wolfe and Rob Levin (developers) have used control directives to great effect in creating the Buttons library that is part of the Unicorn UI framework. You can see their efforts at `http://unicorn-ui.com/buttons/`.

We're not limited to producing elements (such as buttons); control directives such as `@each` can easily be used to produce more complex styles; a perfect example is animation. To see what I mean, let's take some time out and produce a little demo, which fades out a series of boxes on a point-second time delay.

Building animations using Sass

Let's take a look at any recently built site on the Internet, and chances are that it will have animation incorporated into the design in some form. There's a likelihood that it will be using jQuery; the smart option though is to use CSS animations, provided the design allows you to use it.

Creating animations can be fun, but at the same time, it can be a little repetitive; some browsers still require vendor prefixes to be added, which makes for bloated code. Instead of writing each rule manually (which is so old-school!), we can use the power of Sass' `@each` control directive to do the hard work for us. Let's take a look at how using a demo produced by Yonatan Woloweslsky becomes a basis for our exercise:

> You can see the original article by Yonatan at `http://www.everyglobe.biz/getting-started-with-css-animations-using-sass-mixins/`.

1. We'll begin with extracting a copy of `animation.html` from the code download that accompanies this book. Go ahead and save it in our project area. This provides some simple markup for the elements we will animate using Sass. We'll also need a copy of jQuery as well; drop this in the `js` subfolder of our project area too.

2. Our demo uses a little jQuery to help fire off the animations from the text link in the page. In a new file, add the following code and save it as `animation.js` in the `js` subfolder of our project area:

```
var app = function() {
  $('a').click(function() {
    var animation = $(this).attr('data-animation');
    $(this).parent().find('.animated-element')
      .toggleClass(animation);
  })
}

$(document).ready(app);
```

3. Then comes the all-important styles. To animate our content, we will use a blend of mixins, variables, string interpolation, and control directives to produce our rules. Let's start with the mixins. In a new file, add the following code; the first adds vendor prefixes where needed and the second is a simple transition mixin. Note that we will only include the `-webkit` prefix in the first mixin because most browsers do not require prefixes, the only exception being Safari and Safari Mobile:

```
@mixin prefixer($property, $value) {
  -webkit-#{$property}: $value;
  #{$property}: $value;
}
@mixin transition ($value...) {
  @if length($value) >= 1 {
```

```
    transition: $value;
  } @else {
    transition: all 0.2s ease-in 0.05s;
  }
}
```

> The transition values used are not the same; to get around this, Sass allows you to use ... to specify a variable argument. Any leftover arguments will be picked up in the form of a list.

4. Next, we will add some simple styling for our elements; we will set the size and décor of our animated boxes and then call the transition mixin we created in the previous step:

```
.animated-element {
  height: 100px; width: 100px; border-radius: 2px;
    background: #ff6666;
  @include transition();
}
```

5. When animating our boxes, we want them to move upwards while fading out; this rule takes care of the animation effect:

```
.fade-out-top {
  opacity: 0;
  @include prefixer(transform, translate(0px, -20px));
}
```

6. For this next rule, we will animate each of the boxes one by one, but with a point-second time delay:

```
.one-by-one {
  display: inline-block;
  $delay-values: (0.0s, 0.1, 0.2s, 0.3s);
  @for $i from 1 through length($delay-values) {
    &:nth-of-type(#{$i}) {
      transition-delay: nth($delay-values, $i);
    }
  }
}
```

7. We then finish our style sheet with some simple rules to help as follows:

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

.container { max-width: 960px; margin: auto; padding: 10px
  30px; }

section {
  padding: 50px 0;
  a { display: block; }
}

a {
  color: #1698C7; text-decoration: none; cursor: pointer;
    margin-top: 20px;
  &:hover {
    text-decoration: underline;
  }
}
```

8. Save the file as `animation.scss` in the `sass` folder and then use Koala to compile the file to produce our CSS style sheet.

9. If we preview the results of our work, we should see the boxes animate upwards when you click on **Toggle Animation**; the following screenshot shows the boxes on the way back down once **Toggle Animation** has been clicked on for a second time:
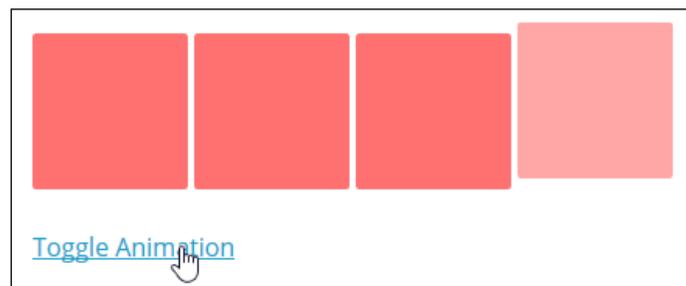
Exploring the key to our animations

Our animation uses three key concepts. We've touched on two of them earlier in the book in the form of nesting and Sass maps; the concept that makes all this work is shown in the following code:

```
.one-by-one {
  display: inline-block;
  $delay-values: (0.0s, 0.1, 0.2s, 0.3s);
  @for $i from 1 through length($delay-values) {
    &:nth-of-type(#{$i}) {
      transition-delay: nth($delay-values, $i);
    }
  }
}
```

Our rule sets up a Sass map that stores the various time delay values that we will use. Of course, we can use a simple calculation to work this out on the fly, but using a Sass map at least makes it easier to know what these values are from the start. We then count our way through each of the time delay values, using them to create a number of repetitions of the `.one-by-one` class, which contains a different `transition-delay` value for each instance. We can see this in the compiled CSS style sheet:

```
13    .one-by-one {
14      display: inline-block; }
15    .one-by-one:nth-of-type(1) {
16      transition-delay: 0s; }
17    .one-by-one:nth-of-type(2) {
18      transition-delay: 0.1; }
19    .one-by-one:nth-of-type(3) {
20      transition-delay: 0.2s; }
21    .one-by-one:nth-of-type(4) {
22      transition-delay: 0.3s; }
```

Our example was relatively simple, so we can see the essential elements required to use Sass to produce animations. To get a flavor for what is required when building more complex animations with Sass, take a look at the article by Hugo Giraudel at `http://hugogiraudel.com/2014/07/16/automating-css-animations-with-sass/`. Take note though that there is some fairly complex math involved!

Summary

Working with directives in Sass opens up a world of possibilities, although we should take care over how we apply techniques when writing our code. We've covered a number of different techniques throughout this chapter. Let's review what you've learned.

We kicked off our journey with directing Sass and looked at how to use the `@extend` directive, where we saw how easy it is to abuse CSS inheritance when writing styles, and how to use `@extend` to help reduce the resulting code bloat. We covered the benefits of using `@extend` and then put it to great effect by creating a simple dialog box demo that extended an existing base style.

Then, we looked at how to use another directive in the form of `@media` and discovered how easy it is to create media queries in Sass; we also looked at the different options available. We explored using the Breakpoint Sass mixin and how to apply it retrospectively to a site, followed by a look at how to provide hi-res image support as one means of using the mixin to manage responsive images.

We then moved on to take a look at controlling the output of our code, using directives such as `@if` or `@for`. Briefly, we looked at each in turn, followed by how to use some of them to build some social media buttons. We then rounded out the chapter with a look at how to apply similar techniques to manage animations, which are more complex, but still just as easy when using Sass' control directives.

It's time to move on. I'm sure that some of you will wonder how easy it is to incorporate Sass into your own projects, right? So far, we've concentrated on the ground up approach. Now, it's time to take a look at how to retrofit Sass support to existing projects, which we will cover in the next chapter.

5
Incorporating Sass into Projects

A key critical question that will be asked by many when using a technology for the first time is how can I apply it to my projects?

The beauty about Sass is that as a superset of CSS, it can be used almost anywhere; if you have CSS that needs to be styled in a site, then chances are that we can use Sass. This may be anything from a single page application site to a complex CMS or e-commerce system.

From a simple one page application to a full-sized content management system, in the next few pages, we'll touch on some of the tips and tricks we can use to migrate any existing site to use Sass. After all, not every project will have the opportunity to use Sass from the start! Throughout this chapter, we will cover the following topics:

- Converting existing style sheets to use Sass
- Incorporating CSS grids
- Applying Sass to frameworks, such as Bootstrap
- Using a CMS system
- Working with the Rails/Rack/Merb plugin

Ready to get stuck in?

> Some of the exercises will require prior knowledge of the application we're working with; if you're unfamiliar with some of the basics, such as installing WordPress plugins, then there are plenty of sources online that will help get you up to speed.

Converting existing projects to use Sass

At this point, I'd be willing to bet you're asking yourself a question, that is, "so far, we've looked at how to use Sass in new projects, where we've built the site from ground up. However, I have a large WordPress site that already uses a huge CSS style sheet. So, how do I move to using Sass?"

It's an excellent question. We can always incorporate Sass from the ground up, but not all of us have the luxury of building it from scratch; some of us have to have existing projects that are complex. Never fear! To start with, incorporating Sass is actually really easy. How complex it becomes will of course depend on how far you want to go with converting each rule to its Sass equivalent. To prove how easy it's to get started with incorporating Sass, all we need to do is, of course, install Sass (goes without saying!). Then, install Compass; we will use this to control how our Sass files are compiled.

Once installed, we'll use the `config.rb` file (provided with Compass) to control how to compile our Sass files. Enter this at Command Prompt:

```
gem install compass
```

We will cover the creation of `config.rb` files in more detail later in this chapter.

We now have the software installed, so we could simply rename our CSS file with a Sass `.scss` extension, but if we want to really make full use of Sass, then there is a little more to the process.

The key to using Sass though isn't in a simple file rename, but in how we structure our Sass files. Don't forget that Sass can easily combine multiple Sass files into one valid CSS style sheet; the effectiveness of our style sheet will depend on how we construct our partials. For example, we may have ten partial files, such as `_buttons. scss, _icons.scss, _responsiveimages.scss`, and so on. We can store these in the `_partials` folder. Then, set our main style sheet to import these at compilation.

There are a few useful tips we can use when it comes to incorporating Sass into our existing projects. Let's take a look at how we can make our lives easier:

- Don't try to do it all in one go—it doesn't matter how large the CSS style sheet is—it's much better to convert the file piece by piece; converting a large file in one go is very likely to be less rewarding. Also, you will lose momentum during the process.

- The key to an effective process is organization. It is as much about how we organize our Sass files as the code we write within. We will cover this in detail later in the chapter when we look at how to convert a WordPress theme to use Sass.

- There are some really useful tools that will help with conversion. Two good examples are CSS2Compass (available at `http://sebastianpontow.de/css2compass/`) and CSS2SASS (at `http://css2sass.herokuapp.com/`). The tools are good to get to grips with Sass initially, but ultimately don't produce the cleanest code. There is nothing wrong with starting small and simple, such as a simple mixin for a particular CSS3 style or perhaps concentrate on setting up variables for colors. If it helps, try copying code in a `http://www.sassmeister.com` session; you can immediately see the impact of any change when compiling your code and easily undo it if it doesn't go according to plan.

- In *Chapter 1*, *Introducing Sass*, we looked at how to add support for Sass to text editors (such as Sublime Text) and then used it on Koala to compile our code. These are perfectly valid ways of compiling Sass, but sometimes, it is better to simply stick to the command line when working with Sass for the first time. Granted that it may seem scary, but it is worth the effort when it comes to familiarizing yourself with how to use it. For example, some GUI-based applications are not able to compile Compass. Also, remember the bug that we came across when compiling Compass code in Koala.

Alright. Now that we've seen some of the tricks we can use to convert to use Sass, let's put some of it to good use and apply them to converting a simple demo, which showcases some of the books available at Packt Publishing.

> To get a feel of a typical journey when converting to use Sass, take a look at the blog post by Rory Douglas at `http://terrificwebdesign.net/converting-css-sass/`; this is definitely worth a read.

Incorporating Sass into existing sites

The beauty about Sass is how we can use it to produce any type of style we need for our projects. Although the steps we take may differ from project to project (for example, using a task runner to compile or a GUI application such as Koala), the end result is still the same: producing valid CSS style sheets.

To prove how versatile Sass can be, we will take a look at a number of different uses for Sass. Let's begin with a simple demo based on a tutorial by Alessandro Atzeni. This demo displays a number of books with an overlay displayed when hovering over one of the images.

> You can see the original tutorial at `http://tympanus.net/` `codrops/2011/11/02/original-hover-effects-with-css3/`. It's a few years old, but still has a great effect.

It's a simple exercise (and yes, a shameless plug for my books!); let's take a look at the effect we're going to produce:
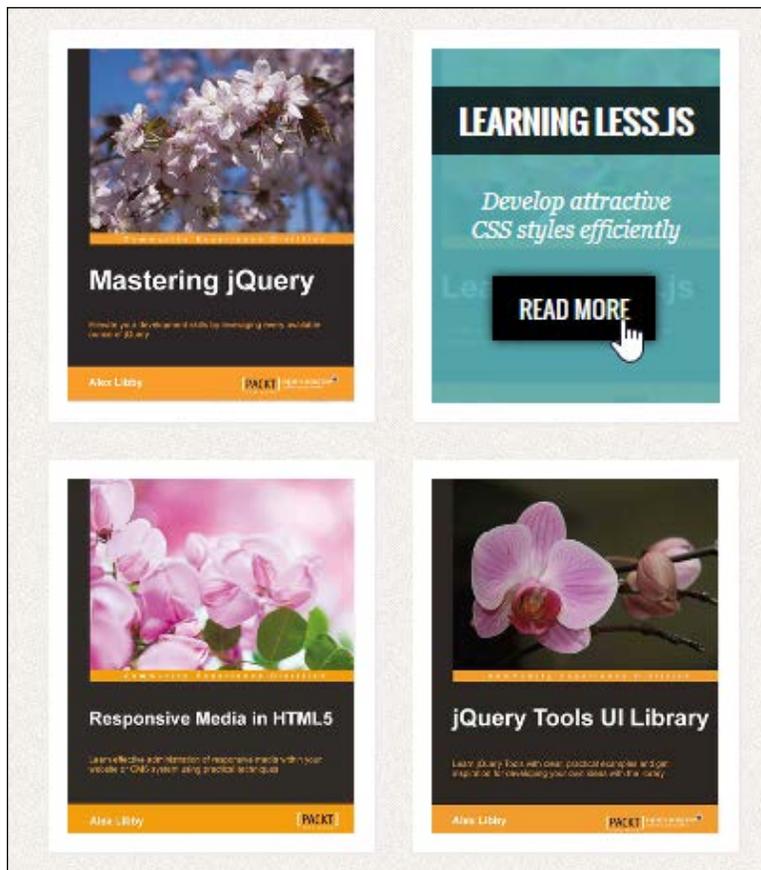
Okay. Let's dig in and make the changes; we'll start with the easiest change, which is switching the code to using Sass. Perform the following steps:

1. We'll begin by extracting a copy of the `original site` folder from the code download that accompanies this book; save this to our project area.

2. Then, we need to set up a project area somewhere to store our Sass files, so go ahead and create a new folder called `sass` in the original folder.

3. Next, copy the `styles.css` and `transitions.css` files to the sass folder; go ahead and rename them as `styles.scss` and `transitions.scss`.

4. Crack open `styles.scss`. At the head of the file, add this mixin; this will display fonts in rem units with a fall back on pixel values if not supported:

    ```
    @function calculateRem($size) {
      $remSize: $size / 16px;
      @return $remSize * 1rem;
    }

    @mixin font-size($size) {
      font-size: $size;
      font-size: calculateRem($size);
    }
    ```

5. We can now make use of this mixin; look for this on or around line 5:

    ```
    font-size: 13px;
    ```

    Change it to `font-size: calculateRem($size);`.

6. We need to make one more change to font size, so go ahead and change the `font-size` rule in `.view p` to this:

    ```
    @include font-size(12px);
    ```

7. Next, look for this on or around line 36:

    ```
    .view .mask {
    ```

8. Add it to the end of the `.view` style rule, as shown in the following code:

    ```
      background: #fff url(../img/bgimg.jpg) no-repeat center
        center;
      .mask { width: 9.5rem; height: 11.69rem; position: absolute;
        overflow: hidden; top: 0; left: 0; }
    }
    ```

9. We'll do the same for the `:hover` pseudo selector on `.view a.info`; the relevant code is on or around line 76. Note that we've replaced `.view a.info` with an ampersand placeholder:

```
box-shadow: 0 0 0.0625rem #000;
&:hover { box-shadow: 0 0 5px #000; }
}
```

10. Save the file. Open up `transforms.scss` and replace all the code with the following (we will start with a mixin to build our transform animations:

```
@mixin transform ($direction, $amount) {
  transform: #{$direction($amount)};
  opacity: $opacity;
}
```

> We're using this purely to illustrate how to add a mixin in our code; best practice in this instance would be to write the value in full, and not use a mixin. Using the Compass library, we can use Koala or the command line to add vendor prefixes automatically.

Next comes the `.overlay` rule; we've nested a number of styles within it:

```
.overlay {
  img { transition: all 0.2s linear; }
  .mask { opacity: 0; background-color: rgba( 75, 168, 175,
    0.9); transition: all 0.4s ease-in-out; }
  h2 { @include transform(translateY, -100px, 0);
    transition: all 0.2s ease-in-out; }
  p { @include transform(translateY, 100px, 0); transition:
    all 0.2s linear; }
  a.info { opacity: 0; transition: all 0.2s ease-in-out; }
}
```

The last part of this file contains the `:hover` pseudo selector. We can (and ideally should) nest it within the previous rule, but I've separated it for clarity, as shown in the following code:

```
.overlay:hover {
  img { @include transform (scale, 1.1,1.1) }
  .mask { opacity: 1; }
  h2, p, a.info { @include transform(translateY, 0px, 1); }
  p { transition-delay: 0.1s; }
  a.info { transition-delay: 0.2s; }
}
```

11. Save the file. For variety, we will use Koala to compile both files so that it can add the vendor prefixes automatically. Make sure you set the options, as shown in the following screenshot:
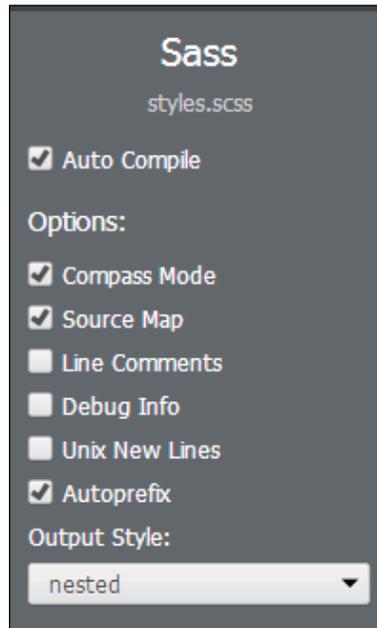
If all is well, we should get some compiled files; when we preview the results, we should see the images appear as shown at the start of this demo.

> If you want to check out a finished version, then extract a copy of the `sass site` folder in the code download that accompanies this book.

A nice easy demo to get us started, but anyone who knows me would know that we can do better than this; chuckle! Absolutely. Let's move on and take a look at another possible use for Sass; hands up those of you who use CSS grids to build sites?

They're an easy way to construct anything from a mock up through to a full-size site; let's dive in and take a look at how Sass can make our lives a whole lot easier when building grid sites.

Incorporating CSS grids

Hands up if you've spent any time constructing sites based on the principle of CSS grids?

The principle may be sound, but after a while, they can become a little tedious to build by hand, right? Sass mixins are a perfect way to help remove some of the tedium associated with grid layouts. It can make building sites a cinch. The great thing about Sass is that—as with any open source project—some kind souls have created a perfect mixin library to help with grids—enter Bourbon Neat. Let's take a look at why there is more to Bourbon than just a being good alcoholic tipple (sorry, couldn't resist!) when working with Sass.

Setting up Bourbon Neat

If you spend any time working with Sass, you will no doubt come across the Compass mixin library. It's a great library, but is often seen as being a little too complex for what it needs to be. Enter Bourbon—it is billed as a simple and lightweight mixin library for Sass and is available at `http://www.bourbon.io`. Bourbon is a great library to work with and is particularly adept at removing some of the bloat often associated with Compass.

> There are several good articles online that focuses on getting up to speed with using Bourbon. As a start, take a look at the blog post by Devin Clark at `http://devin-clark.com/getting-started-with-bourbon/`; there is also a cheat's guide to using Bourbon Neat available at `http://www.cheatography.com/claysmith/cheat-sheets/bourbon-neat/`.

Its sister library, Neat (`http://neat.bourbon.io/`) is designed to construct grids. It can help remove some of the Grunt work involved in building grid-based sites. I see a demo coming on, so enough with the chit chat; let's get down to business and perform the following steps:

1. Let's start with installing Neat. It requires Bourbon, so go ahead and fire up a command session and enter the following commands. Then, press *Enter* after each command:

   ```
   gem install neat
   gem install bourbon
   ```

2. In the Node.js command prompt, change the working folder to our project's `scss` folder, enter these commands, and press *Enter* after each command:

   bourbon install

   neat install

3. If all is well, we should see something akin to the following screenshot to verify that Bourbon and Neat are installed correctly:

At this stage, we're ready to start using Bourbon Neat. I feel a demo coming on, so let's get stuck in and set up some example grids using Bourbon Neat.

Creating grids with Bourbon Neat

For our next demo, we will take a look at one of the examples from Bourbon Neat's website: nesting columns. It's one of several ways we can use Bourbon Neat to construct grids; for more examples, take a look at the main examples page at `http://neat.bourbon.io/examples/`.

Let's make a start:

1. We'll begin by extracting the relevant files from the code download that accompanies this book. For this demo, we'll need copies of `bourbon.html`, `_typography.scss`, `_variables.scss`, and `bourbon.scss`. Save the markup at the root of our project folder, `bourbon.scss` in the `sass` folder, and the two partial files in the `partials` folder under `sass`.

2. The project folder will have two folders within it. One is called `bourbon` and the other is called `neat`. These aren't in the right place, so we need to move them into the `sass` folder; otherwise, our code won't compile correctly.

3. Next, fire up Command Prompt and change to the project's `sass` folder; we need to compile the `bourbon.scss` file to produce our style sheet. Enter this at the prompt and then press *Enter*:

   ```
   sass --watch bourbon.scss:../css/bourbon.css
   ```

4. At this point, we can preview the code. If all is well, we should see something akin to this screenshot:
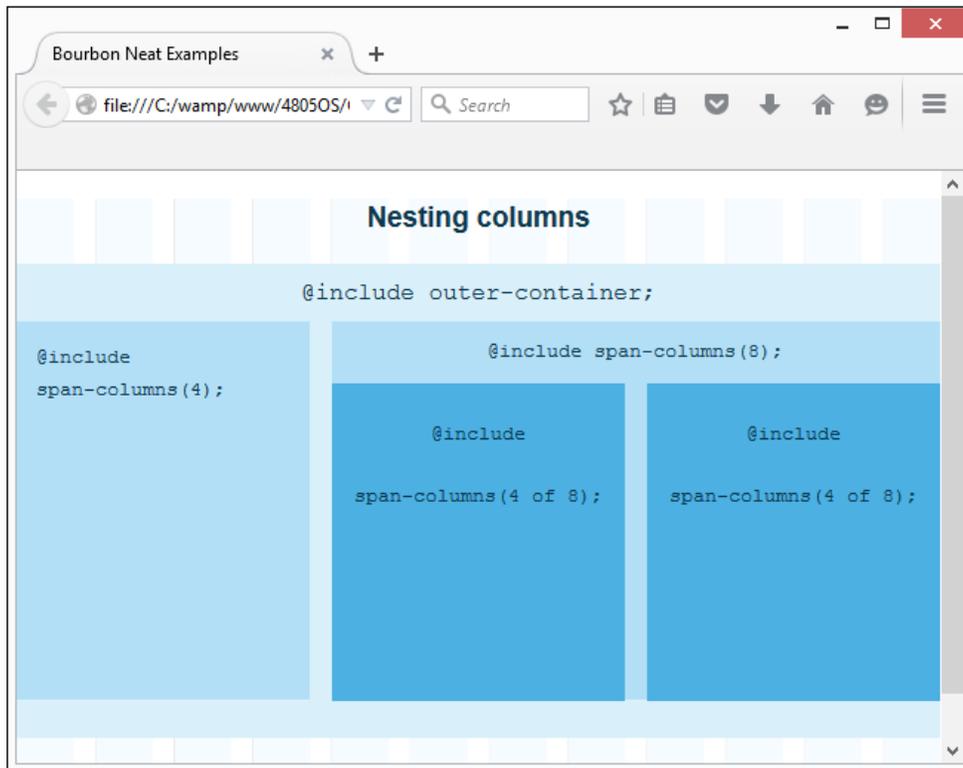
There are lots of things we can do with Bourbon Neat. To get a flavor, take a look at the main documentation available at `http://thoughtbot.github.io/neat-docs/latest/`.

In this demo, we've used Sass to compile the code manually from the command line with Sass' watch facility. It's worth noting though that if we already use a task runner such as Grunt elsewhere, we can always use it to take the manual `grunt` (pun intended!) out of compiling the code so that we can concentrate on building the grid. This has an added benefit that allows you to run additional tasks, such as autoprefixing or linting our code, at the same time as we compile the original Sass.

We covered how to configure Grunt back in *Chapter 1*, *Introducing Sass*. Will Schmierer, the developer, has put together a useful article that ties in using Bourbon with Grunt very nicely. It's available at `http://webdevstudios.com/2014/10/09/wordpress-up-and-running-with-grunt-sass-bourbon-and-neat/`.

> If you prefer using Gulp, then a similar workflow process can be designed using the equivalent Node package: `gulp-sass`.

Let's stay with the theme of frameworks and take it up a notch. I'll bet that there are a few individuals who haven't heard of Bootstrap, right? The great thing about Bootstrap is the way it can be used with preprocessors; it was originally built to use the Less preprocessor, but since then, it has been ported to use Sass. In the next few pages, we will' set up a small demo that uses the Bootstrap-Sass package; this takes out some of the hard work of adding Sass support and makes it easy to construct pages using Sass.

Applying Sass to frameworks, such as Bootstrap

For those of you not already familiar with it, Bootstrap is a powerful framework for end developers. We can use it to quickly produce responsive sites that work across different devices.

For a long time, if you wanted to use a version of Bootstrap in a preprocessor, then Less was the preprocessor language of choice. A recent port to Sass means that we can take advantage of the power of this library to help reduce the effort required to set up a Sass-powered site. We could do this manually, but a smarter choice is to use the Bootstrap-Sass package as a basis for our projects; this package is available at `https://github.com/twbs/bootstrap-sass`.

There are different ways to install Bootstrap-Sass; for the purposes of our demo, we'll begin with a quick look at how to use Bower to install it. We'll use this as a basis to set up automatic compilation with Grunt.

Implementing Bootstrap-Sass

The traditional way to set up a Sass-enabled Bootstrap site is to use the Sass port that is available at `http://getbootstrap.com/getting-started/#download`; there's nothing wrong with this method, but it is old hat!

Instead, we will use the power of Bower to provide the files for us. Let's dive in and see what is involved:

1. Bower uses Node, so let's begin with installing it. At Command Prompt, enter the following command and press Enter:

   `npm install –g grunt-cli`

2. Next, we need to install Bower, so at Command Prompt, enter the following command and press *Enter*:

   `npm install –g bower`

3. Now that Bower is installed, we need to create a `bower.json` file. This takes care of telling Bower what to install. In a new file, add the following code and save it as `bower.json` in our project area:

   ```
   {
     "name": "my-bootstrap-sass-project",
     "version": "0.1.0",
     "private": true,
     "dependencies": {
       "bootstrap-sass": "3.3.5"
     }
   }
   ```

4. Let's revert back to Command Prompt. At the prompt, change to the project folder, run this command, and press *Enter* after each command:

   `bower install`

5. If all is well, we should see Bootstrap-Sass installed, as shown in the following screenshot:

6. Then, we need to copy the contents of `bower_components/bootstrap-sass/assets/stylesheets` to the `sass` folder in our project area. Go ahead and create it if you do not have one present.

7. We will use the Awesome library font as part of this demo. So, go ahead and download it from `http://fortawesome.github.io/Font-Awesome/`, extract the contents of the `sass` folder, and copy it in a new folder called `font-awesome`, which sits in our `sass` folder.

8. In the same `sass` folder, go ahead and create a new file and save it as `style.css` in the `css` subfolder in our project area. Then, add the following styles to the file:

```
// Add custom font
@import url(http://fonts.googleapis.com/css?family=Raleway:
  400,700,300);
$font-family-base: 'Raleway', sans-serif;

// Import bootstrap and fontawesome
@import "bootstrap";
@import "font-awesome/font-awesome";

// Bootstrap Starter Template Styles
body {
  padding-top: 50px;
}
.starter-template {
```

```
    padding: 40px 15px;
    text-align: center;
}
```

We're almost there. Now, we need to add the automatic compilation process, which will use two packages: `grunt-contrib-sass` and `grunt-contrib-watch`.

9. In a new file, add the following code and save it as `package.json` in the root of our project area:

```
{
  "name": "my-bootstrap-sass-project",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "0.4.5",
    "grunt-contrib-sass": "0.8.1",
    "grunt-contrib-watch": "0.6.1"
  }
}
```

10. With the `package.json` file prepared, we now need to install the packages. At Command Prompt, go ahead and enter this command and then press *Enter*:

    npm install

11. The last step is to prepare a Grunt file; this tells Grunt how to handle the compilation process and which files should be compiled when watching for changes. In a new file, go ahead and add the following code and save it as `gruntfile.js`:

```
sassFiles = {
  'css/style.css': 'sass/style.scss'
}

module.exports = function(grunt) {
  grunt.initConfig({
    sass: {
      dev: {
        options: {
          style: 'expanded'
        },
        files: sassFiles
      },
      dist: {
        options: { style: 'compressed' },
        files: sassFiles
```

```
        }
      },
      watch: {
        sass: {
          files: 'sass/*.scss',
          tasks: ['sass:dev']
        }
      }
    });
  grunt.loadNpmTasks('grunt-contrib-sass');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.registerTask('buildcss', ['sass:dist']);
};
```

Phew! The Grunt packages are now installed; we can now compile code. This is as simple as changing to the project folder at the Node.js command prompt. Now, run this command:

grunt buildcss

If all is well, it should not report any errors, as shown in this screenshot:

A quick check in the `css` folder of our project area should reveal two files: the compiled CSS style sheet along with a sourcemap:
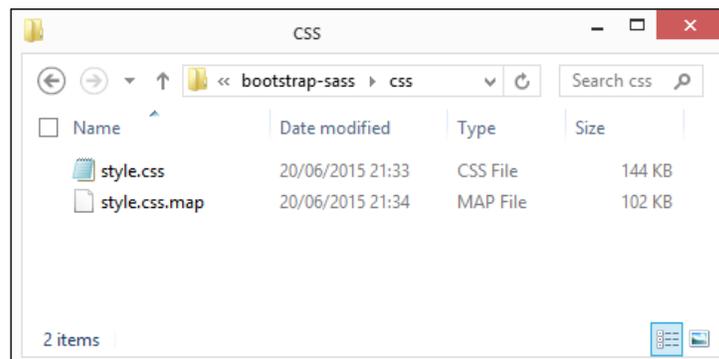
Hold on! Why do we have a sourcemap file present when neither the Grunt file nor `package.json` has this enabled? By default, it's enabled from within Sass itself, so there is no need to configure it separately.

To prove that our compiled code works, we can preview the results of a simple demo that I've included in the code download that accompanies this book. For this, you will need to extract a copy of `index.html` from the code download and then run it from the root of the project area. If all is well, we should see the following screenshot:
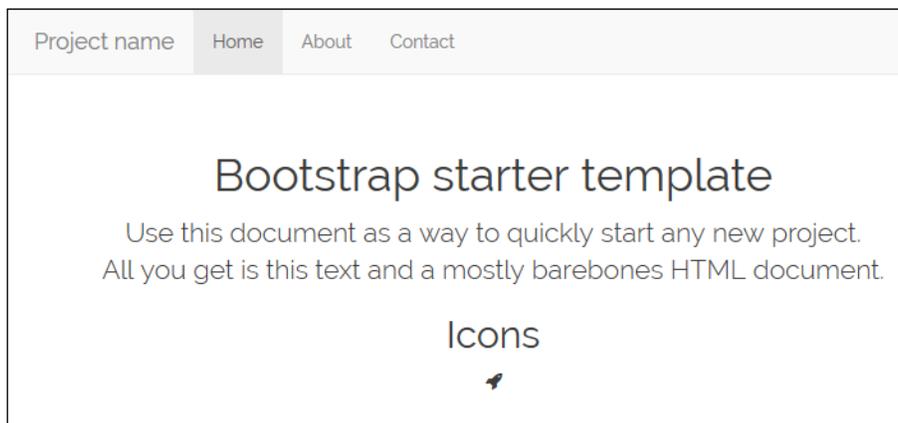
It's a simple example, which hides some of the power of using Sass with Bootstrap. It is well worth taking the time to explore some of the mixins in use with this version of Bootstrap because they may well provide a useful source for your own future projects.

Exploring what happened

If we take the time to explore through some of the files used in this demo, you can be forgiven for thinking, "whoa! There's a lot there, what do they all do?"

Well, most form part of the standard styling for Bootstrap when Sass is used; the key file we need to concentrate on is `style.scss`, which I have reproduced in full, as shown in the following code:

```
// Add custom font
@import url(http://fonts.googleapis.com/css?
  family=Raleway:400,700,300);
$font-family-base: 'Raleway', sans-serif;

// Import bootstrap and fontawesome
@import "bootstrap";
```

```
@import "font-awesome/font-awesome";

// Bootstrap Starter Template Styles
body {
  padding-top: 50px;
}
.starter-template {
  padding: 40px 15px;
  text-align: center;
}
```

The key point to remember when using Bootstrap-Sass is to define any variables **before** importing the main Bootstrap mixin file; otherwise, they won't kick in! We can then override any style afterwards, such as body, as we have done in our demo.

The beauty of using Bootstrap-Sass is that we can tie it in to a task runner such as Grunt; we made good use of the `grunt-contrib-sass` package to compile our code. In our case, we compiled it manually to get a feel of the process; we can easily use Grunt watch to get Grunt to automatically compile the code for us.

Okay, let's move on. Thousands of sites worldwide, including well-known names (such as Mercedes, the fashion magazine Vogue, and New York Post) use WordPress. In recent years, the popular CMS system moved to using Sass as a basis to style the application; let's take a look at some of the tips and tricks we can apply to any theme built for the system.

Using a CMS system

A question—how many of you use a CMS system? To put it another way, how many of you use WordPress?

I'll bet that a fair few of you use WordPress in some guise—after all, it is arguably one of the most well-known and used content management systems worldwide.

If you're relatively new to using WordPress though, you've probably taken a look at the style sheet and must be wondering that how can I convert *that* to Sass? Well, it's not difficult at all because we've covered all the techniques we're going to use already; it's just a matter of planning. Also, don't try to do it all at once, chuckle!

Before we get stuck in to editing code, it's worth spending a little time getting to know something about the history of Sass and WordPress. WordPress Core formally adopted the use of Sass back in 2013. It was picked over other preprocessors for its compatibility with licensing (it's compatible with GPL) and support for more advanced logic as well as the typical reasons to use a preprocessor, such as compiling multiple style sheets into one CSS file.

> If you want to get into the detail of why Sass was chosen, then head over to `https://core.trac.wordpress.org/ticket/22862` for the full discussion.

Okay, enough history. Now, let's get down to coding!

To give you a flavor of how Sass has been used in WordPress, take a look at one of the mixin files from Core at `https://core.trac.wordpress.org/browser/trunk/src/wp-admin/css/colors/_mixins.scss?rev=29648` and see how it uses variables, functions, and the ampersand placeholders.

The great thing about how WordPress uses Sass is that we can easily use the same tools that they have used; the team use UglifyJS and clean-css (via Grunt) to help with their development process. There is nothing stopping us from using these tools ourselves, but for the meantime, let's take a look at the different ways we can get Sass into our theme and look at how to install prebuilt themes for WordPress that use Sass.

Choosing a Sass theme for WordPress

Choices, choices, huh! Now, where have I heard that before, I wonder…?

One of the great things about WordPress, and ironically what makes it harder, is the number of choices that you have. There are hundreds of themes available for download; we can pick one and install it, customize it to meet our requirements, or even build one from scratch.

In the next few pages, we'll take a look at how to customize an existing theme, but for now, let's take a look at some of the options available to help you get started. All the themes can be installed using the normal process for WordPress, either as a main theme or a child of an existing one. Here are some of the options:

- **Forge**: This is available at `http://forge.thethemefoundry.com/`
- **Underscores (or _s)**: This can be downloaded from `http://underscores.me/`; click on **Advanced Options | _sassify!** to download a Sass version of the theme

- **Some Like It Neat**: This is based on the minimalistic underscores theme and can be downloaded from `https://wordpress.org/themes/some-like-it-neat`
- **Bones**: This is a great starter theme that includes support for Sass; it can be downloaded from `http://themble.com/bones/`
- **The WordPress Starter Theme**: This starter theme for WordPress was designed by Matt Banks, which uses Sass among others; his theme is available at `https://github.com/mattbanks/WordPress-Starter-Theme`
- **JointsWP**: This is a cross-over between Sass and Foundation; the theme is available at `http://www.jointswp.com`

It is worth checking online. There are thousands of themes available; some will have Sass support built-in. The great thing about WordPress is that we are not limited in what we do. If we want to change the theme, we can either edit it or swap it for a completely different one.

A small tip though is that if you want to edit an existing theme, then it is good practice to try to build or adapt a child theme; if changes are made to the base theme, then we only need to override those styles that need to be changed, rather than build everything from scratch.

> For a detailed discussion on the merits of using child themes, head over to `https://thethemefoundry.com/blog/wordpress-child-theme/`; the article by Drew Strojny makes for interesting reading.

There may be instances where this won't be possible if we override most of the styles. In this instance, we will need to edit the main theme directly, which will be the subject of our next demo.

Adapting a WordPress theme to use Sass

Okay, so you've searched everywhere for a suitable Sass-based theme, but can't quite find one that works for you. Therefore, you want to try adapting an existing one to suit your needs.

Sound familiar? If there was a theme available that suited everyone's needs, then the author would make nothing short of a small fortune. This is unlikely to happen; a more realistic prospect is that we need to adapt an existing theme to match our needs.

We can edit the style sheet directly using any text editor for the purpose of compiling it; we have two options. One option is that we can compile manually (either from the command line or using a GUI compiler). Another option is that we can compile using a plugin directly in WordPress.

Let's delve in and take a look at the latter option first to see what is involved.

Using a plugin to compile Sass in WordPress

In this first demo, we 'will use the WP-SCSS plugin, which is available at `https://wordpress.org/plugins/wp-scss/`; once installed, we will use this to compile the code directly in WordPress.

> The WordPress plugin download page shows this plugin as being compatible up to version 3.8.8 of WordPress; I've seen no issues with running it in version 4.2.2 of WordPress.

For the purposes of this demo and the next, you will need a working copy of WordPress installed; ideally this will be a local version on a web server, such as WAMP or Apache. Alternatively, any online space will work equally well. We will assume that you have the former installed with the Twenty Fifteen theme activated and working under `c:\wamp\www\wordpress`. If the location is different, then adjust the code accordingly.

> If you are unsure about how to install WordPress locally, then head over to `https://wordpress.org/download/` for full details.

Assuming that we're all set, let's make a start. Perform the following steps:

1. We first need to set up some folders in our theme's folder. For this, browse to the theme folder in your WordPress installation and create a folder called `library`. In this folder, go ahead and create two subfolders (`scss` and `css`); we also need another folder called `partials`; the end result should look like this:
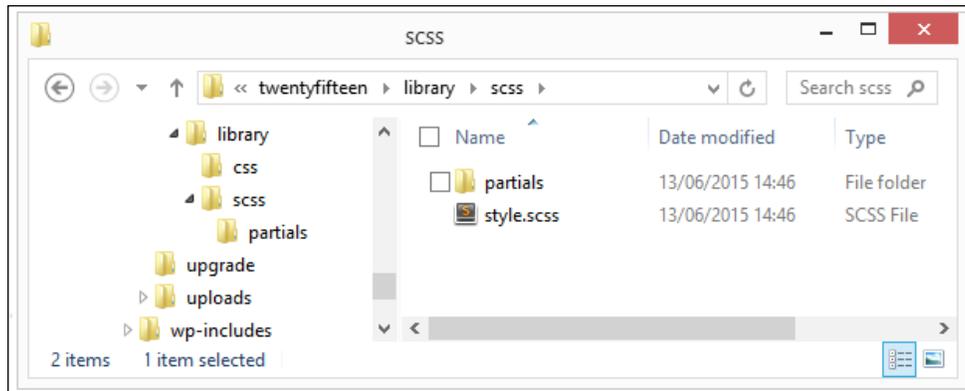
2. You will notice the presence of `style.scss` in the screenshot for the previous step. Now, move the styles.css file from the root of the theme folder and rename it as `styles.scss`.

3. We now need to download the plugin, so navigate to `https://wordpress.org/plugins/wp-scss/` and click on the **Download Version…** button.

4. Next, go ahead and install the plugin using the standard process to install and activate plugins in WordPress. (If you are unsure of the process, refer to `https://codex.wordpress.org/Managing_Plugins` for full details).

5. Once installed, click on **Settings** | **WP-SCSS** in your WordPress' administration area; it will display the configuration page for the plugin along with this message:
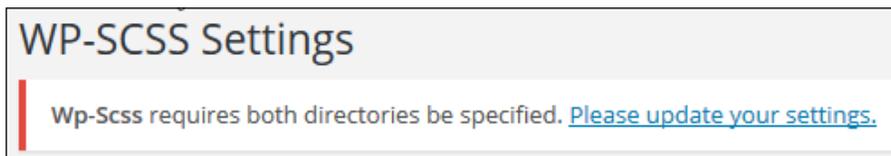
6. In the **Configure Paths** section on this page, add the following code to the **Scss Location** field:

    `/library/scss/`

7. Next, add this code to the **CSS Location** field:

    `/library/css/`

8. Leave the other fields as default. Then, click on **Save Changes**. Now, WordPress will display a message to verify that the settings have been saved.

At this point, WordPress is now set up and ready to make changes. To prove that it works, let's start by making sure that we have a valid compiled file in the css folder. Navigate to the front page of your WordPress installation; this may take a moment or two, but eventually, we should see a new CSS file in the css folder:
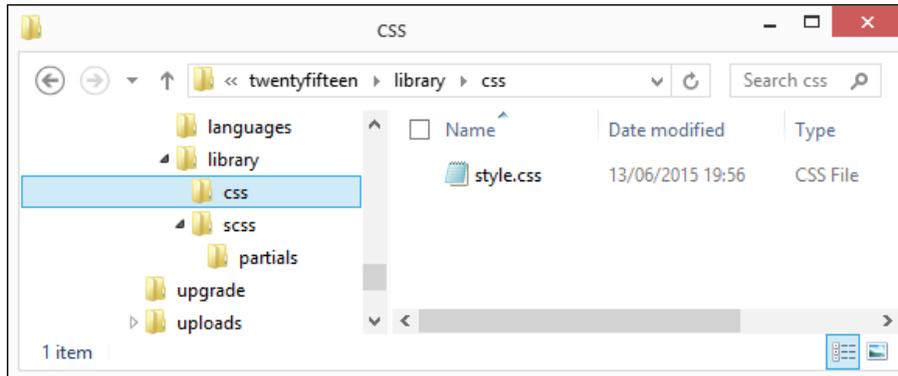
Assuming that a new file does appear, we're now good to go with editing the style.scss file; we will cover this in more detail in the next two exercises.

Let's move on. Using a plugin to compile our code works perfectly well, but it has a couple of drawbacks. You may have noticed an option to enqueue the compiled style sheet; if we enable this, it will most likely break our installation of WordPress if we were to remove it at a later date (the enqueue link would point to the wrong area). We can get around this by only using it in a development or testing capacity and then use the final version to replace the original style.css file that normally sits at the root of our theme folder.

The second issue though is that the current release of this plugin supports full compilation of Sass, but limited compilation for Compass and Bourbon. This makes the plugin a suitable option for you if your theme requires either library. Instead, we will have to compile our style sheet outside of WordPress, which just happens to be the subject of our next demo, so let's take a look at what is involved.

> If you want to get really adventurous, you may like to try updating the plugin to a newer version of scssphp, which is available at https://github.com/leafo/scssphp.

Compiling code outside of WordPress

If using a plugin to compile code directly in WordPress isn't an option, we can resort to using an alternative. In this instance, we will use Compass to compile from the command line. This allows the use of the `config.rb` file to specify exactly how files should be compiled.

> This demo assumes the use of a vanilla copy of the Twenty Fifteen theme, so I would recommend renaming the theme you used from the previous demo and dropping in a new copy from the WordPress download archive at `https://wordpress.org/latest.zip`.

Assuming that we have something installed, let's make a start:

1. We first need to install Compass, so go ahead and bring up Command Prompt, change to the root of our project area, enter this command and then press Enter:

   ```
   $ gem install compass
   ```

   > You may need to update your current Ruby gems; run the following command to do so:
   > ```
   > $ gem update –system
   > ```

2. Next, we need to create the `config.rb` file; we can do this at the command line with this command:

   ```
   compass init
   ```

3. Once created, edit the file so that it matches the code, as shown here:

   ```
   http_path = "/"
   css_dir = ".."
   sass_dir = "sass"
   images_dir = "images"
   javascripts_dir = "js"

   environment = :development
   output_style = (environment == :production) ? :compressed :
     :expanded
   ```

4. Save the file in the root of the theme folder.

5. In the theme folder, create a new folder called `sass`; in this folder, create another folder called `_partials`. We also need to move `styles.css` into this folder and rename it as `style.scss`; if all is well, we should have this:
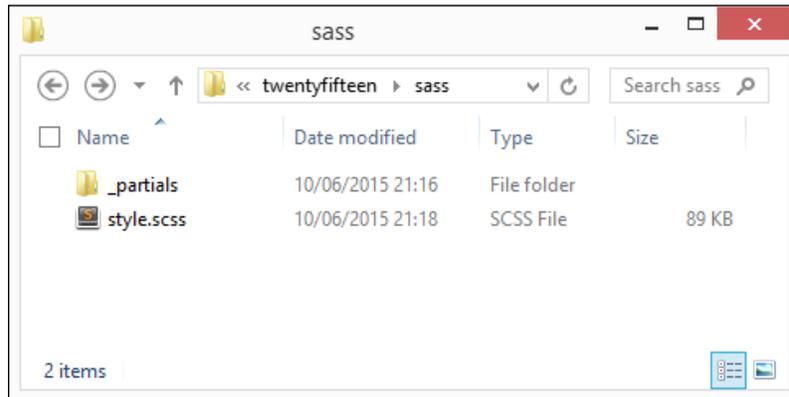
6. In the same command line (or terminal) session, change to the theme folder, which will be at `C:\wamp\www\wordpress\wp-content\themes\twentyfifteen`.

7. Enter the following code at the prompt and press *Enter*:

```
compass watch –sourcemap
```

At this stage, we're ready to compile code. If we make a change to our `style.scss` file, Compass will kick in automatically and compile the file into valid CSS.

Let's put this to the test. Open up `style.scss`. Perform the following steps:

1. At or around line 1161 is the start of section 10 or `header`. Take a copy of lines 1165 to 1195; the code should start and end as follows:

```
.site-header {
  background-color: #fff;
...
  opacity: 0.7;
}
```

2. Paste these in a new file, saving it as `_header.scss` in the `_partials` folder of our theme.

3. Remove the block of code we've just copied and replace it with this line:

```
@import "_partials/header.scss";
```

4. Save the file; if all is well, Sass will kick in and compile our file to produce a valid CSS style sheet, as shown in the following screenshot:
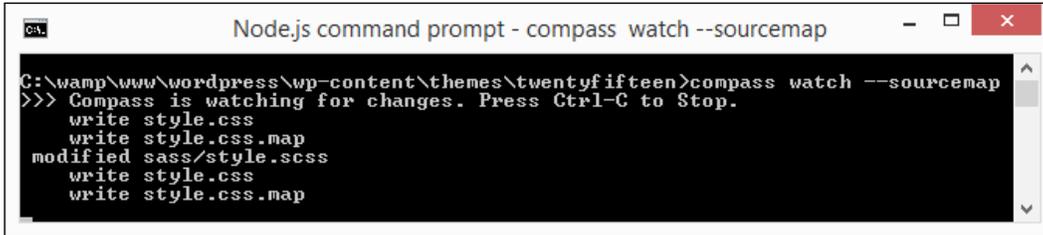
```
Node.js command prompt - compass watch --sourcemap

C:\wamp\www\wordpress\wp-content\themes\twentyfifteen>compass watch --sourcemap
>>> Compass is watching for changes. Press Ctrl-C to Stop.
    write style.css
    write style.css.map
modified sass/style.scss
    write style.css
    write style.css.map
```

5. To prove this, perform the same process with section 8 ("Alignments") and compile the code; if all is well, we should see something akin to this when you preview the compiled style sheet:

```
1340    * 8.0 Alignments CONVERTED TO SASS
1341    */
1342    /* line 1, sass/_partials/_alignments.scss */
1343    .alignleft {
1344      display: inline;
1345      float: left;
1346    }
1347
1348    /* line 6, sass/_partials/_alignments.scss */
1349    .alignright {
1350      display: inline;
1351      float: right;
1352    }
1353
1354    /* line 11, sass/_partials/_alignments.scss */
1355    .aligncenter {
1356      display: block;
1357      margin-right: auto;
1358      margin-left: auto;
1359    }
1360
```

> If you have difficulty with getting `style.css` to compile in the right folder, then take a look at the end of this post at `https://css-tricks.com/compass-compiling-and-wordpress-themes/`; there is a great tip on how to use Ruby to move the file automatically that can be added to the `config.rb` file if needed.

We're now ready to adapt our theme; we have Sass and Compass installed, the folder structure is set up, and we've proven that Sass is correctly compiling some simple `@import` requests.

What next…? Easy! Let's now look at the real meat and potatoes of this process or how to adapt our existing code to use Sass. There are lots of changes we can make; let's explore a few as part of the next demo.

Making changes to our WordPress theme

If you take a look at any style sheet, you may at first think there is some magical formula to convert it. You may also think that there must be a set of rules or principles we must follow.

The truth is that there is no secret formula; the key to it is planning and to not do it all in one go. There are a number of tricks we can use to start introducing Sass, so let's take a look at a few now. These will use the Twenty Fifteen theme as our example, but can equally apply to any theme:

- A quick win is to work on CSS3 vendor prefixes; a good example in the theme is this block of code:

```
html {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  font-size: 62.5%;
  overflow-y: scroll;
  -webkit-text-size-adjust: 100%;
  -ms-text-size-adjust: 100%;
}
```

If we're using a GUI application, such as Koala or the Compass library (via the command line), we can get it to automatically add vendor prefixes; the two highlighted lines can be removed because they will be added at compilation. We can even use a task runner and package, such as Dmitry Nikitendo's Grunt-autocomplete, with Node and Grunt to add these as well.

- Instead of setting color values individually, we can build these in a variable list, assign human-readable names to each color value, and assign these as required in code. We covered this in detail back in *Chapter 3, Building Functions, Operations, and Nested Styles*.

- Have a look at line 188, which sets this attribute:

  ```
  font-size: 16px;
  ```

  We can use a mixin to convert this to display rem values by default; these work better in modern browsers. Also, the mixin can include pixel values as a fall back if rem values are not supported. Note that `font-family: "Noto Sans", sans-serif;` is used throughout the code; we should look to include this in the mixin as well.

- Have a good look throughout the entire style sheet; there are plenty of opportunities where we can nest styles to make it easier to read. A good example is in section 3.0 (typography), which can easily be converted to nested styling. Another good example is from line 1413 onward. In the `.widget-archive` rule, there is a good opportunity to reorganize the code to use nesting.

- A good principle to use is the SMACSS approach to separate code into smaller sections; the idea here is to split it on a section-by-section basis, that is, 1.0 Reset to `_reset.scss`, 2.0 Genericons to `_genericons.scss`, and so on.

> The SMACSS principle is covered in detail at `https://smacss.com/`. A good basis to use for this is the 7-1 Pattern by Hugo Giraudel. To see it in action, take a look at the article by David Khourshid at `https://scotch.io/tutorials/aesthetic-sass-1-architecture-and-style-organization`.

- In section 17 (Print), we can use Breakpoint-Sass to create media queries; these can easily be put in a separate file using the SMACSS principle.

- At line 2507 onward, we can use the `@for` statement to build the `gallery-columns-2` to `9` statements.

- In lines 1897 to 1957, we have a number of very similar statements that use `a:before` pseudo selector; we can look to create a mixin with the string interpolation.

- In lines 1663 to 1680 and 1694 to 1717, there are good opportunities here to use the `@for` statement to construct the statements automatically.

- Once you've made some of the simpler changes, don't forget to look at how to use existing mixin libraries (such as Bourbon or Compass). A perfect example is the font mixin; we can create our own, but sometimes, it is less work to use a mixin created by someone else if they've already done the work for us.

- If we want to use a CSS normalization style sheet in our themes, then take a look at `Normalize.css`; there are **dozens** of versions available. The beauty is that a lot of individuals have created or forked versions of `normalize.css` or converted it for use in Sass. We can even use Node to install a version; take a look on the Internet for a version that suits your needs.

Ultimately, the nature of any changes we make will vary from style sheet to style sheet; the key to it is using an iterative process. There may be occasions where making improvements to code would mean that we have to alter existing code first before making the most important change.

> To get a feel of how to create a suitable WordPress / Sass workflow, take a look at the article by Andy Leverenz at `http://www.elegantthemes.com/blog/tips-tricks/how-to-use-sass-with-wordpress-a-step-by-step-guide`. Alternatively, Eric Barnes has an extensive article on how to use Gulp with Sass and WordPress. It is available at `https://github.com/synapticism/wordpress-gulp-bower-sass`.

A good example is to create a mixin that handles font styling; we may set some rules with pixel values, some with rem units, and a mix of rules that have similar font attributes, but not all the same. We would first need to ensure that our font sizes have both pixel and rem units set. We also need to ensure that any instance of font styling has a consistent set of attributes set (for example, font weight, name, or color). Once these are all set, we can then construct our mixin to automatically handle the creation of each style.

Alright. Now, let's change tack. Before we round out our journey through the essentials of Sass, there is one small point we need to cover: the use of comments. Hold on though! Didn't we cover that earlier? We did. There are some quirks though when it comes to working with WordPress, so let's cover these now.

Adding comments to our code

If you've spent any time developing WordPress themes, you will of course know that WordPress requires a particular format of comments at the head of any theme style sheet. This allows you to display details of the theme in the dashboard.

In this instance, any single line comments added using `//` will be removed at the compilation stage. WordPress uses a multiline comment. This will be preserved when possible; sometimes, it will be removed even if compilation is set to render output in the `:compressed` mode. To get around this, we need to add an exclamation mark to preserve the comments, as shown in the following code:

```
/*!
Theme Name: Sassy Theme
Theme URI: http://jamessteinbach.com/themes/sassy/
Author: James Steinbach
Author URI: http://jamessteinbach.com
Description: From CSS to Sass Sample Theme Code
*/

//import all your partials
```

Okay, let's move on. We've almost come to the end of our journey through the essentials of Sass, but before we round out the chapter, now is a good moment to take a look at some other uses of Sass.

Other projects using Sass

There is more to Sass than simply creating static websites; throughout the chapter, we've scratched the surface on a handful of use cases, where we can incorporate the power of Sass into existing projects.

There are plenty of opportunities to use Sass in a variety of different instances; before we round out our journey on the essentials of Sass, let's take a look at a couple of examples of where you can use Sass:

- The classic is with Rails; the Merb plugin was available, but functionality has been merged in the latest version of Rails to prevent duplication. For more details on the plugin, take a look at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#rackrailsmerb_plugin` or head over to `https://github.com/rails/sass-rails` for an alternative plugin that we can use to add Sass support to Rails.

- For fans of e-commerce, Sass isn't an issue either; the WooCommerce platform is a perfect place to try out using the library. Head over to `http://www.creare.co.uk/woocommerce-and-sass` for a tutorial on how to get started.

- Anyone using AngularJS? No problem. Sass cam be used in Angular projects; Jake Marsh has a useful project on GitHub to help you get started at `https://github.com/jakemmarsh/angularjs-gulp-browserify-boilerplate`.

There will be plenty of examples available online, where others have built projects using Sass or have experience of using it with particular systems as we have done throughout this chapter. It's worth taking the time to look; after all, why reinvent the wheel if someone else has already done the work for you?

Summary

Throughout this chapter, we've covered a number of tips and tricks that can be applied to any site, from something as simple as a one-page application to a 2000+ line behemoth of a style sheet that we may find in use in WordPress. Let's take a few minutes to recap what we've covered.

We started with a quick recap on how to install Sass and Compass, of which the former is obligatory, but the latter is optional. We then looked at a number of tips and tricks to help with the initial few steps in converting to use Sass. We used these to work through converting a simple demo from using standard CSS to the Sass equivalent code.

Then, we looked at that classic styling piece in the form of CSS grids. We saw how the popular Bourbon Neat library can be used to make building such grids a cinch. Our next demo came in the form of applying Sass to existing frameworks. Also, we used Bootstrap as the basis for this demo with the Bootstrap-Sass package providing the styles.

Our last demo looked at how to use WordPress with Sass. We began with taking a look at some of the themes that use Sass by default before adapting WordPress to using Sass through the use of a plugin or manually. We then looked at some of the changes we can make to incorporate Sass into our themes. We concentrated on how to use the Twenty Fifteen theme, but the tricks can be applied to any theme as appropriate. We then rounded out our journey through WordPress with a quick look at how comments have special significance when compiling Sass.

Our journey through the essentials of Sass is almost complete. We finished the chapter with a quick look at some of the other projects or application that use Sass. We also looked at where we can apply similar principles that we have covered throughout the book. I hope that what we have covered will inspire you to start using Sass, and you have enjoyed our travels through the world of Sass.

Index

Symbols

@each directive
  exploring 119
@extend directive
  about 101, 102
  benefits 104
  URL 107
  used, for creating dialog boxes 104-107
@for directive
  using 118
@if directive
  using 116, 117
@import command
  URL 53
@media queries
  URL 109
  used, for making site responsive 107-109

A

animations
  building, Sass used 122-125
  key 126

B

bones
  URL 147
Bootstrap-Sass
  implementing 140-144
Bourbon Neat
  grids, creating with 137-139

  setting up 136, 137
  URL 136
Breakpoint mixin
  installing, for Sass 110
  URL 109

C

child themes
  URL 147
CMS system
  using 145, 146
code
  comments, adding 157
  linting code, using 32
  outcome, controlling 116
  writing 32-35
CodeKit
  URL 10
CodePen
  example, URL 104
  URL 79
colors
  changing, URL 67
  creating, with functions 66-68
  documentation, URL 70
  in use colors, changing 69, 70
  mixing 71, 72
  selecting, URL 71
COLOURlovers
  URL 80

source maps
  URL 20
  using 19-22
SublimeLinter SCSS plugin
  URL 32
Sublime Text
  Sass support, adding to 6-8
  URL 6

U

Underscores
  URL 146

V

variables
  about 28, 37
  code key points, exploring 38, 39
  creating 37
  naming, URL 40
  URL 38
  used, for creating colors 39-42

W

watcher
  URL 17
web font generator
  URL 60
WooCommerce
  URL 157
WordPress
  code, compiling outside 151-154
  download archive, URL 151
  plugin, used for compiling Sass 148-150
  Sass theme, selecting 146, 147
  Starter Theme, URL 147
WordPress theme
  adapting, to use Sass 147
  changes, making to 154-156

Thank you for buying
Sass Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
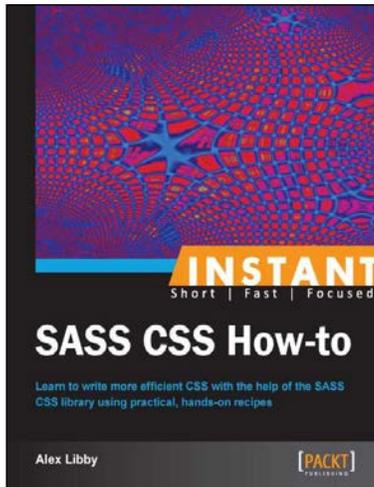
Instant SASS CSS How-to

ISBN: 978-1-78216-378-7   Paperback: 80 pages

Learn to write more efficient CSS with the help of the SASS CSS library using practical, hands-on recipes

1. Learn something new in an Instant!
    A short, fast, focused guide delivering immediate results.

2. Learn how to download and install SASS and compile SASS code to validate CSS.

3. Learn how to use the various elements of SASS, such as mixins, variables, control directives, and functions to create valid CSS.
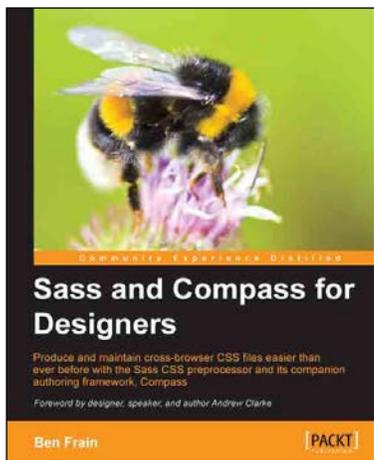
Sass and Compass for Designers

ISBN: 978-1-84969-454-4   Paperback: 274 pages

Produce and maintain cross-browser CSS files easier than ever before with the Sass CSS preprocessor and its companion authoring framework, Compass

1. Simple, clear, and thorough. This book ensures you don't need to be a programming mastermind to wield the power of Sass and Compass!

2. Previously tricky and time-consuming CSS tasks will become trivial. Easily produce cross-browser CSS3 gradients, shadows, and transformations along with image sprites, data URIs, and more.

Please check **www.PacktPub.com** for information on our titles
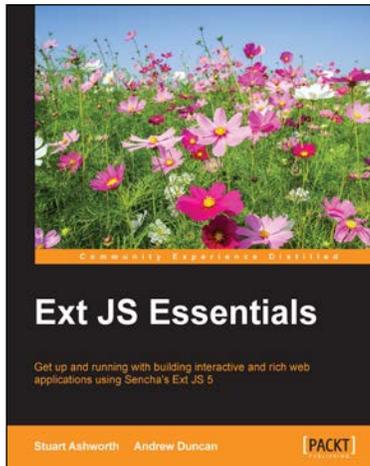
Ext JS Essentials

ISBN: 978-1-78439-662-6   Paperback: 216 pages

Get up and running with building interactive and rich web applications using Sencha's Ext JS 5

1. Learn the Ext JS framework for developing rich web applications.

2. Understand how the framework works under the hood.

3. Explore the main tools and widgets of the framework for use in your own applications.
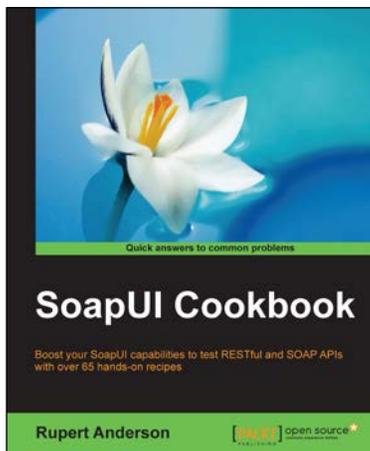
SoapUI Cookbook

ISBN: 978-1-78439-421-9   Paperback: 328 pages

Boost your SoapUI capabilities to test RESTful and SOAP APIs with over 65 hands-on recipes

1. Quickly gain simple-to-use building blocks to power up your SoapUI toolkit.

2. Use Groovy scripting and open source technologies to add the SoapUI functionality you need to successfully test RESTful and SOAP APIs.

3. Access reusable step-by-step technical solutions to common and more advanced real-world test scenarios.

Please check **www.PacktPub.com** for information on our titles