GitBook

Sass in the Real World

Book I of IV

Kianosh Pourian & Dale Sande

Table of Contents

iv. The Turducken mixin

Sass in the Real World: Book I of IV

In our lives, whenever we encounter a great experience whether it be a great movie or a great vacation location or a great technology toolset, our desire is to share it with everyone we encounter. That is how we feel about Sass CSS and that is why we decided to write a book about it. There have been plenty of introductory books on Sass, however we felt that our field was missing a more advanced book on Sass, a book that goes beyond the basics of Sass and CSS and lays the proper groundwork for starting a CSS project with Sass.

We are very excited to take this journey and hope that you will join us and share this experience with us.

-- Kianosh Pourian and Dale Sande

About the book and series

Our initial goal of writing this book was to fill a void in Sass CSS books which is a book that covered beyond the basics of Sass CSS development. To fulfill this goal, our initial approach is to provide a single book that covered the A to Z of professional Sass CSS development. While this goal has not changed, our approach has made a small "pivot" (word du jour in today's technology world).

We have divided our single book approach into a 4 part book series. This was done to achieve several goals:

- Be able to publish a book quicker and bringing it into the market faster
- Allow users to select the desired part of the series without having to purchase the entire series. This was a very important part of our approach, as we have seen that different developers are at different levels of Sass CSS learning and development. This will give all a chance to fill in the gaps as needed.

The four part series consist of the following parts:

- *Part 1: Getting Started with Sass*. This part of the series concentrates on the basics of Sass development however with a deeper context and history behind all that is Sass. Our goal for this part of the series was to not only review the basics but also present an explanation behind all the decisions that was made and decisions that a developer must consider and make when developing with Sass
- *Part 2: Deeper Dive*. A continuation of our philosophy on not only understanding why certain structure has been built but also the deeper underlying structure behind it. In this part of the series, we have taken a deper dive into functions (both out-of-the-box functions and custom functions), when to use functions vs. mixins and other tools and processes that accompany Sass CSS development.
- *Part 3: Getting Really Sass'y*. In this part of the series, we continue with all the development needs by talking about issues like responsive design, testing, debugging, and working with frameworks like Zurb Foundation or Twitter Bootstrap. We will also talk a bit about Compass, the Sass framework that can be accompnied with Sass CSS development.
- *Part 4: Sass in the Stack*. We end the book series with the implementation of Sass in different technology stacks like NodeJS implementation or the Rails asset pipeline. We will also touch upon some performance issues and how to handle these issues.

We hope you enjoy these books, either through individual series or the entire four part series, and this will help you further in your Sass CSS development.

Assumptions

This is the part that most book will label as **"Who is this book for?"** but since we are very strict in our grammar and refuse

to end a sentence in a preposition, we have called it "Assumptions", but the sentiment is the same. These are some of the assumptions that we are making:

- We assume that the reader is familiar with Sass CSS and has install Sass on their development machine. If you have not installed Sass CSS, it is very easy, go to Sass-lang.com.
- We are using a command line interface to run all of our Sass development. However if you want to use some applications like Codekit, Compass.app, Scout, or any other development application, please feel free.
- Although we are very opinionated about some of our approaches in Sass CSS development, we are agnostic to the technology stack that is being used and promote the usage of Sass in any environment that is suitable and will meet your needs.

So let's start learning about Sass CSS and develop CSS with its' rightful accompanying tool.

A little under the hood

Sass: Syntactically Awesome Style Sheets. A stylesheet language initially designed by Hampton Catlin and developed by Nathan Weizenbaum and Chris Eppstein. Sass clearly is a pioneer in it's field. Despite it's many competitors, Sass continues to pave the way not only for preprocessors, but continues to gain support with the W3C standards group and influential browser manufactures.

One of Sass' many strengths we will explore is it's support of two distinctly different syntaxes. A feature that has lead to confusion amongst newcomers and controversy with the more seasoned developers. Be it the the original Sass .sass syntax or the newer SCSS .scss syntax, they are equally committed to by the Sass core team and have no functional differences. In this chapter we will go into where Sass came from, why the SCSS syntax was created and what the differences really are.

Sass' next learning curve typically falls into structural and architectural disciplines. It is not that developers don't know how to write code, but more to the point of writing well structured, readable, manageable and scalable code. The one who wrote the code is not always the one who maintains the code. Remember the campfire rule, "*You must always leave the code as good or in a better state than you found it.*"

From indentation syntax (whitespace) Sass to Sassy CSS (SCSS)

In May 2006 Hampton Catlin introduced us to HAML. A lightweight markup preprocessor language that uses indentation (whitespace) to separate blocks. HAML replaces the more common HTML syntax `<tag> ... </tag>` with a simpler, more lightweight syntax `%tag`. HAML's true strength lies within it's use of indentation to determine selector nesting and block separation.

In this section I will discuss the differences between HTML and HAML and how these ideas inspired Sass.

From HTML to HAML

HTML relies on open and closing tags `<tag> ... </tag>` to separate code blocks. Placement of HTML selectors within a pair of tags designates nesting.

In the following example you will see that all the content for this block is nested within the `<article> ... </article>` tags. The nested `<p> ... </p>` tags contain the nested `<a> ... </a>` tags stating that the `<a>` inline-block element will be inside the `<p>` block element.

```
<article>
  <h1>primary header title for paragraph below</h1>
  <p>
    <a href="#">Example of body text link</a>
  </p>
</article>
```

It's interesting to note, that while returns and tabbing in HTML is a good form of writing clean code and does make the code easier to read, it is not necessary for nesting or block separation. The clear advantage here is with HTML minification of course.

It isn't until we begin to append styles, actions and other attributes to the markup, that reading becomes increasingly difficult. Notice in the following example by simply adding a few additional attributes to the HTML that it becomes more difficult to visually parse out the content from the functionality.

```
<article class="primary-article" id="main-author" data-author-id="999">
  <h1>primary header title for paragraph below</h1>
  <p>
    <a href="#" class="external-link">Example of body text link</a>
  </p>
</article>
```

The worst part is, this is still static HTML. Things only get worse as we begin to add logic and insert dynamic content.

Hampton invents HAML

In the previous section we saw when traditional HTML increases with complexity, it decreases in readability. It was from this perspective that Hampton came up with the four principals to good programming with HAML.

- Markup should be beautiful
- Markup should be DRY (Don't Repeat Yourself)
- Markup should be well-indented
- XHTML structure should be clear

Where HTML requires open and closing tags, HAML does not. Where HTML does not require indentation, HAML enforces it. The beauty of HAML is in it's lack of repetition, aka Do Not Repeat Yourself (DRY). HAML removes the angle brackets (chevrons) `< ... >` and replaces them with a single `%` symbol. To reduce repetition, HAML also removes of the need for a closing tag. Notice in the following example that only an opening tag with a `%` symbol is required to open a new HTML block.

```
%article
  ... content ...
```

Earlier I stated that HTML requires nested tags to be *physically nested* within each other in the markup. Following the principals of *Markup should be well-indented* and *XHTML structure should be clear*, HAML takes a more specific approach. Each return signifies a new block and a return with a two-space indent signifies a nested tag.

In the following example you will see how the `<article> ... </article>` tags are simply replaced with `%article`. You should also notice how I used indentation to nest the `%h1` and `%p` tags within `%article`. Last, see how I nested the `%a` tag within the `%p` tag.

```
%article
  %h1 primary header title for paragraph below
  %p
    %a{:href => "#"} Example of body text link
```

HAML doesn't stop there. In the previous section I showed how simple HTML can get harder to read by adding attributes to the markup. HAML helps this by removing statements like `class="..."` and `id="..."`.

The following example illustrates how HAML simply refers to an id by using the pound # symbol and a class is simply referred to with a period . symbol. Using this method I can take a HTML statement like `<article class="primary-article" id="main-author" ...` and reduce it to something as simple as `%article#main-author.primary-article`.

```
%article#main-author.primary-article
  %h1 primary header title for paragraph below
  %p
    %a{:href => "#"} Example of body text link
```

HTML href tags aren't exactly reduced in HAML, but the syntax uses a Ruby `key:value` pair style. Notice in the example how I replaced `<a href="#"></a>` with `%a{:href => "#"}`.

HAML was a radical new way of looking at HTML. HAML embraces standardized HTML and makes it easier to write, and cleaner to read.

It was from the success of these powerful concepts that the idea of Sass was born.

Good 'ol CSS

When Hampton Catlin first approached Nathan Weizenbaum, his idea was for a *HAML for CSS*. Much like HTML, standard CSS is full of repetition and the syntax relies on characters like semi-colons **;** to separate declarations and curly-brackets `{ ... }` to separate blocks of style rules.

The following example illustrates a common CSS selector with CSS rules. However, unlike HTML, nesting requires duplication of the parent selector(s).

```
header {
  width: 100%;
  height: 100px;
}

header .nav {             /* header is repeated */
  text-decoration: none;
  background: #fff;
  color: #333;
  border-radius: 5px 5px 0 0;
}

header .nav p {           /* header and .nav are repeated */
  font-weight: bold;
}
```

I don't know about you, but this repetition was something that always bothered me. Years before I ever encountered Sass I would beg the question, "*Why can't I just return+tab?*" As it turns out, others were thinking the same thing.

It should be noted, in the same way returns and indentation are best practice for formatting HTML, the same goes for block and declaration separation in CSS.

In the following example I illustrate three styles of CSS, expanded, nested and compact.

```
/* Expanded */
header {
width: 100%;
  height: 100px;
}
header p {
  font-size: 48px;
}

/* Nested */
header {
  width: 100%;
  height: 100px; }
  header p {
    font-size: 48px; }

/* Compact */
header { width: 100%; height: 100px; }
header p { font-size: 48px; }
```

Since CSS formatting relies on syntax for block and rule separation, CSS authors are able to use all three styles at the same time in the same document if desired.

In many cases there is a distinct reason why multiple styles are used in a single document. There are also a lot of cases where the code was simply edited by multiple developers who preferred one style over another. Going unchecked, this lack of style enforcement can quickly lead to chaos in the code and makes it almost impossible to maintain. Remember the campfire rule?

Another format style of CSS is `compressed`. This is typically used for file compression and not a writing style.

HAML for CSS == Sass

Like it's older brother HAML, Sass lacks the dependency on special characters, curly-brackets `{ ... }` and semi-colons `;`, that CSS does. Which means, Sass requires whitespace, returns to designate block and rule separation, and indentation for nesting.

The following example illustrates a CSS declaration that is associated to the parent selector by a `return` and then a `two-space tab`. Each following declaration is separated by a `return`. The level of indentation maintains the relationship between parent and siblings.

Notice that the nested `.nav` and `p` selectors are not duplicated but are using a `return` and then a `two-space tab`. Sass understands these relationship between selectors. The ability to nest without duplication, Sass' most fundamental feature, is clearly inherited from one of HAML's core principals.

```
header
  width: 100%
  height: 100px
  .nav
    text-decoration: none
    background: #fff
    color: #333
    border-radius: 5px 5px 0 0
    p
      font-weight: bold
```

It was this un-CSS looking CSS that made Sass a popular choice with early adopters using HAML. Many developers, especially those in the Rails community, were quickly replacing HTML with HAML and Sass was a clear choice as a replacement for CSS.

Some would argue, it was this same un-CSS looking CSS that kept many new users from using Sass. This new way of writing CSS, this new syntax, meant that your current code had to be either converted or trashed. To new users, not familiar with how CSS preprocessors worked, it was one thing to use Sass and it was another thing entirely to convert a project over.

Other CSS pre-processors force the hand

As Sass' popularity began to grow, competing CSS preprocessors began to emerge. A trend that clearly showed that not only Rails developers were interested in these ideas.

New languages, specifically LESS and xCSS, showed up on the scene and share many, but not all, of the same features of Sass. What made them stand out from Sass was their adoption of the standard CSS syntax. Developers using languages such as PHP or .NET were not yet used to the idea of indentation languages like HAML and Sass. Adding to this, UI developers who's bread-n-butter is CSS, were not to excited about Sass' syntactical approach either.

Sass found itself at the crossroads. CSS3 was gaining in strength and popularity. New features were being added to the spec at lightening speed. Not to mention, code examples that could quickly be executed in other competing languages required conversion in order to be used with Sass. Sure there was a quick command line tool to convert CSS to Sass, but those who were bothered by the syntax typically were not fans of the command line either. Use of the standard CSS syntax by competing languages made them more desirable to a new audience of preprocessor users. Frankly, Sass' original syntax was being viewed by some as a barrier to the language.

March 31, 2010, Nathan Weizenbaum announced Sass 3 beta release the new CSS-superset syntax to be known as "*SCSS*" or "*Sassy CSS*". It was becoming clear that Sass had outgrown it's humble beginnings as simply an aesthetic alternative to CSS. Sass was becoming a language all of it's own. A language that has it's own path and requires an approach that is not exclusively tied to HAML. This announcement, and the work that followed it, was in direct response to the changing world. Whereas HAML and the whitespace writing style was leveraged for Sass adoption in the beginning, SCSS was to intended to bring in a much larger community of CSS developers that Sass was eager to have.

The addition of the SCSS syntax was a major undertaking by the core team and one that was fully embraced. Even to the extent that the Sass reference docs were rewritten to use the new CSS extension syntax. A brief summary of the work done is as follows:

- SCSS was built from the ground up based on the CSS3 spec, and is 100% CSS3-compatible
- SCSS can do anything Sass can do
- SCSS files can import Sass files, and vice versa

It is important to note that the inclusion of the SCSS syntax did not mean the deprecation of the original whitespace syntax. Nathan Weizenbaum recognizes the indented sass syntax is here to stay that a large part of Sass' initial success was due in part to people who do not prefer the standard CSS syntax and he was not about to alienate a large portion of it's users.

Be it the original Sass syntax or the newer SCSS syntax, the good news is that you don't really have to choose. Positions, personalities and preferences will vary greatly as to how to write Sass. Maybe you choose to only write Sass, or maybe you write only SCSS, or maybe you do a little bit of both? Due to the complete compatibility of the two syntax, `.sass` files can live harmoniously with `.scss` files in the same project.

I should note, the Sass whitespace syntax and the SCSS syntax cannot live in the same file. That is it's only limitation.

Sass compared to SCSS

Sass is a language and a syntax. SCSS is a syntax of Sass. Both syntax support the same features equally. Throughout the book, I will interchangeably refer to Sass as the language and the syntax. SCSS will only refer to the syntax of the language.

The easiest way to identify Sass, of course, is the lack of semicolons ; to separate declarations, or curly brackets `{ ... }` to separate selector rules or blocks as shown in this example.

```
.block  // no opening curly-bracket
  font-size: 1em  // no semi-colons
  color: $text-color
  border: 1px solid $border-color
  // no closing curly-bracket
```

Advantage: Less characters to write. Style of writing that enforces code standards. Disadvantage: Any conventional CSS needs to be converted before it can be used. Using Sass meant learning a new style of writing for CSS.

SCSS, on the other hand, brings back the more familiar CSS appearance with the use of semicolons `;` to separate declarations and curly brackets `{ ... }` to separate rules as shown in the following example.

```
.block {              // opening curly-bracket
  font-size: 1em;     // semi-colons separating declarations
  color: $text-color;
  border: 1px solid $border-color;
}                     // closing curly-bracket
```

Advantage: No conversion required. Any correctly formatted CSS can be imported. Converting whole sites to SCSS simply means updating `.css` to `.scss` . If you are already comfortable writing CSS, you are good to go with SCSS. Disadvantage: Curly brackets `{ ... }` and semicolons `;` are back. CSS selectors are required to be more expressive.

I should also mention that all valid forms of writing CSS is now acceptable in writing SCSS. Illustrated in the following example, you will see the familiar CSS writing styles of `expanded` , `nested` and `compact` . Without a doubt, we will leave `compressed` for magnification and not illustrate that as an acceptable writing style.

```
// Expanded
header {
  width: $header-width;
  height: $header-height;
}

// Nested
header .nav {
  text-decoration: none;
  background: $background-color;
  color: $text-color;
  border-radius: $default-radius $default-radius 0 0; }

// Compact
header .nav p { font-weight: bold; }
```

SCSS' ability to combine different styles of writing is embraced by many of Sass' power users as illustrated in this sample CodePen example by John Long where he is creating a mixin using the `@content` directive.

```
@mixin keyframes($name) {
```

```
  @-webkit-keyframes $name { @content }
  @-moz-keyframes $name { @content }
  @-o-keyframes $name { @content }
  @keyframes $name { @content }
}
```

Sass, on the other hand, requires this code to be written in the following way, illustrating the necessity of whitespace.

```
=keyframes($name)
  @-webkit-keyframes #{$name}
  @content
  @-moz-keyframes #{$name}
    @content
  @-o-keyframes #{$name}
    @content
  @keyframes #{$name}
    @content
```

While SCSS supports a combination of writing styles that authors have come to appreciate, Sass' streamlined code writing style has a few shortcuts that are desired by many SCSS authors. A common feature, *mixins*, only requires a + symbol followed by the name of the mixin and the optional argument(s) as shown in this example.

```
block
  +buttons($button-color)
```

Whereas SCSS requires you to be more expressive, the full `@include` statement is required as shown.

```
block {
  @include buttons($button-color);
}
```

Sass' shorthand style of writing also carries forward when creating a new mixin. Notice the new mixin buttons is created by only using the = symbol.

```
=buttons($color)
  border-radius(3px)
  bacground-color: $color
  line-height: 1.5em
```

Whereas again, SCSS' expressiveness requires the full `@mixin` directive to be clearly stated.

```
@mixin buttons($color) {
  border-radius: 3px;
  bacground-color: $color;
  line-height: 1.5em;
}
```

Other directives like `@extend` and `@function` currently do not have a shorthand versions in either the Sass or SCSS syntax.

Probably one of the more interesting uses of SCSS is to import vendor styles written in CSS. As a matter of convention, I will create a folder in my Sass directory called vendor and include all my vendor specific CSS. By simply updating the file type from `.css` to `.scss`, these styles are now part of my Sass architecture.

Currently there is a debate about the ability to directly import a CSS file. While this can be accomplished within a Rails project via the asset pipeline, this is not a feature that Sass directly supports. I feel that there is some logic to this decision.

It is definitely considered best practice that once you start using Sass that all your files should be Sass and that having a mix of Sass and CSS can only create more problems then it solves.

One of the luxuries of working with a preprocessor is that at any time you can toss the temporarily processed CSS files and re-process your Sass. But if you have a mix of Sass and CSS files in your project, this increases maintenance complexity. I don't like things to be unnecessarily complex, so 100% of my processed CSS comes from either Sass or SCSS files.

This is not to say that this solution is for everyone. If you are a user that requires additional CSS resources to be included as native CSS, Chris Eppstein released a Gem that will do this for you. The Sass CSS Importer Plugin, as Chris describes it, "*The Sass CSS Importer allows you to import a CSS file into Sass.*"

In this section I took us on a short journey from the days when Sass was just a gleam in Hampton's and Nathan's eye all the way to present day when Sass has matured into a fully featured language that supports to distinctly different writing styles. Each style with impassioned users on either side. In the end, one is not better then the other and these different syntax types were created, and continually maintained, to serve different purposes.

It is up to you, the user, to decide what is best for you. Myself, I have taken the journey 360. I stated with the original Sass syntax, fully converted over to SCSS and now find myself easily writing both. Use each syntax to their strengths. Don't be overly dogmatic for one over the other, I hear these silly arguments all the time. There is a lot to be gained by simply understanding why someone else may prefer one style over the other.

Now that we know where Sass came from, in the next section I will discuss best practices for how to write clean Sass that will keep you from being a pain in the Sass to your team.

A Sass Style Guide

With all of Sass' new found powers, the responsibility of code quality is even more paramount. Leaving code littered with unclear rule separations, randomly imported mixins and no clear use of extended placeholders will quickly decrease the readability, scaleability and maintainability of your code. This will also increase the sneering and aggravation of other developers on your team. Guaranteed.

While there are no guarantees that following a few simple best practices will make your code better, it will make it easer to read and maintain.

Nesting: don't go too deep

Earlier we discussed a core strength of Sass, this being it's ability to remove unnecessary repetition from your nested CSS selectors. Whereas CSS requires selectors to be duplicated, Sass does not.

In the following CSS example, the parent selector `.block` is repeated each time as the author nests additional selectors for specificity. This pattern continues to repeat itself with each additionally nested selector. The `div`, the `ul` and the `li` are repeated as the author finally reaches the `p`.

```
.block div { border: 1px solid black; }
.block div ul { width: 100%; float: left; }
.block div ul li { float: left; width: 100%; background: orange; }
.block div ul li p { font-weight: bold; line-height: 1.5em; }
```

Sass' indentation syntax allows for the author to simply indent the nested child selector without repeating it's parent. As shown in the following example, each additional indentation tells Sass to inherit the previously nested selector.

```
.block div
  border: 1px solid black
  ul
    width: 100%
    float: left
    li
      float: left
      width: 100%
      background: orange
      p
        font-weight: bold
        line-height: 1.5em
```

With SCSS, the principal is the same. Indentation is used to assist in readability of the code, but it's the semi-colons ; that are required to separate declarations. Curly-brackets `{ ... }` are required for block separation and to designate selector nesting. Shown in the following example each newly nested selector is placed within a new set of curly-brackets `{ ... }`.

```
.block {                /* .block opening bracket */
  div {                 /* div opening bracket */
    border: 1px solid black;
    ul {                /* ul opening bracket */
      width: 100%;
      float: left;
      li {              /* li opening bracket */
        float: left;
        width: 100%;
        background: orange;
        p {             /* p opening bracket */
          font-weight: bold;
```

```
            line-height: 1.5em;
          }               /* p closing bracket */
        }                 /* li closing bracket */
      }                   /* ul closing bracket */
    }                     /* div closing bracket */
}                         /* .block closing bracket */
```

Nesting is a powerful feature, but be mindful of the inception rule. It is easy to fall into this comfortable trap of using selector inheritance to mimic your markup structure in attempts to dominate the cascade as clearly was the argument in this 37Signals article. Succumbing to the pressures of specificity, you will find yourself in a CSS selector nightmare. Style rules will be extremely difficult to reuse and very fragile to edit.

A common rule of thumb, if you are thinking of nesting a fourth time, you should ask yourself, "*Does this selector really requires this much specification?*" Can the code be abstracted so it can be used more universally? In CSS, and even more-so in Sass, keeping your selectors shallow and properly extending declarations will increase portability and reusability of your code.

Let's take an opportunity to refactor the previous example. In Sass we can easily reduce the level of specificity by simply removing some of the indentation.

```
.block
  div
    border: 1px solid black
  ul
    width: 100%
    float: left
  li
    float: left
    width: 100%
    background: orange
  p
    font-weight: bold
    line-height: 1.5em
```

With SCSS we can accomplish the same, but SCSS requires you redefine your nesting by moving the curly-brackets `{ ... }` as shown in the following example.

```
.block {
  div {
    border: 1px solid black;
  }
  ul {
    width: 100%;
    float: left;
  }
  li {
    float: left;
    width: 100%;
    background: orange;
  }
  p {
    font-weight: bold;
    line-height: 1.5em;
  }
}
```

Either Sass or SCSS, the following CSS output will be the same. The result, as shown, less nesting means shallower style rule definitions, better portability and faster code.

```
.block div { border: 1px solid black; }
.block ul { width: 100%; float: left; }
.block li { float: left; width: 100%; background: orange; }
```

```
.block p { font-weight: bold; line-height: 1.5em; }
```

Declaration listing best practices

How you write out your selectors has a direct impact on the readability of your code. Compounded with dynamic features like injecting code with mixins, this also has a direct impact on the output CSS cascade.

Much like standard CSS, I want my code to be readable, simple to follow and easy to update. I always list my parent specific declarations directly under it's selector and then list the indented child selectors to keep readability at a maximum.

See how I listed the CSS rules specific to `.foo` directly after declaring the selector. At the end of the parent selector's rules is when I declare it's nested, or child selector `.nested-foo`. Within this child selector I will continue to follow the same pattern.

```
.foo {
  font-size: 12px;
  padding: 10px;
  width: 50%;
  .nested-foo {
    background-color: green;
  }
}
```

Using mixins

Using the `@mixin` directive you can engineer smart and reusable code to be used throughout your application.

When using mixins within selectors, the placement of mixins has a direct impact on the cascade of your output CSS. The role of a mixin is to physically inject, or mix-in code, where referenced. When a mixin is randomly placed into a selector, depending on what is in the mixin, this could either accidentally over-write a preceding CSS rule or you may unknowingly write a CSS rule that is duplicated by the mixin.

Consistency plays a huge role in clean code. When mixins are always in a consistently expected location, this will help other authors be able to quickly scan and edit code.

In the following example notice how I included the mixin `transition` directly after declaring the selector. By doing so, we have a clear expectation as to the output CSS. After the included mixin is when I list the rules that are specific to this selector. Again, this pattern would follow suit with the nested `.nested-foo` selector.

```
.foo {
  @include transition(all, 0.6s, ease);
  background-color: orange;
  width: 50%;
  .nested-foo {
    width: 25%;
    margin: 0 auto;
  }
}
```

When using mixins, tt is considered best practice to only create mixins that use keyword arguments. By doing so, you aren't simply repeating code, but leveraging a pattern of attributes whose vales are being dynamically updated with each use.

Using extended selectors

If you are a follower of OOCSS, extending selectors should feel very natural to you. Much like mixins, Sass' `@extends` directive is a great tool for creating and managing repeated code. Unlike mixins, extends do not accept arguments and do not inject code where called. Instead, an extended selector will be modified in it's place of origin within the cascade by appending the extending selector.

In the following example I created the selector of `.default_gray_border` . In the following selector `.promoters_box` I extended `.default_gray_border` using the `@extend` directive.

```
// Style class object
.default_gray_border {
  @include border-radius(25px);
  border: 1px solid gray;
}
// Semantically named class
.promoters_box {
  @extend .default_gray_border;
}
```

You will see in the following output CSS that Sass concatenated the two selectors together as these selectors share the exact same CSS rules.

```
// Output CSS
.default_gray_border, .promoters_box {  // notice the chained selectors
  -webkit-border-radius: 25px;
  -moz-border-radius: 25px;
  border-radius: 25px;
  border: 1px solid gray;
}
```

When working with extends, I follow the same placement rules as with mixins as the consistent placement of extends will increase readability of the code. In the following example the extended selector is listed directly after the declared selector and then parent specific styled follow.

```
.foo {
  @extend .default-transition;
  background-color: orange;
  width: 50%;
}
```

Sass for styles, SCSS for logic

Leveraging the fact that .sass files can live harmoniously with `.scss` files, some developers have adopted a workflow of, using SCSS for all logic (mixins, extends and functions) and then using Sass for the CSS styling.

When writing style rules for your design, Sass is quick and get's the job done. Less key strokes and a quick tab in for nesting are all great things when writing code. An added benefit of Sass is the ability to quickly redefine a series of selectors and/or rules by simply changing the level of indentation, versus having to redefine nesting by moving around curly-brackets `{ ... }`. Let's face it, if you are using Sass, writing CSS should pretty much roll off your fingertips. Writing out selectors and rules is sans logical thought process and you are simply executing output at this time.

On the other hand, when it comes to complex thought processes, the use of curly brackets `{ ... }` has been stated to assist in a developer's ability to see the logical code groupings I have found that using the SCSS syntax when writing logical code helps me to slow down and really take a hard look at the code I am writing.

Good code habits include writing comments and Sass is no different. SCSS allows for comments to be placed inline with the line code you are referencing. Due to the whitespace specifications, this is not possible in Sass. SCSS' form of commenting allows developers to really be expressive in their notes.

Code comments

Sass supports both invisible and visible comments. Using `//` before any Sass, this will place a comment in your code, but will not be output in the processed CSS. Using the standard `/* */` CSS comments in your Sass, when processed this will be output in your CSS.

Leaving comments or instructions in your code is just good practice. I find it essential to leave good instructions behind about my code using the invisible technique as I begin to engineer increasingly more complicated Sass.

The following example is a sample of code from the Compass library illustrating a good use of comments.

```
//  override to change the default
$default-background-size: 100% auto !default;
// Set the size of background images using px,
// width and height, or percentages.
// Currently supported in: Opera, Gecko, Webkit.
//
// * percentages are relative to the background-origin
// (default = padding-box)
// * mixin defaults to: `$default-background-size`
@mixin background-size(
  $size-1: $default-background-size,
  $size-2: false,
  $size-3: false,
  $size-4: false,
...
```

Mixin, selector, function naming conventions

Sass' naming conventions are inherited from CSS. Lowercase, hyphen-separated names, like the following function examples, are considered standard.

```
@mixin text-format($size, $family, $color) {
  font: {
    size: $size;
    family: $family;
  };
  color: $color;
}

.site-header {
  @include text-format(12px, verdana, red);
}
```

While this may be preferred, you will see many examples out there using underscores `example_name` as well camel case `exampleName` or Pascal case `ExampleName`, there is no right or wrong here. As long as you are consistent in your naming convention, that's what really matters.

A word of caution. When naming mixins, but sure to always be consistent with using either dashes `-`, or underscores `_`. I am not sure if this is a bug or a feature, but when importing mixins, the dash and underscore can be used interchangeably with the same name and it will work.

In the following example I will crate a mixin and name it using a dash `-`. But in the selector below, I will include the mixin using an underscore `_`. The result will be the mixin will process into the selector without issue.

```
=block-mixin
  background: green
.block
  +block_mixin
```

The resulting CSS

```
.block {
  background: green;
}
```

Working with partials, manifests and globbing

With CSS, all of your code is contained in a single document and with each new feature this document increases in complexity while decreasing in readability and maintainability. Sadly, these same poor development practices have made their way into Sass development as well. To add insult to injury, it is not uncommon to see files with large blocks of code that include functional Sass like variables, functions, mixins as well the presentational selector specific Sass. While this will work in a pinch, it is not best practice to have all your code in one place.

Sass gives us the power to break our code into smaller, easier to manage chunks of code called *partials*. In this section we will discuss how to best break apart our Sass and how to stitch it back together via techniques like *manifests* and *globbing*.

Partials

Breaking code down to smaller chunks can be a difficult process if you do not have a good convention to follow. In the next chapter I will go into greater detail about how to best manage resources like variables, functions, mixins and presentational styles, but for now, let's understand that it is a better management technique to break out your mixins, functions, variables and presentational styles into different partials.

A partial is any file with an underscore `_` preceding the name. When Sass sees these files, it will not process them into CSS files. A partial requires that it be imported into another file that will inevitability be processed into CSS in order for it to be output.

In the following example, you will see a simple Sass architecture that illustrates this principal. Notice how `application.sass` is the only file that does not contain an underscore `_` in the name as this will be the file that is output to CSS.

```
stylesheets/
|-- application.sass    // Sass manifest file
|
|-- _reset.sass         // Partials
|-- _variables.scss             |
|-- _functions.scss             |
|-- _mixins.scss                |
|-- _base.sass                  |
|-- _layout.sass                |
|-- _module.sass                |
|-- _state.sass                 |
|-- _theme.sass         // Partials
```

You are not limited by the number of CSS files you need to output. This strategy could be used for the creation of browser or device specific styles as well. As illustrated in the following example, I have added more files for browsers like Internet Explorer and devices like mobile and tablets that I intend to output CSS.

```
stylesheets/
|-- application.sass        // Core manifest file
|-- IE8.sass                // Browser manifest file
|-- mobile.sass             // Device manifest file
|-- tablet.sass             // Device manifest file
|
|-- _reset.sass             // Partials
|-- _variables.scss                 |
...
|-- _module.sass                    |
|-- _state.sass                     |
|-- _theme.sass             // Partials
```

Manifests

When using a Sass file architecture made up mostly of partials, except for the ones that we intend to output as CSS, these output files will be your *Sass manifest file(s)*. A manifest will manage all of the Sass partials to be imported, as well import any Compass extensions or additional Sass code libraries in your project.

Unlike standard CSS, Sass' `@import` directive is part of the preprocess compile of your code. This does not require additional HTTP requests when sent to the client as all your CSS will be compiled, and minified if desired, into a single document. As your project scales, you are encouraged to break your files into smaller manageable chunks of code and reassemble via a manifest.

In the following example see how I use the `@import` directive to load all the partials into a single document and output as CSS. Another thing to note is the absence of a file type extension. Sass or SCSS, just like Honey Badger, `@import` doesn't care.

```
@import "variables";
@import "functions";
@import "mixins";
@import "reset";
@import "base";
@import "layout";
@import "module";
@import "state";
@import "theme";
```

The order in which you list your imports is the order that Sass will follow when processing your code. If the selector you wrote requires a mixin to be loaded before it is used, be sure to list the file containing the mixin prior to the file that uses the mixin.

It is common place in architectures like this that all logical Sass are imported first. In the previous example you can see that I loaded all the site's global variables, then functions followed by mixins. Once these files are loaded into memory, the following Sass files, who's functions are to create CSS, will be able to take advantage of the logical Sass code. This is a pattern we will want to repeat as we get deeper into a more complex file architecture.

Take note, loading logical code into memory for remaining Sass files to take advantage of, only works with imported partials. Remember that any file without a preceding underscore will be processed and output into a CSS file. If this requires any logic to be present in order to process rules, those files must be imported first.

In the following example I will illustrate how a stand alone CSS file needs to import Sass logic before it can process the CSS rules contained within.

```
@import "variables";
@import "functions";
@import "mixins";

.foo {
  background-color: $default-color;
  ...
}
...
```

When loading additional Sass code libraries such as Compass, in order to take advantage of the library's power, you are required to load these libraries first.

On the other hand, when loading CSS from plug-in apps, is most likely that you will want to load these last as not conflict with custom selectors you have written.

As well, if these plug-in styles require customization, importing them last will allow you to take full advantage of any libraries you have imported and custom code you have written.

The following example notice how I imported Compass first so that my project specific code can take full advantage of the Compass library. At the end of the manifest I then import my plug-in `flipclock` library.

```
// Included libraries
@import "compass/css3";

// Project specific code
@import "reset";
@import "variables";
@import "functions";
@import "mixins";
@import "base";
@import "layout";
@import "module";
@import "state";
@import "theme";

// Imported plug-in libraries
@import "flipclock";
```

While using a Sass manifest file is a great solution, there are some additional patterns we can use in order to keep this file from becoming a giant dumping ground.

A pattern I leverage is the use of additional manifests within sub-directories. Let's say for example that you begin to create a large resource of mixins in your project. As this file grows in size, it becomes increasingly harder to mentally parse. The suggested pattern is to break this file into smaller, more digestible chunks of code and place them into a directory. Using a manifest file within that directory, you import a single reference into your application manifest and add new imports to your more specific manifest.

The following example illustrates an updated file structure with a `mixins` directory containing a Sass manifest file. Notice the `_manifest` file contained within the `mixins` sub-directory.

```
stylesheets/
|-- application.sass        // Sass manifest file
|
|-- _reset.sass             // Partials
|-- _variables.scss                 |
|-- _functions.scss                 |
|-- _base.sass                      |
|-- _layout.sass                    |
|-- _module.sass                    |
|-- _state.sass                     |
|-- _theme.sass             // Partials
|
|-- _mixins/                // Directory
|  |-- _manifest.scss
|  |-- _grid_calc.scss
|  |-- _arrow_tooltip.scss
```

This update requires a very simple update to my site manifest file. In the following example you will see that I updated from a simple reference to a mixin Sass file, `mixins.scss`, to a manifest file contained within a sub-directory, `mixins/manifest`.

The naming of this file `_manifest.scss` is purely convention. Feel free to name this file anything you like, as long as it makes sense to you and your team.

```
// Included libraries
@import "compass/css3";

// Project specific code
```

```
@import "reset";
@import "variables";
@import "functions";
@import "mixins/manifest";  // import sub-directory manifest
@import "base";
@import "layout";
@import "module";
@import "state";
@import "theme";

// Imported vender libraries
@import "flipclock";
```

Globbing

Another technique available, although not native to Sass, is file globbing. Globbing refers to pattern matching based on wildcard characters that allows Sass to assemble all the partials within a directory without a specific manifest file. Whereas I stated earlier, the order in which files are imported and processed is dictated by the order in which they are listed, this is not the case with globbing. Without a specific list to go by, Sass will assemble the files in alphabetical order.

Globbing is not the answer to all importing cases. For example, if you have a sub-directory for a UI pattern or module, it is common place to see files like `_variables.sass`, `_mixins.sass` and `_module.sass` within the directory. By order of the alphabet, the `_variables.sass` file will be loaded last. This will break the Sass processor as it is likely that the mixin or module will require a value for a variable listed in `_variables.sass`. In these cases I am left with coming up with a naming convention to ensure the appropriate alphabetical order. That is a really bad idea.

I strongly recommend that in the cases where a specific order of importing is required, globbing is not the answer and make use of the sub-directory manifest pattern.

On the other hand, if you have a library of code where it doesn't matter at all what the order of import is, then this is a great solution. For example, a sub-directory of animation mixins is a great use for globbing. A directory of functions, again, a great use for globbing.

Globbing is used by a lot of developers. If you are a Rails developer, this feature is made available to you via the `sass-rails` Gem. If you are not using Rails, Chris Eppstein has made this feature available to all users via a plug-in Gem.

In the following example I will illustrate how globbing allows me to do away with sub-directory partials. See how all the files contained within `mixins` are imported via the wildcard /* expression.

```
...
@import "functions";
@import "mixins/*";   // import sub-directory manifest
@import "base";
...
```

The wildcard expression */ *is optimal if there are no sub-directories contained within the directory you are globbing. In the situation I am using additional sub-directories, the expression of /**/* is required.

```
...
@import "functions";
@import "mixins/**/*";   // import sub-directory manifest
@import "base";
...
```

Keeping your code modular and managing an easy-to-follow manifest file will reap great rewards as your project scales. This process also will assist you in the future as you begin to engineer smarter and smarter code that you would like to

reuse between projects.

Rules to live by

Starting a new project, as a developer you are typically given a design. The color scheme has been selected and the action buttons have been outlined. The Header, Footer, and most of the content is documented for you. Before you dive into writing the CSS, it is prudent to look at the design as a whole and take the time to break it down to it's individual components or objects. The mistake of writing the CSS before these points are considered can lead to a great deal of bloat within the stylesheet.

If you are familiar with the concepts of Object Orientated Cascading Stylesheets (OOCSS) and semantic HTML/CSS, then this chapter should be a good review.

For those of you who are new to OOCSS, I encourage you to pay close attention. Although OOCSS has little to nothing to do with Sass, learning these principals is a solid foundation to writing scalable and maintainable Sass.

Object Oriented Cascading Stylesheet (OOCSS)

OOCSS is built upon two main principles:

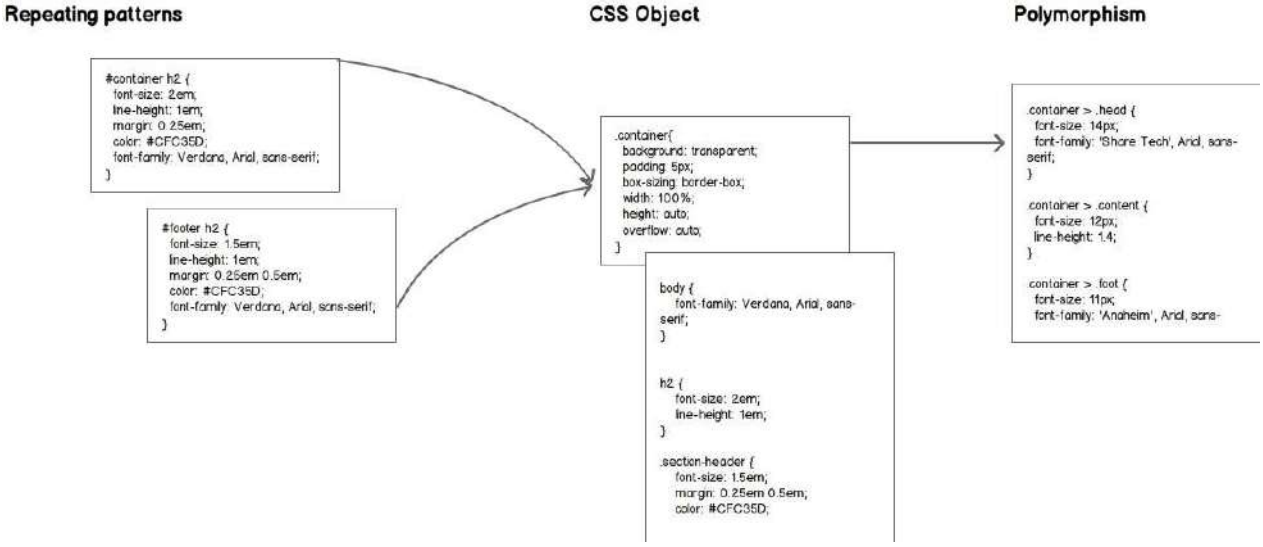
- Separation of structure and skin
- Separation of container and content

To better understand these principles, let's take a look at what OOCSS is and what it is trying to solve.

What is OOCSS?

A concept established by Nicole Sullivan with the idea being very simple; write CSS that is scalable, maintainable, and semantic. In a nutshell - object oriented. OOCSS by itself is a powerful tool. OOCSS combined with Sass, is even a more powerful tool.

Repeating patterns in your stylesheets lend themselves to the creation of objects which can be sub-typed or super-typed. These patterns can be polymorphic, thus infinitely reusable through inheritance.



Unruly CSS written without the forethought of OOCSS can have any or all of the following problems:

- Writing styles based on the view or the components as a whole can lead to bloated CSS
- Lack of global defaults will lead to repeating style patterns, adding more bloat
- Incorrect use of selector inheritance combined with heavy bloat can lead to poor CSS performance and typically exaggerated use of the !important tag

In this section, I will discuss how to avoid these issues by properly implementing the tenets of OOCSS. This will allow you to create a well established framework that, when combined with Sass, will allow you to further enhance the scalability, maintainability and re-use of the CSS.

To better understand OOCSS, we must be able to breakdown our design into small re-usable components. In order accomplish this task, we must understand what is a CSS object.

CSS Object

Think of your CSS as a bicycle assembly line using individual objects like wheels, handlebars, brakes, and seats. Our assembly will produce the final product, a bicycle. There will be slight variations on some of the objects, for example size of the wheels or color of the frame, which are issues that can be handled along the assembly line.

In order to efficiently assemble the bicycle, the original prototype is broken down into its individual pieces and the assembly. These individual parts are analyzed and the assembly line is created in order to efficiently put together the bicycle. A very similar concept can be applied to CSS. The design is analyzed and broken down into its individual working pieces. These CSS Objects are created in order to be re-used on the site, or our assembly line. The final product, the implementation of the design, is the culmination of the individual CSS objects. The OOCSS wiki further explains CSS objects as:

> "A CSS object consists of four things: HTML, which can be one or more nodes of the DOM, CSS declarations about the style of those nodes all of which begin with the class name of the wrapper node Components like background images and sprites required for display, and JavaScript behaviors, listeners, or methods associated with the object."

This can be confusing because each CSS class is not necessarily an object in its own right, but can be a property of a wrapper class.

Let's take a closer look at a section heading example. We can write the heading CSS like this:

```
#container h2 { /*IDs limit use of rules*/
/* Over-qualification with h2 tag restricts use to only this element */
  font-size: 2em;
  line-height: 1em;
  margin: 0.25em;
  color: #CFC35D;
  font-family: Verdana, Arial, sans-serif;
}
```

An issue with the above code is that it is an over-qualified selector that is limited to a h2 element that is nested within #container. When the same or slightly similar style needs to be applied in another area, the footer for example, I will have to duplicate the CSS rules as such.

```
#footer h2 { /* Only difference between this and the previous style is the font-size and extra margins. */
  font-size: 1.5em;
  line-height: 1em;
  margin: 0.25em 0.5em;
  color: #CFC35D;
  font-family: Verdana, Arial, sans-serif;
}
```

The following is an example of a repeating pattern that can be abstracted as one or several CSS objects to be used individually or as an assembly of styles. First I need to abstract some of the styles to the default globals section.

The following selectors will create CSS rules that are applied to base elements like the body and h2. These rules are no longer assigned to an over-qualified selector as before. However, more minute styling needs to be created in order to finalize the desired display.

```
body {
  font-family: Verdana, Arial, sans-serif;
}
h2 {
  font-size: 2em;
  line-height: 1em;
```

```
}
```

At this time, it is prudent to have an understanding of what the style is supposed to be used for and what it will represent. In this case, we are trying to create a header style for a section on the site. This knowledge will help us not only create the style, but also give it a semantic class name.

```
/* The font-family, which is usually applied on the entire site, is applied at the body. */
body {
  font-family: Verdana, Arial, sans-serif;
}

/* The <h2> element has CSS rules that will be applied anywhere the element is used. */
h2 {
  font-size: 2em;
  line-height: 1em;
}

/* The new semantic class .section-header reduces the font-size and increases margin-left and margin-right. This new se
.section-header {
  font-size: 1.5em;
  margin: 0.25em 0.5em;
  color: #CFC35D;
}
```

The above example is a simple implementation of OOCSS. As we can see, the design of a section header has been broken down into it's individual styles and CSS objects have been created accordingly. Continuing, I will look further into the OOCSS guidelines and how to properly implement them.

OOCSS Guidelines

Now that we are familiar with the building blocks of CSS, a CSS object, we will dive further into the guidelines of OOCSS. We delve deeper into the two main principles of OOCSS:

- Separation of content from container
- Separation of structure from skin

Implementation of OOCSS requires not only to follow the mentioned principles but also the prudent review of the design and understanding the containers, content, structure and skin. Let's take closer look at these guidelines and principles.

Separation of content from container

The main culprit that violate this guideline are location dependent styles. A simple example of location dependent style is something like this:

```
.footer a {
  color: #ccc;
}
```

This example is applying a gray color to the anchor tag in the footer (or the anchor tags that are children of an element with the class of .footer). This will limit the usage of this style (although it should be noted that this style can be used outside of the footer, however this will make the style non-semantic and confusing). The correct approach is to allow the default style of the anchor tag be applied to it and create a style to override and apply the needed change.

```
a {
  color: blue;
}

.footer-link {
  color: #ccc;
}
```

Let's take a look at a more complicated (and more wrong) example:

```
#content #mainInfo p img.left {
  float: left;
  border: 0;
  padding: 2px;
  padding-right: 25px;
}
```

This style will only be applied to the `<img>` element with a class of `.left` that is nested in a `<p>` element which in turn is nested in an element with the id of `#mainInfo` and which in turn is nested in an element with an id of `#content`. If we want to apply the same style somewhere else we will have to re-created this style all over and apply it specifically to that element. So, how do we fix this? First thing is to examine what it is that we are trying to accomplish. What we are trying to accomplish is to float the `<img>` element to the left. The question is, do we only want to float an `<img>` element? Most likely the answer is no. We want to create a style that is robust enough to be applied to any element. As a result the first class we are going to create is this:

```
.float-left {
  float: left;
}
```

The style of setting no borders on the `<img>` element is something that maybe we want to set for all `<img>` elements and the exception would be to add border to the `<img>`. As a result, our CSS styles would be:

```
img {
  border: 0;
}

.float-left {
  float: left;
}
```

To attack the padding, we must consider padding for the element on the entire site. If we think that all `<img>` elements should have a padding of 2px then this is a style that should be applied at the `<img>` element level and then the exceptions should be overridden. The same logic applies to all other elements like `<div>`, `<section>`, `<ul>`, etc... So in this case, I would amend the code to something like this:

```
img {
  border: 0;
  padding: 2px;
}

.float-left {
  float: left;
}
```

This style also needs a padding on the right hand side of 20px. In order to apply this desired effect, we must consider the two options that we have. First option is the consider what we are creating this padding. In this situation, let's say that these images are thumbnails for an image gallery that we will be lining up in order for the user to view and click on them to view a larger view of the image. Therefore in this case, we will create a semantic class and apply the padding style to it.

```
.gallery-thumbnail {
  padding-right: 20px;
}
```

Our other option is to create a series of styles that will allow us to override individual paddings (and margins) not only in this situation but also in all others. The OOCSS Github site has the following padding and margin CSS code:

```
/**
 * Spacing classes
 * Should be used to modify the default
 * spacing between objects (not between nodes of the same
 * object)
 * Please use judiciously. You want to be
 * using defaults most of the time, these are
 * exceptions!
 * <type><location><size>
 */
/* spacing helpers
p,m = padding,margin
a,t,r,b,l,h,v = all,top,right,bottom,left,
horizontal,vertical
s,m,l,n = small(5px),medium(10px),large(20px),none(0px)
*/
.ptn,.pvn,.pan{padding-top:0px !important}
.pts,.pvs,.pas{padding-top:5px !important}
.ptm,.pvm,.pam{padding-top:10px !important}
.ptl,.pvl,.pal{padding-top:20px !important}
.prn,.phn,.pan{padding-right:0px !important}
.prs,.phs,.pas{padding-right:5px !important}
.prm,.phm,.pam{padding-right:10px !important}
.prl,.phl,.pal{padding-right:20px !important}
.pbn,.pvn,.pan{padding-bottom:0px !important}
.pbs,.pvs,.pas{padding-bottom:5px !important}
.pbm,.pvm,.pam{padding-bottom:10px !important}
.pbl,.pvl,.pal{padding-bottom:20px !important}
.pln,.phn,.pan{padding-left:0px !important}
.pls,.phs,.pas{padding-left:5px !important}
.plm,.phm,.pam{padding-left:10px !important}
.pll,.phl,.pal{padding-left:20px !important}
.mtn,.mvn,.man{margin-top:0px !important}
.mts,.mvs,.mas{margin-top:5px !important}
.mtm,.mvm,.mam{margin-top:10px !important}
.mtl,.mvl,.mal{margin-top:20px !important}
.mrn,.mhn,.man{margin-right:0px !important}
.mrs,.mhs,.mas{margin-right:5px !important}
```

```
.mrm,.mhm,.mam{margin-right:10px !important}
.mrl,.mhl,.mal{margin-right:20px !important}
.mbn,.mvn,.man{margin-bottom:0px !important}
.mbs,.mvs,.mas{margin-bottom:5px !important}
.mbm,.mvm,.mam{margin-bottom:10px !important}
.mbl,.mvl,.mal{margin-bottom:20px !important}
.mln,.mhn,.man{margin-left:0px !important}
.mls,.mhs,.mas{margin-left:5px !important}
.mlm,.mhm,.mam{margin-left:10px !important}
.mll,.mhl,.mal{margin-left:20px !important}
.mra,.mha{margin-right:auto !important}
.mla,.mha{margin-left:auto !important}
```

The above code can be placed in a separate CSS file and imported into any CSS file as desired. It is ideal that the styles that are applied to an element tell the story of all the styles being applied. So I don't see any issue with substituting the class `.ptn` with `.padding-top-none` or `.pbl` with `.padding-bottom-large`. So in order apply the desired original style to the `<img>` element, the HTML will look something like this:

```
<img src='path/to/the/image'
alt='appropriate text' class='float-left padding-right-large' />
```

Or to apply the `.gallery-thumbnail`, it will look something like this:

```
<img src='path/to/the/image'
alt='appropriate text' class='float-left gallery-thumbnail' />
```

Separation of structure from skin

To build a site that the structure or the markup or the HTML separated from the skin or the style or the CSS, requires a lot of dedication and forethought. Prime example of this separation is the example of CSS *Zen Garden*. Zen Garden is a site that allows designers and developers to create new websites not by changing the HTML but by creating a whole new CSS.

In order to have a good grasp of this concept, let's create some individual building blocks for our site. Creating individual CSS objects that can be assembled to create the desired style is one of the main guidelines of OOCSS. One of the most commonly used and created CSS object is the container object (which also sometimes goes by the name of block object or panel object). A starting container object can look something like this:

```
.container {
  background: transparent;
  padding: 5px;
  box-sizing: border-box;
  width: 100%;
  height: auto;
  overflow: auto;
}
```

However there will subtle variation that will be extended or created to compliment or assemble other styles. For example in our online resume project, we have a container on the left, our side navigation bar, which has dark background and light colored text. As a result, we will create a style as such:

```
.side-nav {
  background: black;
  color: white
}
```

If a bordered container is needed then create a selctor like the following.

```
.bordered {
  border: 1px solid #cecece;
}

.main-content-bordered {
  border: 5px solid #b7d9a8;
}
```

It is the assembly of the individual objects that will allow for flexibility in creating individual components and that is part of the OOCSS process. As we get into more details, we can see that more assets can be added to individual areas. For example, a container may contain individual areas of header, footer, and body that can be expressed in the following way.

```
.container {
  background: transparent;
  padding: 5px;
  box-sizing: border-box;
  width: 100%;
  height: auto;
  overflow: auto;
}

.container > .head {
  font-size: 14px;
  font-family: 'Share Tech', Arial, sans-serif;
}
```

```
.container > .content {
  font-size: 12px;
  line-height: 1.4;
}

.container > .foot {
  font-size: 11px;
  font-family: 'Anaheim', Arial, sans-serif;
}
```

OOCSS is a great set of rules that will allow for a better built CSS framework. It will also help in reducing a tremendous amount of CSS bloat which in turn will help quite a bit in CSS and site performance. For further information on OOCSS visit OOCSS Github site.

Semantic vs. presentational classes

As I was learning to style web pages through CSS (a while back), I went through a lot of books and articles in order to learn the best practices. Writing semantic HTML and CSS was on top of the list of best practices. However the recommendations for best practices varied. Some recommend strict adherence to semantic nomenclature yet some see the benefits of a mixture of semantic and presentational classes. Before we dive into the *semantics* of these arguments, lets chat a bit about what is meant by semantic and presentational classes.

Semantic Classes

The definition of Semantic classes can vary among different developers. Its true definition is:

- of or relating to meaning in language
- of or relating to semantics

Here are what some people in the CSS and development community discuss Semantic classes:

> "Class names need to represent the object structure you are defining, not the specific visual look and feel of this particular instance."

- **Nicole Sullivan June 12th 2010**

> "I'd describe semantics as it relates to HTML as tags, classes, IDs, and attributes describing but not specifying the content they enclose."

- **Chris Coyier August 4th 2011**

> "A semantic CSS selector should reflect the intended structure or meaning of the element it is applied to."

- **Rob Dodson June 9th 2012**

> "Semantics concerns itself with elements and not the names assigned to them. Using the correct element for the correct job is as far as semantics goes. Standards concerning naming of those elements is all about sensibility."

- **Harry Roberts August 2010**

When it comes to writing semantic code, not only do opinions differ, but implementation is somewhat subjective and slightly elusive. Some follow a strict adherence to semantic naming that starts from the HTML tags and extends into the selector naming. This is an area where developers are opinionated. What makes it harder is that there is no framework or guideline that can be used to point us in the right direction.

We know that we stand on murky ground, so let us concentrate on what we know that is stable and can be reliable. As stated, a semantically named selector describes the object's structure and the content which will give us an idea of how the selector's rule can be applied.

Using `.article-title` for example, gives us an insight into the semantic nature of the style. On the other hand, a selector named `.blue-header` in not only un-semantic, but is clearly overly descriptive of the presentation. What happens when the design changes? Do you change the rules of `.blue-header` or do you create a new selector and then update the HTML elements that are referring to `.blue-header`?

A selector like `.article-title` is more descriptive of the content and will probably remain in the re-design of the site. This does beg the question, is it acceptable to ever use presentational styles?

Presentational Classes

Presentational classes are selector names that give us an insight into the presentation that is being applied. Some are like our previous example `.blue-header`, when applied will apply the style that is reserved headers type that has a blue color. These classes are considered un-semantic and not recommended for usage. Even in the W3C specification draft states the following:

There are no additional restrictions on the tokens authors can use in the class attribute, but authors are encouraged to use values that describe the nature of the content, rather than values that describe the desired presentation of the content.

An exception to this are examples of presentation styles used as helper or utility classes. For example, `.pull-left`, `.is-hidden`, `.display-inline` or `.is-active`. These types of classes are created to be used as modifiers where, for example, by applying the selector to an element that is to hidden from view, pulled to the left or is showing it's active state.

Semantic and Presentational Classes

When it comes to writing CSS, some fall under the camp of never using presentational classes and always being semantic vs. some that have the lax approach of not applying the semantic rules, sometimes to a fault.

The reasons behind the semantic class guideline are:

- Separation of content from presentation
- Search Engine Optimization
- Sensibility

While I do not disagree with the semantic approach, I do however do not adhere to the strict "*all semantic all the time*" motto. The principle of "*describe the nature of the content, rather than [...] presentation of the content*" is the major guideline that should be followed. In that same line of ideology, we should use all the weapons available to us i.e. class, attributes, id, and tags to tell the story of content. For example, let's examine this bit of code for a blog site.

```
<div>
  <div id="article">
    <p class="bold-font 14px-large-font">
      How to write an un-semantic front end
    </p>
    <p class="regular-font">The content will go here</p>
    <div id="comments"></div>
  </div>
</div>
```

In comparison to the story teller's version.

```
<section id="main-content" role="main">
  <article class="blog-content">
    <h3>How to write an semantic front end</h3>
    <p>The content will go here</p>
    <section class="comments"></section>
  </article>
</section>
```

The latter example can let us know all the information about the different sections of site without exposing the individual presentational aspects of it. However I do not think that the total avoidance of presentational classes are necessary. The existence of some utility/presentation classes like `.is-hidden`, `.pull-left`, or `.image-replacement` is necessary only to be used in exceptional circumstances. However if you review your code and see too many of these utility classes, it is time to refactor your solution and pull them into the semantic frame of your stylesheet.

File management

CSS has had a long and sordid past. A developer never sets out with the goal of making a complete and total mess of things. Their intention is not to build something that is practically illegible, impractical to maintain and is limited in scale. But somehow, this is where many inevitably end up. Luckily, all is not lost. With some simple strategies, organizational methods and out-of-the box tools, we can really help get things in order.

For many getting started with Sass, at one time or another, have created a junk-drawer of files. For most, this was a rookie mistake, but for others, this is a continuing issue with our architecture and file management techniques. Sass doesn't come with any real rules for file management so developers are pretty much left to their own devices.

In this chapter we will cover an array of file management practices that have varying levels of success and failure. It is over the years that I have honed in on one particular structure that continues to be successful in multiple development environments.

Large CSS files and increased complexity

CSS started out with very simple intentions, but as table-less web design began to really take a foothold, our stylesheets quickly began to grow in size. Developers tried to break them into smaller documents, but these strategies proved to have serious performance issues. Linking to multiple style-sheets meant multiple server round-trips adding the time it takes for a page to acquire it's necessary resources, not including the time it takes to transfer the data.

It's not entirely uncommon to see multiple links to stylesheets in websites.

```
<link rel="stylesheet" href="stylesheets/reset.css">
<link rel="stylesheet" href="stylesheets/base.css">
<link rel="stylesheet" href="stylesheets/skeleton.css">
<link rel="stylesheet" href="stylesheets/font-awesome.css">
<link rel="stylesheet" href="stylesheets/buttons.css">
<link rel="stylesheet" href="stylesheets/layout.css">
```

The practice of importing stylesheets into the one stylesheet linked in the HTML document was adopted by many as shown in this example from the MOZILLA DEVELOPER NETWORK. This technique had many promisses. It not only supported breaking your CSS into manageable documents but it also supported media types.

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
@import 'custom.css';
@import url("chrome://communicator/skin/");
@import "common.css" screen, projection;
@import url('landscape.css') screen and (orientation:landscape);
```

It's ultimate failure was that this feature has such a negative impact on web page performance, as pointed out by Steve Sounders back in 2009, it's practice was quickly abandoned. Using the link method, the stylesheets are loaded parallel (faster and better). The `@import` method loads any extra css files one-by-one (slower), and potentially gives you flash of un-styled content.

Looking for new solutions, developers began to adopt the use of CSS preprocessors to manage growing CSS code bases, but sadly didn't change old habits. Still clinging to the practice of creating large documents, many placed mixins and variables at the head of the doc and simply hashed out a bunch of CSS rules in the body. To make matters worse, as the documents began to grow in size, mixins and variables began to show up at random places within the stylesheet.

Realizing the need for better management techniques, many began to break these large stylesheets into smaller documents based on common principles like variables and mixins. Typography, forms and global design soon followed. Sure this reduced file size and increased readability, but without a real strategy this process was easily doomed. As files grew in number, sub-directories quickly gave way to junk-drawers of haphazardly daisy-chained files, filterable only my failed attempts at naming conventions.

Controller/action based styles

Inspired by Model-View-Controller(MVC) frameworks, mainly Rails, developers began to adopt these file structure solutions to help solve their issues. While there is some merit to this in regards to template/layout styles, this practice inevitably lends itself to creating styles that are too specific to each individual view and not easily reused throughout the rest of the application.

As a site's design progressed, duplicated code began to reveal itself between views. Attempts at abstraction, following another MVC pattern, typically resulted in the creation of a `/partials` directory. Basically, a simple repository for custom built mixins, variables and other reusable code. In essence, a Sass junk-drawer.

Following another Rails MVC patterns, developers attempted abstract away from the view and organize their files based on actions. If the visual elements are part of an action, making directories based on these actions makes sense, right? Sadly, this quickly falls apart as not all UI elements can be easily categorized this way. Random files again populate the directory, universal widgets, plug-ins, and custom mixins begin to collect. Once again we find ourselves suffering from the Sass junk-drawer effect.

Learning from our mistakes

Life is about journeys. It was during my journey of 'doing it wrong' that I began to see clearly. Ironically, I had the right solution all along, but didn't realize it. While part of a team developing an enterprise CMS, our process was to decompose a site's UI to it's lowest common elements. From those elements we could then build modules and then finally assemble the view templates. Each step building on the previous. Although my stylesheet management techniques weren't perfect, my concept of UI abstraction was solid.

In 2009, I began working with a new team, sans a CMS and my first encounter with Sass. I approached the project with the same conceptual understanding, but the outcome was drastically different. The code became increasingly harder to reuse and making simple edits resulted in the re-engineering of HTML as well as CSS. Post launch, I sat down and analyzed the code I wrote. I came to the realization that, as a team, we were engineering our UI (CSS and HTML) from entirely the wrong perspective. We were approaching our development from the full page perspective. Engineering all our visual elements from the outside-in and scoped to a specific view.

I started thinking back to the processes I pioneered with the CMS. Patterns established in the framework dictated we start from the elemental perspective; type, colors, forms, basic UI chrome (borders, shadows, icons, etc) all coded first. Once those base element styles were completed, it was a matter of applying the skin to the individual CMS modules. The modules then in-turn were used to assemble the view within the various automated templates. It worked quickly and seamlessly. Building the UIs from the inside-out was clearly the right solution.

Elements, modules and layouts

Applying these principals to new projects was the next challenge. First we need to be better at decomposing our designs. The inside-out approach is key to this process, the goal of the file structure, and necessary for a scalable architecture. Simply put, code the element, create the module and assemble the layout.

At this point you may be drawing parallels with Jonathan Snook's SMACSS approach, as you should. The approach I adopted shares many similarities that were later outlined in Jonathan's book. But alas, there are differences due to the vary nature of Sass and the additional features that it supports over vanilla CSS.

- Base
- Layout
- Module
- State
- Theme

While many of these concepts are solid ways to architect a scalable CSS structure, they are somewhat constrained by the limitations of vanilla CSS. But this is a book about Sass and we are not as easily constrained by these limitations.

In SMCASS, there are five types of style categories:

Base

Core styles that will make up your site or application. Base rules are commonly applied to an element using the element selector itself. Thrown into this bucket would be the CSS reset, typography, forms and buttons. These base styles will each have their own Sass file for easy management, here is an example from a `_typography.scss` Sass file.

```
html {
  font: em($font_size, 16) $primary_font_family;
  line-height: baseline($font_size);
  color: $primary_text
}

// Heading CSS rules
// -----------------------------------------
h1, h2, h3, h4, h5, h6 {
  @include heading();
}

// Standard body text support
// -----------------------------------------
p {
  margin-bottom: baseline-margin($font_size);
  text-indent: 0;
}

a {
  color: $href_color;
  text-decoration: none;
  &:hover, &:active {
    text-decoration: underline;
  }
  &:visited {
    color: $href_color;
  }
}
```

Modules

SMACSS and I share a similar point of view. Consider modules as 'nouns' in your code. These are the 'things' that you will be making the most use of. Leveraging the styles created in base, I can begin to create small and large modules that can be infinitely reused.

Where Sass begins to add super powers is it's ability to break down modules even further into infinitely more reusable code. The use of mixins and placeholder selectors strike an even more amazing balance between presentational selectors in your CSS and semantic selectors in your markup.

State

State is a powerful concept and again fully supported. State typically is an over-ride to the default style placed on an element or module. If the module is a noun, then the state is the module's verb. Common concepts are `.is-hidden` or `.is-error`.

Where I slightly differ with Sass is I feel that state rules are better managed within the parent selector versus always having completely separate rules. This is not a fundamental change in the state concept, but an area where Sass shines a light on the limitations of vanilla CSS.

SMACSS emphasizes using the `!important` tag when creating stand alone selectors. I find this approach to be somewhat problematic. Instead, I advocate for not only creating a standalone default rule for the management of state, but in the cases where specificity is needed, nesting the state rule in the module it is designed to augment is preferred.

Layout

In regards to layout, I take a different approach all together. SMACSS considers layout as slightly different representation of the module, conceptualizing them as major page components. Examples would be `.site-header`, `.site-footer` and `.main-nav`.

I on the other hand, as discussed further in this chapter, consider layout to be a more holistic assembly of a view or template for modules to be inserted. I consider the layout to be the structural CSS that comprises the grid in various states depending on user input and environment. To me, the layout is the key to easily manage responsive web designs as so may of the UI decisions are based on the layout, not the module themselves.

Theme

Much like layout, I take a completely different approach all together. Sass' ability to create UI variables, mixins and placeholder selectors allows me to re-write the concept of CSS theming. In SMACSS theme is not considered part of the core types, where I see it as an essential player on the construction of the site design.

A good example of a theming solution would be with a tool-tip bubble. For example the tool-tip module would consist of inheriting typography, applying some shape and aesthetics such as a carrot. The theme of the tool-tip would be it's padding, color, border and possibly a shadow effect. With vanilla CSS, it is recommended that you engineer the module itself, then later in a "theme" section of your CSS you again reference the selector to add it's theme rules.

With Sass we can take this to a whole new level by engineering the tool-tip as a self contained module either as a mixin that accepts arguments or a placeholder selector with variables. Selectively enhancing the module at the time of placement and coupled with the concept of a `_config.scss` file, I can list out all the configurable "theming" parts of the site's design.

File structure

Here I propose the following file structure that embodies this point of view. In the root there are individual Sass partials to address the elemental parts, directories for more complex concepts and last is a manifest file to aggregate all the awesome.

```
|- sass/
|--- buttons/
|--- color/
|--- forms/
|--- layouts/
|--- modules/
|--- typography/
|--- ui_patterns/
|--- vendors/
|--- _buttons.scss
|--- _config.scss
|--- _forms.scss
|--- _reset.scss
|--- _typography.scss
|--- application.scss  // the aggregate manifest file
```

Sass partials and the manifest

As discussed up to this point, it is the desire of many developers to break down their CSS into smaller, more manageable resource files. Even SMACSS clearly advocates for breaking down all of it's core types and modules into separate `.css` files. There is a huge flaw in this since vanilla CSS does not support the aggregation of these resources. You are either left with linking multiple CSS resources, using the non-performant `@import` native CSS feature or looking to more complex solutions.

Sass' solution to the problem is to natively support partials. Sass can be broken into smaller, more manageable resource files and by default any `.sass` or `.scss` fill will be processed into CSS.

```
|- sass/
|--- buttons.scss
|--- forms.scss
|--- reset.scss
|--- typography.scss
```

Will output the following:

```
|- stylesheets/
|--- buttons.css
|--- forms.css
|--- reset.css
|--- typography.css
```

Frameworks like Rails will take all these individual CSS files and compress them into a single CSS file for production, but we are not all Rails developers. Not to mention that the Rails documentation states that this solution was not designed for multiple Sass files and you should use Sass' native `@import` rule instead of Rails' Sprockets directives.

Partials are a powerful weapon in the Sass arsenal. Simply put, any file that has an underscore before the name, `_partialName.scss`, will not be processed into a `.css` file by itself. It is required to be imported into a file that will be processed into CSS. Editing the previous example, we will make each Sass file a partial and use a manifest file to aggregate the partials using the `@import` rule.

```
|- sass/
|--- _buttons.scss
|--- _forms.scss
|--- _reset.scss
|--- _typography.scss
|--- application.scss
```

Will output the following:

```
|- stylesheets/
|--- application.css
```

Manifest files

At this level of the file structure example, the only file that is processed into CSS is the `application.css` manifest. It's here where all your custom add-ons, configs, elements, modules, views, mixins, extends, etc., are all imported and processed into a production style-sheet. It is important that this file be kept devoid of any presentational CSS rules.

An example manifest file would only contain instructions and the imported Sass files as illustrated in the following.

```
// App Config - this is where most of your magic will happen
// ------------------------------------------------
@import "config";  // The config file sets the theme for the project

// Import core Sass libraries
// ------------------------------------------------
@import "lib/bourbon/bourbon";
@import "lib/colors/manifest";
@import "lib/typography/manifest";

// Standard CSS reset stuff here
// ------------------------------------------------
@import "reset";
```

This is not to say that application.scss is the only Sass manifest file. Manifests can import other manifests. A pattern that helps to keep the application.scss manifest easy to read while keeping sub-directory files nicely organized.

Manual manifest files or glob-imports

Manual manifests are just that, manual management of imported Sass files from sub-directories. This is a good practice to follow when you need specific control over the inheritance of files. Example, rule A needs to come before rule B in the output cascade.

Glob-imports on the other hand is a way for you to simply point to a directory in your manifest, `@import "directory/*"`; and Sass will import all the files in alphabetical order. This is great for a directory of mixins or functions that simply need to be loaded in memory for Sass to process the CSS. If you want to use the glob function but require a specific order, a naming convention like `_01-mixin.scss` could work as well.

If you are a Rails developer, this feature is made available to you via the sass-rails Gem. If you are not using Rails, Chris Eppstein has made this feature available to all users via a plug-in Ruby Gem, Sass globbing.

Configurable theme option

An advanced concept of using a Sass structure like this is using a `_config.scss` file to manage the smart defaults for your UI. Using this technique will help to keep all your UI configuration options easily accessible and manageable, especially when you are using extended Sass libraries like Zurb's Foundation or Toadstool w/Stipe.

```
// We use these as default colors throughout
$primary-color: #008CBA;
$secondary-color: #e7e7e7;
$alert-color: #f04124;
$success-color: #43AC6A;
$warning-color: #f08a24;
$info-color: #a0d3e8;

// We use these to make sure border radius matches.
$global-radius: 3px;
$global-rounded: 1000px;

// We use these to control inset shadow shiny edges and depressions.
$shiny-edge-size: 0 1px 0;
$shiny-edge-color: rgba(#fff, .5);
$shiny-edge-active-color: rgba(#000, .2);
```

It is important that there are no presentational CSS rules in the `_config.scss` file. Typically you will include it at the head of your primary Sass manifest file, such as `application.scss`. Depending on how your architecture progresses, there may be times when you need to import your `_config.scss` file again in another module. As long as you keep any CSS rules out of this document, there is nothing wrong with this practice.

Module partials

Module partials is where we get to work. Here we write Sass rules that will create your UI foundational layer. `_buttons.scss`, `_forms.scss`, `_global-design.scss`, `_reset.scs`s and `_typography.scss` all contain Sass rules that will process into CSS. While they will import other *partials*, *mixins* and *placeholder selector* rules, it is important to remember that these files are engineered only to output CSS.

Taking *buttons* as an example; between gradients, `:hover` and `:active` states, one could go a little mad over the complexities in styling. It is important to keep your Sass logic out of these files and focus purely on the rules that will produce CSS for your selector.

Using a Compass Extension to quickly engineer a button is a great example. In our `_buttons.scss` partial we would only have code like so:

```
button, a.button {
  @include button($button-color);
}
```

Keeping functional Sass separate from presentational Sass is important in order to maintain readability, search-ability and scalability of your code. Patterns like placing mixins in the same file as presentational Sass leads to overly complex files to scan and opportunities for accidental pollution of your processed CSS.

Custom mixins, placeholder selectors and custom function organization

Custom code for a project is the one area where I see the most issues with file management. Polarizing concepts, like keeping things global or local, paralyze many developers. They want to keep the code as accessible as possible, but inevitably end up creating functional Sass that is specific to a type of UI or module. So instead of ignoring these issues, I recommend embracing the concepts that allow developers to create abstract concepts while maintaining a code organization that makes sense.

Using our button example again, let's say that you need to roll your own from scratch. In the file structure there is a corresponding `buttons/` directory where you will keep your `_mixin.scss`, `_extend.scss` and custom function files. This solution will keep your presentational Sass clean and readable, while placing your functional Sass in a directory that is modular and is easy to find.

Modules and UI patterns

Now that we have established the architecture for our UI foundation, it is time to start assembling some modules. In essence, modular Sass is an assembly of foundational elements with only enough additional presentational Sass to hold it together. The use of elemental styles to build a module is strongly encouraged; while defining new elements in the scope of building a module is strongly discouraged.

A module's Sass is exclusive to a particular interaction of the application. Modules will come in all shapes and sizes, while larger modules may also consist of smaller modules or UI patterns.

UI patterns are subject to personal interpretation. In practice when engineering modules, from one to the next, UI patterns will emerge. It is practical to try and encapsulate these smaller patterns for reuse, but I don't lose sleep over them.

Module

A module is a singular functional deliverable and will be unchanged in form, functionality and content. Examples are site header, main navigation and footer.

One could argue that a module is engineered as a '*plug-n-play*' element. Taking the main navigation for example, there would be no good reason why you would want to re-purpose this UI element and functionality in another module? That would be very confusing to your users. Your functional code, your Sass and your application should represent this as a singular modular object.

UI Patterns

UI Patterns, on the other hand, are representations of assembled UI elements. Dialog boxes are a great example. These patterns consist of design elements such as, typography, arrangement, color, border and spacing. These patterns can be re-purposed again and again throughout the application/site. But the content and functionality of this pattern are subject to redefinition based on use.

The module file structure

Module and UI patterns are directories unto themselves. An exploded module directory may look like the following:

```
|- sass/
|--- modules/
|----- registration/
|------- _extends.scss
|------- _functions.scss
|------- _mixin.scss
|------- _module_registration.scss
|------- _module_personal-info.scss
|----- purchase/
|------- _extends.scss
|------- _functions.scss
|------- _mixin.scss
|------- _module_summary.scss
|------- _module_purchase.scss
```

The idea here is that while engineering modules you may need to create complex functional Sass that is exclusive to a module. While I strongly encourage making UI logic as abstract as possible and available to the whole app, this process discourages the practice of creating *junk-drawers*.

Keeping these logic files close to the actual use-case helps maintain clean organization of your code. As shown in the example above, a primary module may consist of smaller modules. As a naming convention, I will name the primary module Sass file after the name of the directory prefixed with `module_`. For example: `_module_registration.scss`. Any sub-modules in this directory will simply be named by the purpose in which it serves, `_module_personal-info.scss` for example.

Sub-modules of a UI in many cases will contain similar characteristics. This close relationship between a module's functional Sass and it's presentational Sass also serves code reuse and management purposes. Take for example the use of a *silent placeholder*. In the extends.scss file you may engineer a reusable UI module that utilizes several variables. In the corresponding presentational Sass module file you can @extend this UI while resetting some of the default variables.

All modules should be name-spaced by the semantic name of the module itself. `.registration {}` or `.purchase {}` for example. If the sub-module is exclusive to the primary module then it would extend the name like so, `.purchase_summary {}`.

Keep in mind that at the level we are working at it is scoped to the module itself. Keeping the selectors shallow will encourage reuse throughout the application without causing additional engineering. If you find yourself engineering complex UIs within the module, this may be an opportunity to abstract into a mixin or silent placeholder selector.

Assemble the layout

Prior to this process, I started developing at the layout level. Abstracting concepts like modules and elements were extremely difficult to do and typically overlooked. By completing my UI development journey with the layout, we get to take advantage of all of the hard work done so far. Our layout Sass files should contain no more information then is needed to assemble a series of modules and elements. Imagine a sketch with gray boxes in the view, this is the document that creates that structure. Elements are never defined and modules are never engineered here.

I never advocate for sub-directories per layout, as things should never get that complex. At this level, assembling the layout should be taking 100% advantage of the elements and modules already engineered. If you find yourself involved in more complex development at this phase, I would argue that you need to review your work before engaging in more complex levels of code. An ideal file structure would look similar to the following.

```
|- sass/
|--- layouts/
|----- _home-layout.scss
|----- _marketing-layout.scss
|----- _search-results-layout.scss
|----- _order-summary-layout.scss
...
```

Typically I will name a layout Sass file after the semantic meaning of the view. In an MVC app, a great convention is either use the name of the `controller` or the `layout` template file and append the word `-container` or `-layout`. An example would be `_sessions-container.scss` or `_sessions-layout.scss`.

To keep things simple, I would then scope all the presentational Sass in a document by the same name, `.sessions-layout {}` for example. Using this class name can be achieved by dynamically adding the class to the `<body>` tag when the view renders or creating multiple layout templates with a static class applied. Use whatever works for you.

Our #1 goal with layout Sass files is to place control of the template UI into the hands of the CSS itself rather then depending on presentational classes in our markup. This becomes even more important when considering mobile/content first and responsive web design strategies.

Handy tools

So far we have learned about the history behind Sass, `.sass` files vs. `.scss` files, and a bit of a refresher on OOCSS. Now you are ready to get your hands dirty and writing some sass code. The code you'll see in this chapter is not too different from writing any CSS project you have seen; we have to evaluate the design, all the elements involved, and the modules to be created. We will continue with our sample project, incorporating the design elements using Sass. We will also look at some of the existing code and evaluate how we will optimize it further using Sass.

In this chapter, while continuing our project, we will learn about variables, scoping, `!default` flag, and how to setup a `_config.scss` file. We will also cover an introductory example of mixins and some of the guidelines for creating a mixin. These are some of the infrastructure that we need to setup in order to further expand on the design and build our stylesheet, so let's get started by looking at the needed variables.

Core data types

Sass, like many languages, consist of a series of core types. A data type, or simply type, is a classification for identifying one of various types of data from which a language can operate with. Data types typically consist of real, integer or Boolean values that determine the possible outputs for that type. Sass' data types are:

- numbers (e.g. 1.2, 13, 10px)
- strings of text, with and without quotes (e.g. "foo", 'bar', baz)
- colors (e.g. blue, #04a3f9, rgba(255, 0, 0, 0.5))
- booleans (e.g. true, false)
- nulls (e.g. null)
- lists of values, separated by spaces or commas (e.g. 1.5em 1em 0 2em, Helvetica, Arial, sans-serif)
- maps from one value to another (e.g. (key1: value1, key2: value2))

Numbers

Sass numbers consist of floating point values, integers, and values with units. In the following example the parentheses are being used as a separator to maintain the order of operations so that the floating point value is divided by integer and then multiplied by the value with the unit.

By the way, multiplying a floating point or integer by a value of `1` with a unit is a very common pattern for adding a unit to a unit-less number.

SCSS

```
block {
  width: (9.9 / 3) * 1em;
}
```

CSS

```
block {
  width: 3.3em;
}
```

Strings

CSS specifies two kinds of strings: those with quotes, such as double quotes `"Lucida Grande"` or single quotes `'http://sass-lang.com'`, and those without quotes, such as `sans-serif` or `bold`. SassScript recognizes all kinds. In general if one kind of string is used in the Sass document, that kind of string will be used in the resulting CSS.

String operations

String operations allow you to build phrases on the fly by concatenating a series of variables that consist of strings.

SCSS

```
$foo: 'foo';
$bar: bar;
```

```
block {
  content: $foo + $bar;
}
```

CSS

```
block {
  content: "foobar";
}
```

Interpolation

Consider interpolation as a 'replacement' method. In some cases the value of a variable cannot be used when creating different types of strings in the processed CSS. A very common use case is when you are using the value of a variable to create a CSS attribute as illustrated in the following.

SCSS

```
@mixin firefox-message($selector, $value) {
  body.firefox #{$selector}:before {
    content: "Hi, Firefox users!";
    content: "I ate #{5 + $value} pies!";
  }
}

@include firefox-message(".header", 10);
```

CSS

```
body.firefox .header:before {
  content: "Hi, Firefox users!";
  content: "I ate 15 pies!";
}
```

Colors

If it can be written in CSS, it will work in Sass. In addition, Sass supports a wide range of Color Operators. All arithmetic operations are supported for color values, where they work piecewise. Meaning operation are performed on the red, green, and blue components in turn. For example:

SCSS

```
div {
  color: #010203 + #040506;
  background: #010203 * 2;
  border: rgba(255, 0, 0, 0.75) + rgba(0, 255, 0, 0.75);
  background-color: blue + red;
}
```

CSS

```
div {
  color: #050709;
```

```
  background: #020406;
  border: rgba(255, 255, 0, 0.75);
  background-color: magenta;
}
```

Sass also has support for a large range of names colors that are indistinguishable from typical strings. See the full list of colors as written in the code-base itself.

Boolean

SassScript supports `and`, `or`, and `not` operators for boolean values. When using either `and` or `or` evaluators you are required to wrap the evaluation in parentheses `()` as illustrated in the examples below.

SCSS

```
$alpha: red;
$beta: green;
$charlie: yellow;
$delta: red;

.block {
  @if $alpha != $beta {
    content: 'winner!';
  } @else {
    content: 'loser!';
  }
}

.block {
  @if $alpha == ($beta or $charlie) {
    content: 'winner!';
  } @else {
    content: 'loser!';
  }
}

.block {
  @if $alpha == ($beta and $delta) {
    content: 'winner!';
  } @else {
    content: 'loser!';
  }
}
```

Null

Null values are treated as empty strings for string interpolation. The best use case for `null` is when you need to define a value, but don't have a specific value to be used. When Sass encounters an empty string, it's default functionality is to NOT print out the CSS. In this example, if we have an empty selector, Sass will not output any CSS.

```
.selector {

}
```

But you can't set an empty string to a variable. In this example, a common practice of using empty quotes, Sass will output that as a value.

```
$color: ' ';
```

```
.block {
  color: $color;
}
```

The output CSS will be:

```
.block {
  color: " ";
}
```

But if we use a `null` value:

```
$color: null;

.block {
  border: 1px solid $color;
  color: $color;
}
```

We would expect the following:

```
.block {
  border: 1px solid;
}
```

Lists

Lists are how Sass represents the values of CSS declarations like `margin: 10px 15px 0 0` or `font-face: Helvetica, Arial, sans-serif`. Lists are just a series of other values, separated by either spaces, commas `,`, quotes `" "` or parentheses `()`. In fact, individual values count as lists, too: they're just lists with one item.

There are some rules you need to be aware of when building a list. Due to all the operator types, you can actually build out a list in the following way, this will actually create 10 individual strings in the list.

SCSS

```
$string: this is a string of words "more words" then even 'more words';
```

While it is common to see lists separated with commas, e.g. `$var: red, blue, yellow`, since spaces are an operator, this is pretty redundant. The following examples produce the exact same number of items in the list.

```
$string: this is a string of words more words then even more words;

$string: this, is, a, string, of, words, more, words, then, even, more, words;
```

Parentheses `()` on the other had, you can use as separators, but these are most commonly used when you want to group different styles of values within a given list item. Keep in mind that strings within parentheses need to be in quotes. The following example will represent 3 items within the list.

```
$string: (12 "foo" 8em) (9 "bar" 1px) (66 "baz" 100%);
```

Again, you will commonly see examples where there are commas between the parentheses as illustrated below.

```
$string: (12 "foo" 8em), (9 "bar" 1px), (66 "baz" 100%);
```

While this works, it is unnecessary in this example because of the spaces. And to even point out, since we are using parentheses to group the items in the list, we don't even need the spaces, but it does help for readability.

Alone, lists don't provide any output for CSS. In most cases you will see lists used in conjunction with loops or cherry picking values using the `nth()` function. The `nth()` function provides a method to seperate a list into its assembled values. The following is an example of `nth()` functiona usage:

SCSS

```
// Browser prefixes
// -------------------------
$browser: -moz- -webkit- -o- -ms-;

.border-image {
  #{nth($browser, 2)}border-image: url("/files/4127/border.png") 30 30 repeat;
  #{nth($browser, 3)}border-image: url("/files/4127/border.png") 30 30 repeat;
  border-image: url("/files/4127/border.png") 30 30 repeat;
}
```

CSS

```
.border-image {
  -webkit-border-image: url("/files/4127/border.png") 30 30 repeat;
  -o-border-image: url("/files/4127/border.png") 30 30 repeat;
  border-image: url("/files/4127/border.png") 30 30 repeat;
}
```

Maps

Maps represent an association between keys and values, where keys are used to look up values. They make it easy to collect values into named groups and access those groups dynamically.

Note the syntax, in SCSS there is a semi-colon that follows the declaration as this is a single Sass statement. List-maps have no direct parallel in CSS, although they're syntactically similar to media query expressions.

```
$name-space: (key: value, key: value);
```

Making use of List-Maps typically is done with the `map-get` function. When using List-Maps, the variable is commonly referred to as the name-space. In this example I will create the `$input` name-space and nest some `key:value` pairs within it.

```
$input-disabled-color: #333 !default;

$input: (
  disabled-background lighten($input-disabled-color, 75%),
  disabled-border lighten($input-disabled-color, 50%),
  disabled-text lighten($input-disabled-color, 50%)
);
```

To extract these values we will use `map-get` function and pass in the name-space and the key from which we intend to get it's value.

```
input[disabled] {
  background-color: map-get($input, disabled-background);
  border-color: map-get($input, disabled-border);
  color: map-get($input, disabled-text);
}
```

In our output CSS I would expect something like the following:

```
input[disabled] {
  background-color: #f2f2f2;
  border-color: #b3b3b3;
  color: #b3b3b3;
}
```

As you can see, using list-maps can drastically change how you manage and maintain a series of related variables and their values.

Number Operations

SassScript supports the standard arithmetic operations on numbers (addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`). Sass math functions preserve units during arithmetic operations.

SCSS

```
p {
  $width: 1000px;

  font: 10px/8px;             // Plain CSS, no division
  width: $width/2;            // Uses a variable, does division
  width: round(1.5)/2;        // Uses a function, does division
  height: (500px/2);          // Uses parentheses, does division
  margin-left: 5px + 8px/2px; // Uses +, does division
}
```

CSS

```
p {
  font: 10px/8px;
  width: 500px;
  width: 1;
  height: 250px;
  margin-left: 9px;
}
```

Setting Variables

When writing any CSS, look at some of the common values like the colors, fonts, font sizes, and the grid layout. These basic design elements need to be incorporated in your styelsheet. One of the ways Sass can greatly help is with variables. With Sass these common values can be abstracted and placed in associated variables which can be referenced when needed.

> Variables allow you to name CSS values that you use repeatedly and then refer to them by name rather than repeating the value over and over. You can also name values you only use once in order to make it more clear what they're for.
>
> Sass and Compass in Action page 32

The major advantage of using variables in Sass is that now you have a single point of reference which allows for better maintainability and code extensibility. If there is any change to the CSS style, in theory changing the variable should handle the change (I mention in theory because sometimes practical issues provide exceptions to this rule). Let's look at the following variable declarations:

```
// Create primary color palette for the site

//Primary colors palette
// -------------------------
$blue:           #3481CF !default;
$white:          #FFFFFF !default;
$black:          #000000 !default;
$gray:           #7F7F7F !default;

// Create  derivative color palette from the primary color palette

//Derivative colors
// -------------------------
$dark-gray:      darken($gray, 23.14%) !default; //#444
$darker-gray:    darken($dark-gray, 6.6667%) !default; //#333
$darkest-gray:   darken($dark-gray, 17.25%) !default; //#181818

// Set a semantic alias to the color variable based on the dark gray color that was instantiated from the derivative co

//Font information
// -------------------------
$font-color:    $darker-gray;
$anchor-color:  $darker-gray;

// Additionally, primary, secondary, tertiary, and any number of colors can be created

//Color use palette
// -------------------------
$primary-color: $blue;
$secondary-color: $black;
$tertiary-color: $white;

// When creating the base CSS, I will assign the font color alias variable that was instantiated earlier

p {
  color: $font-color;
}

a {
  color: $anchor-color;
  &:hover {
    color: darken($anchor-color, 20%);
  }
}
```

As illustrated in this this example, we are able to instantiate variables using values which, in this case, are hexadecimal colors. We can also instantiate additional variables using existing variables, this is referred to as aliasing. The functions used for the `$dark-gray` variable are existing color functions provided by Sass. We will cover functions in more detail in chapter 6, however in the meantime lets surmise that this function will take the existing gray color and darken it by 23.14%.

It is important to keep in mind that we now have created a maintainable and easily scaleable color scheme for our site. We will be referencing this set of colors throughout our Sass stylesheets. In the future, if there is a requirement for the color scheme to be changed, the change can be done at the point of instantiation in the `_config.scss` file. For example, if the requirement is that the the grays should be a shade darker for example instead of `#7f7f7f` we want `#7b7b7b`, we can change the `$gray` variable which will trickle down to all referenced variables in the stylesheet.

We have now created a file that will contain the basic variables for our site. A maintainable and scalable file for the stylesheet and site design. We will discuss this in more detail in using a `_config` file section.

When discussing variables in any programming language, it's important to understand how variables are scoped. Let's take a look at how Sass handles the scoping of variables.

Variable Scoping

In any programming language, scoping should be considered when setting variables. In Sass, all variables declared outside of a mixin or function will have a global scope and can be referenced in any Sass selector that uses the variable.

```
$text-color: blue;

html {
  font-family: Arial, sans-serfi;
  font-size: 1em;
  color: $text-color;
}
```

Keeping true to the Cascading part of CSS, currently if the value of the variable is changed, all further reference to the variable will be updated to the new value. This sounds logical but consider the following example:

```
// I instantiated the $text-color variable to Blue
$text-color: blue;

// Here, the intent was to change the color for the .error style
.error {
  $text-color: red;
  color: $text-color;
}

// Following the cascade, in .normal-text, I want Blue, but get Red.
.normal-text {
  color: $text-color;
}
```

The above Sass will compile to the following CSS:

```
.error {
  color: red;
}

.normal-text {
  color: red;
}
```

As you can see, the variable `$text-color` has a global scope and it is set to `blue`. However when I changed the `$text-color` variable to `red`, you will see that all further instances of `$text-color` variable will be `red`. This is common pitfall among novice Sass users. The best way to prevent this pitfall is to follow these guidelines:

- Always set global variables and do not reset them throughout the stylesheet
- Make use of `!default` flag

Arguments within Mixins and Functions

When setting variables in mixins or functions, keep the above scoping scenario in mind. If there is a variable that needs to be scoped within a mixin or function, declare it within the required scope. Consider the following:

```
@mixin add-border($border-position: all, $border-size: 1px,
  $border-pattern: solid, $border-color: black) {
```

```
    $border-position-all: all;

    @if $border-position == $border-position-all {
      border: $border-size $border-pattern $border-color;
    }
    @else {
      border-#{$border-position}: $border-size
      $border-pattern $border-color;
    }
  }

  block {
    @include add-border();
  }
```

In this example, we set a local variable `border-position-all: all`. We could also write the mixin as such:

```
$border-position-all: all !default;

@mixin add-border(
  $border-position: $border-position-all, $border-size: 1px,
  $border-pattern: solid, $border-color: black) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}
```

Setting the `border-position-all` as a global variable, it can now be referenced throughout the application. The other difference here is that the `$border-position-all` variable uses the `!default` flag.

As you can looking at our `$border-postion` variable declaration, we have used a flag called `!default`. There are two flags that can be set when declaring a variable:

- `!default`
- `!global`

In the next section, we will take a closer look at how these flags work and how we can take advantage of them.

The !default and !global flags

In Sass using variables is a keystone in the language. In fact it was one of the first selling points when I was introduced, "Did you know you could set variables for colors?" Little did I know what that statement really meant.

A few years later, variables in Sass continue to be as powerful if not more powerful. Thus is the case of using `!default` and `!global` flags when setting variable precedence and scope.

In this section we will discuss best practices for using these flags and how to maintain scope with variables.

The !default flag

Placing `!default` at the end of a variable declaration will have the following effect:

- If the variable already has an assignment, it will not be re-assigned
- Variables with null value will be considered unassigned and will be assigned with `!default`

The `!default` flag is extremely useful when creating plug-in type code and with mixins. Let's look at this `text-color` example mixin:

```
// Variable for $text-color is set to Blue

$text-color: blue;

@mixin text-color {
// Variable is only set to Red if it has not been set beforehand
  $text-color: red !default;
  color: $text-color;
}

.error {
  // Include mixin with !default color set
  @include text-color;
}

.normal-text {
  @include text-color;
}
```

Notice the `!default` flag at the end of the `text-color` variable inside the mixin? This allows the global variable of `blue` to override the value of `red`. Therefore the Sass will compile to the following:

```
.error {
  color: blue;
}

.normal-text {
  color: blue;
}
```

If we remove the global `$text-color` variable, Sass will make use of the `!default` set variable inside the mixin.

```
.error {
  color: red;
}

.normal-text {
```

```
  color: red;
}
```

It is important to remember that if the `$text-color` variable inside the mixin DID NOT have the `!default` flag, this variable's value will currently override any previously set value due to the cascade. I say currently because this is a deprecated concept.

As illustrated, a more practical use of the `!default` flag is within mixins along with implementing a modular Sass architecture. Let's move our mixin, `add-border`, to a module file which I will call `_decoration-mixins.scss`:

```
$border-position-all: all !default;
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: $black !default;

@mixin add-border($border-position: $border-position-all,
  $border-size: $border-default-size,
  $border-pattern: $border-default-pattern,
  $border-color: $border-default-color) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}
```

To make use of this new tool, we will use Sass' `@import` rule to import into the website's stylesheet. Once imported, in the `_config.scss` file we can override any of the values set in this mixin, if necessary:

```
$border-default-pattern: dotted;
$border-default-color: lighten($gray, 25%);

@import "border";

.block-border {
  @include add-border($border-size: 2px);
}
```

This Sass will compile to:

```
.block-radius {
  border: 2px dotted #bfbfbf;
}
```

As you can see from these examples, a variable with the `!default` flag will only be set if the said variable has not been instantiated beforehand (or it is null). This is a very useful feature for modular (or OOCSS) design of our CSS. It is best practice to place the majority of the variables in a file for better maintenance and accessibility. The exception to this practice is when the variable is used only within a modular segment of the architecture.

In the above mixin, the variable `$border-position-all: all !default;` is instantiated in the mixin file. All other variables can be instantiated in a single file as in the `_config.scss` shown in furhter detail in the _config.scss section.

The !global flag

Sass version 3.3 included several important additions, one them being the `!global` flag. According to the release notes, the purpose of the `!global` flag is:

> "As part of a migration to cleaner variable semantics, assigning to global variables in a local context by default is deprecated. If there's a global variable named `$color` and you write `color: blue` within a CSS rule, Sass will now print a warning; in the future, it will create a new local variable named `$color`. You may now explicitly assign to global variables using the `!global` flag; for example, `color: blue !global` will always assign to the global `color` variable."

Let's expand on this a bit more. As discussed on the section on variable scoping, all variables declared outside of a mixin or a function will have a global scope and can be referenced in any Sass selector that uses the variable.

For example, Let's look at our original example from the variable scoping section:

```
// I instantiated the $text-color variable to Blue
$text-color: blue;

// Here, the intent was to change the color for the .error style
.error {
  $text-color: red;
  color: $text-color;
}

// Following the cascade, in .normal-text, I want Blue, but get Red.
.normal-text {
  color: $text-color;
}
```

Currently, using variables in Sass where there is a value set in the global space and then one set within the context of a selector, the value of the variable set within the selector will bleed into the global name space. Running the above Sass in the terminal we can see that this concept is deprecated.

```
DEPRECATION WARNING on line 6 of style.scss:
Assigning to global variable "$text-color" by default is deprecated.
In future versions of Sass, this will create a new local variable.
If you want to assign to the global variable, use "$text-color: red !global" instead.
Note that this will be incompatible with Sass 3.2.

.error {
  color: red; }

.normal-text {
  color: red; }
```

The intention of this warning is to state that changes will be coming in future versions of Sass. Sass will know that there is a global `$text-color` and a scoped `$text-color` within a selector. The scoped `$text-color` will NOT bleed into the global space and alter the value of any variables that follow unless you add the `!global` flag.

The following example is in speculation of future functionality.

```
$text-color: blue;

.error {
  $text-color: red; // This is now a new local scoped variable
  color: $text-color;
}

.normal-text {
  color: $text-color;
}
```

It is assumed that the above Sass will output the following CSS. This is not yet implemented, that is why there is a DPERECATION warning.

```
.error {
  color: red;
}

.normal-text {
  color: blue;
}
```

If you do want to modify a global variable within a local scope, use the `!global` flag. The above example can be re-written as such so that we can modify the global `$text-color` value:

```
$text-color: blue;

.error {
  $text-color: red;
  color: $text-color;
  $text-color: green !global;
}

.normal-text {
  color: $text-color;
}
```

Which will compile to the following (WITHOUT the deprecation warning):

```
.error {
  color: red;
}

.normal-text {
  color: green;
}
```

To see the true implementation of `!global`, let's examine how local and scoped variables work within mixins and functions.

!global flag + variables/arguments in mixins and functions

For clarification, variables used within mixins or functions are NEVER global. For example, if we declare a global variable `$var` with a value and then include local variable with tha same name `$var` into a mixin, the value will not follow:

```
$var: yellow;

@mixin foo($var) {
  color: $var;
}

.block {
  @include foo;
}
```

The above Sass will throw an error during compilation:

```
Syntax error: Mixin foo is missing argument $var.
        on line 8 of test.scss, in `foo'
        from line 8 of test.scss
```

In order to get this value to pass from the global var to the mixin we need to do this:

```
$default-var: yellow;

@mixin foo($var: $default-var) {
  color: $var;
}

.block {
  @include foo;
}

//or it can written as such
$default-var: yellow;

@mixin foo($var) {
  color: $var;
}

.block {
  @include foo($default-var);
}
```

The above Sass will compile to:

```
.block {
  color: yellow;
}
```

As mentioned before, all variables declared within a mixin or function have a local scope and will not affect the global variables. So if within the mixin we redefine the value of `$var`, this will effect the value of the following `$var`, but this will not bleed out into the global space because all the variables in a mixin or function are scoped locally. If I add `$var` with a new value within a selector with `!global` flag, this WILL bleed into the global space.

```
$var: yellow;

@mixin foo($var: $var) {
  global-color: $var;
  $var: purple; // this is trapped within the mixin and has a local scope
  scoped-color: $var;
}

.block {
  @include foo;
  $var: lime !global; // added to global scope
}

block {
  global-color: $var;
}
```

Running this in the command line, we will get the following css:

```
.block {
  global-color: yellow;
  scoped-color: purple;
}

block {
  global-color: lime;
}
```

If I wanted to get PURPLE to be in the global space when the mixin is used, we can do that by adding the `!global` flag

```
$var: yellow;

@mixin foo($var: $var) {
  global-color: $var; // local variable color coming from global variable passed into the mixin
  $var: purple !global; //changing the global variable within the local context of a mixin
  scoped-color: $var;
}

.block {
  @include foo;
}

.block {
  global-color: $var;
}
```

Doing so also changes the way the variable's value is used, notice how the `scoped-color` is not effected by the global setting as illustrated in this output CSS.

```
.block {
  global-color: yellow;
  scoped-color: yellow;
}

.block {
  global-color: purple;
}
```

This is another powerful aspect of Sass which will allow us to change a global variable based on the processes ran within a function or variables set within a mixin.

Using a _config.scss file

When working with variables, creating `_config.scss` files are definitely considered best practice. Typically in projects I will have a `_config.scss` file at the root of the Sass directory and it's not entirely uncommon to have one in close relation to a module UI if the complexity requires such a tool.

Looking back at File Management, specifically the Configurable theme option section, we talk about this in the context of a project for use with theming, but `_config.scss` files go way past simple theming. Consider `_config.scss` files as the operations center of your UI architecture.

Root _config.scss file

The following is an example `_config.scss` file that would appear at the root of the Sass directory. A collection of variables that are made available to the Sass at the time of processing.

```
// URL variable
//------------------------------
$base-img-url: '/images';

// Primary colors
//------------------------------
$blue:                  #3481CF;
$white:                 #FFFFFF;
$black:                 #000000;
$gray:                  #7F7F7F;

// Derivative colors
//------------------------------
$dark-gray:             darken($gray, 23.14%); //#444
$darker-gray:           darken($dark-gray, 6.6667%); //#333
$darkest-gray:          darken($dark-gray, 17.25%); //#181818

// Color palette
//------------------------------
$primary-color: $blue;
$secondary-color: $black;
$tertiary-color: $white;

// Font information
//------------------------------
$header-font-family: "Georgia", "Times New Roman", serif;
$default-font-family: "HelveticaNeue", "Helvetica Neue", Helvetica,
Arial, sans-serif;
$default-browser-size: 16;
$default-font-size: 14px;
$font-color:    $darker-gray;
$anchor-color: $darker-gray;

// Z-index variable
//------------------------------
$starting-zindex: 1000;
$zindex-modal-backdrop: $starting-zindex * 3;
$zindex-modal: $zindex-modal-backdrop + 1;

// Responsive
//------------------------------
$small-screen-min-width: 320px;
$small-screen-max-width: 568px;
$medium-screen-min-width: 768px;
$medium-screen-max-width: 1024px;
$large-screen-min-width: 1824px;
```

Local _config.scss file

Another example would be a `_config.scss` file that would be local to a module, as in this example, a Button mixin.

```
// Default values - edit in `_config.scss` file
// ------------------------------------------------------------
$button-color:  $button-color !default;
$button-text-color: $white !default;
$button-line-height: 32 !default;
$button-border-radius: 3 !default;
$button-padding: 20 !default;
$button-font-size: 18 !default;
$button-weight: bold !default;
$button-text-shadow: true !default;
$button-box-shadow: true !default;
```

Notice the use of the `!default` flag. This process of defining defaults in a `_config.scss` that would probably appear in the same directory as the button mixins themselves allows for defaults to be set, but are easily over-written in the primary root `_config.scss` file.

For more about `!default` functionality, please read The !default and !global flags.

Mixins

So far we have covered detailed information about variables. One of the major consumers of variables are mixins. Let's a quick introductory but detailed view of mixins.

While engineering your project you will begin to discover a number of repeated CSS patterns in your code. This is where mixins become very handy. While variables allow you to re-use values, mixins allow you to re-use blocks of CSS and Sass. Mixins can be an assortment of CSS rules and Sass logic that can be used repeated throughout the site. It is a complex and dynamic set of styles, similar to variables, that will allow for a single point of change which will reverberate through all the inclusion points. As the repeated patterns become apparent to you, writing the mixin can become self evident. While some of these patterns may be a direct 1:1 copy of the code, for example:

```
@mixin hand-cursor {
    cursor: hand;
    cursor: pointer;
}
```

The above mixin is an example of a direct mixin and when used will change the cursor to a hand or pointer on the desired element. As you can see, this mixin does not have any variables and is a very simple example of a mixin.

While other mixins deal with similar patterns with subtle differences that will be handled by the parameters and the dynamic structure of a mixin. Let's start looking into mixins in more detail.

Anatomy of a mixin

A mixin is composed of the following structure:

```
// A mixin always begins with @mixin keyword. Parameter(s) for the mixin is not required but it will make the mixin mor
@mixin [mixin-name] ([mixin-parameters...]) {
    // Logic along with different functions and evaluations can also add a more dynamic dimension to the mixin. It can
    [Mixin logic/Sass functions/CSS rules]
    // The end result of the mixin is a return of property and value that will output to the stylesheet
    [property]:[value];
}
```

The purpose of a mixin is to create a series of functions that will allow the user to create a robust style based on different circumstances or criteria. Let's examine the `add-border` mixin.

```
@mixin add-border(
    $border-position: all,
    $border-size: 1px,
    $border-pattern: solid,
    $border-color: $black) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}
```

In the case of this mixin, we are trying to add a border to any element that requires it. Our default border is one that will add a 1 pixel solid black border in all directions (top, right, bottom, and left) to an element. However there are times that we will desire a border only in a certain direction/position or a different border style or border color and this is where a mixin can help us, in particular the `add-border` mixin.

To include a mixin within a style or element, using the `@include` keyword and adding it to an element is the only step needed.

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    @include add-border;
}
```

The above code will add the default border, a 1 pixel solid black border in all directions (top, right, bottom, and left), to the `block-border` style.

One thing to keep in mind, When placing your @include into a CSS selector, it is considered best practice to include the mixin directly after the parent selector as seen in the following example.

Placing the mixin first serves a couple of purposes:

- Its consistent placement makes your code easier to scan.
- It takes advantage of the cascade.

Lets' look at how to use the arguments within a mixin to give us the most advantage and robustness.

Set default values for mixins

When using mixins that require arguments, it is a best practice to always specify a default value for each argument. There are a number of good reasons as to why you should do this. The leading reason is to reduce duplication when including this mixin into a selector.

It is quite probable that the mixin you are using will reuse the same values a number of times with slight variations. Having to always search through previous code to remember what common values you applied last is time consuming and prone to mistakes. Using a good default strategy will reduce time and code error. This is why I have written the `add-border` mixin as such:

```
@mixin add-border(
    $border-position: all,
    $border-size: 1px,
    $border-pattern: solid,
    $border-color: $black) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}
```

When we include the mixin into our CSS as such:

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    @include add-border;
}
```

The above example includes the mixin using the default values but what if we wanted a border that was 2 pixels dotted and gray in all directions? How would we include our mixin to incorporate the different parameters? Here is an example:

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    // This will list all the parameters and pass them to the mixin
    @include add-border(all, 2px, dotted, $gray);
}

// or

.block-border {
    // Although this is a more verbose example, it does allow for other developers to be able to better understand the
```

```
    @include add-border(
        $border-position: all,
        $border-size: 2px,
        $border-pattern: dotted,
        $border-color: $gray);
}
```

Another scenario is when we only require a change to some of the parameters and not all of them. For example, if we need a top border that is a light gray color, this is how we would implement it:

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    @include add-border(all, 2px, dotted, $gray);
}

.non-semantic-class-name-top-border-light-gray {
    @include add-border($border-position: top,
                        $border-color: lighten($gray, 30%));
}
```

As you can see, we only pass the parameters that we want changed and all the other parameters are driven from the default parameters in the mixin. The above sass file will compile to css as such:

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    border: 2px dotted #7F7F7F;
}

.top-border-light-gray {
    border-top: 1px solid #CCCCCC;
}
```

Setting global default argument variables

Setting default argument values within the mixin is a good idea, but in our example the arguments themselves are non-variable values, and any changes to them down the road would require searching through our Sass to update the argument values. Suddenly we feel like we are back to the old days of CSS again.

A preferred method of managing argument default values is to set a global default variable. Leveraging Sass' `!default` flag will allow you to set a default value to a variable associated to an argument within a mixin that can be over-ridden at a global level.

Let's refactor our previous example, except this time I will replace all the hard coded default argument values with more flexible variable names. It is important to note that setting default values to an argument always require a key:value pair. Variables are scoped within a mixin, so setting a global variable of `$padding` will not work. While we could set a key:value pair like `$padding: $padding` this is considered a poor practice, especially with something named as generic as `$padding`.

In the following example, you will see how I set a simpler named argument to a mixin as this will be scoped, but we are assigning a global variable that is using a more specific naming convention to declare use.

```
$border-position-all: all !default;
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: $black !default;

@mixin add-border(
    $border-position: $border-position-all,
    $border-size: $border-default-size,
    $border-pattern: $border-default-pattern,
    $border-color: $border-default-color) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}
```

To set the default values to these newly created variables, typically in close relation to the mixin itself, we will set the values using the `!default` flag as illustrated in the following example.

Remember that global variables follow the rules of the cascade? Meaning, as a value is reset to a global variable within the document, each time that variable is referenced again thereafter, that new value will be used. This is not true for global variables that are using the `!default` flag.

In the following example we will see that by setting a global value to the variable `$border-default-color` higher up in the document, the `!default` value will be over-ridden, essentially breaking the pattern of the cascade.

```
// Global variable
$border-default-color: $dark-gray;

// !default values assigned mixin specified variables
$border-position-all: all !default;
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: $black !default;

// Mixin arguments set to specified variables as defaults
@mixin add-border(
```

```
    $border-position: $border-position-all,
    $border-size: $border-default-size,
    $border-pattern: $border-default-pattern,
    $border-color: $border-default-color) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}


// Selector using the mixin w/o passing in arguments
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    @include add-border;
}
```

Notice in the output CSS how the `$border-default-color` is set to `$dark-gray` which in this example is equal to `#444`.

```
.block {
    width: 100%;
    padding: 5px;
    display: block;
    background-color: transparent;
    overflow: hidden;
    height: auto;
}

.block-border {
    border: 1px solid #444;
}
```

Following this pattern will allows you to use a more module style of creating mixins with default values that are easily over-ridden either by a global scope or local keyword value assignment.

The Turducken mixin: using mixins within a mixin

The power of mixins is not limited to inclusion in the CSS. Just like a variable can be assigned to another variable, a mixin can be included in another mixin allowing for another level of modularity in your stylesheet. As we build a module, for example a button, we are not only building a modular component that can be used as a variety of buttons are created but also we are breaking down our component into smaller components like gradient color, border, padding, and etc...

Let's take a closer look at the assembly of a button. Here are the rules for our `.primary-button` selector:

```
.primary-button {
    font-family: 'merriweather_sanslight', sans-serif;
    -moz-box-shadow:inset 0px 1px 0px 0px #ebf5ff;
    -webkit-box-shadow:inset 0px 1px 0px 0px #ebf5ff;
    box-shadow:inset 0px 1px 0px 0px #ebf5ff;
    background:-webkit-gradient( linear,
        left top,
        left bottom,
        color-stop(0.05, #70d9ff),
        color-stop(1, #3481cf) );
    background:-moz-linear-gradient( center top,
                    #70d9ff 5%,
                    #3481cf 100% );
    background-color: #70d9ff;
    border-radius: 0;
    text-indent: 0;
    border: 1px solid #9cceff;
    display: inline-block;
    color: #ffffff;
    font-size: 28px;
    font-weight: normal;
    font-style: normal;
    line-height: 48px;
    padding-left: 30px;
    padding-right: 30px;
    text-decoration: none;
    text-align: center;
}
```

I have created some mixins in order to handle some of the rules that we use often:

```
$border-position-all: all !default;
$border-default-size: 1px !default;
$border-default-pattern: solid !default;
$border-default-color: $black !default;

// Mixin arguments set to specified variables as defaults
// This is a mixin that will allow the user to add border to an element.
// It is robust enough to allow the user to select a certain side where the border
// should be applied or it can applied on all sides.
@mixin add-border(
    $border-position: $border-position-all,
    $border-size: $border-default-size,
    $border-pattern: $border-default-pattern,
    $border-color: $border-default-color) {

  @if $border-position == $border-position-all {
    border: $border-size $border-pattern $border-color;
  }
  @else {
    border-#{$border-position}: $border-size
    $border-pattern $border-color;
  }
}

// This function was created in order to be able to take a variable argument, in this case the number
// of color stops for the gradient, and return a comma separated list.
```

```
@function linearGradientColors($stop-colors...) {
    $full: false;
    @each $stop-color in $stop-colors{
        @if $full {
            $full: $full + ',' + $stop-color;
        } @else {
            $full: $stop-color;
        }
    }

    $full: unquote($full);

    @return $full;
}

// When creating a function, we place the functional name however for expediency sake, an overloaded
// function is created with a smaller name (usually an acronym of the functions name) so that re-use of the function wo
@function lgc($stop-colors...) {
    @return linearGradientColors($stop-colors...);
}

// This mixin will create a linear gradient using a variable argument for any number of color stops desired.
// The $pos variable allows use to set the gradient line.
@mixin linear-gradient($pos, $stop-colors...) {

  // Detect what type of value exists in $pos
  $pos-type: type-of(nth($pos, 1));

  // If $pos is missing from mixin, reassign vars
  // and add default position
  @if ($pos-type == color) or (nth($pos, 1) == "transparent")  {
    $pos: top; // Default position
    }

  $pos: unquote($pos);

  $full: lgc($stop-colors...);

  // Set the first stop-color as the default fallback color
  $fallback-color: nth(nth($stop-colors, 1), 1);

  background: $fallback-color;
  background: linear-gradient($pos, $full);
}

// This mixin allows us to add box shadows to an element handling the option for an inset box shadow.
@mixin box-shadow ($isInset: false,
                   $hOffset: 0,
                   $vOffset: 0,
                   $blur: 0,
                   $spread: 0,
                   $color: #ccc) {
    @if $isInset {
        box-shadow: inset $hOffset $vOffset $blur $spread $color;
    } @else {
        box-shadow: $hOffset $vOffset $blur $spread $color;
    }
}
```

These mixins are useful for not only the CSS stylesheets but also can be used anywhere within our Sass environment. It can be used in mixins or functions (although the use in functions is uncommon). Let's take a look at our `.primary-button` selector and see how I can improve it using some of the above mixins.

While examine the `primary-button` style, I found the following aspects:

- The button has four different shades of blue associated to it (#ebf5ff, #70d9ff, #3481cf, #9cceff)
- The button has a linear gradient
- The button has a border
- The button has a inset box shadow

Let's optimize the `.primary-button` selector using Sass. First I will add the colors of the button to our `_config.scss` file. However, I will add the primary blue color of the button and all other shades/variations of the color will be used. For example, I could write the colors in the `_config.scss` file as such:

```
... //some config variables here
// Primary colors
// -------------------------
$blue:                  #3481CF !default;
$white:                 #FFFFFF !default;
$black:                 #000000 !default;
$gray:                  #7F7F7F !default;

//button colors
// -------------------------
$primary-button-primary-color: $blue !default;
$primary-button-secondary-color: #70D9FF !default;
$primary-button-tertiary-color: #9cceff !default;
$primary-button-quaternary-color: #ebf5ff !default;
```

Although this is not wrong, it is inefficient and a common mistake many developers new Sass will make. The better approach is to use the color functions that come with Sass to your advantage which will also extend the color scheme so that if the color of the button changes from blue to green, for example, there is only one color to change.

I get the exact colors using Sass' color functions. However, this is also a good time to sit with the designer (if it is not you) and reign in the number of colors used on the site. It's helpful to make the different shades of a color, in this example the blue color of `#3481CF`, an easy off shoot of the primary color being used. In this manner, I can re-write the above as such:

```
... //some config variables here
// Primary colors
// -------------------------
$blue:                  #3481CF !default;
$white:                 #FFFFFF !default;
$black:                 #000000 !default;
$gray:                  #7F7F7F !default;

//button colors
// -------------------------
$primary-button-primary-color: $blue !default;
$primary-button-secondary-color:
    adjust-hue(lighten($blue, 11%), -14deg) !default;
$primary-button-tertiary-color: lighten($blue, 36%) !default;
$primary-button-quaternary-color: lighten($blue, 44%) !default;
```

This approach is advantageous in the following manner:

- Reduced the number of color used on the site
- Single point of change when any change would be necessary

Now that I the colors set, I can start re-writing some of the elements of the `.primary-button` selector. First step, incorporate our linear gradient mixin.

```
.primary-button {
    @include linear-gradient(center top,
        $primary-button-secondary-color 5%,
        $primary-button-primary-color 100%);
    //... remaining styles
}
```

Let's add the box shadow styling:

```
.primary-button {
    @include linear-gradient(center top,
        $primary-button-secondary-color 5%,
        $primary-button-primary-color 100%);
    @include box-shadow (@isInset: true,
                         $vOffset: 1px,
                         $color: $primary-button-quaternary-color);
    //... remaining styles
}
```

Let's add the borders:

```
.primary-button {
    @include linear-gradient(center top,
        $primary-button-secondary-color 5%,
        $primary-button-primary-color 100%);
    @include box-shadow (@isInset: true,
                         $vOffset: 1px,
                         $color: $primary-button-quaternary-color);
    //... remaining styles
}
```

Here is the final style as the mixins are incorporated:

```
.primary-button {
    @include linear-gradient(center top,
        $primary-button-secondary-color 5%,
        $primary-button-primary-color 100%);
    @include box-shadow (@isInset: true,
                         $vOffset: 1px,
                         $color: $primary-button-quaternary-color);
    @include add-border($border-color: $primary-button-tertiary-color);
    border-radius: 0;
    display: inline-block;
    color: #ffffff;
    font: {
        size: 28px;
        weight: normal;
        style: normal;
        family: 'merriweather_sanslight', sans-serif;
    }
    line-height: 48px;
    padding-left: 30px;
    padding-right: 30px;
    text-indent: 0;
    text-decoration: none;
    text-align: center;
}
```

By incorporating the mixins, I have not only re-used code, but also in further implementations of this button, whether it be a different type of button or the implementation of pseudo classes like `:hover`, I can further use and extend this code base.