Lesson 5

Preprocessors

5

In this Lesson

SHARE

In time writing HTML and CSS may feel a bit taxing, requiring a lot of the same tasks to be completed over and over again. Tasks such as closing tags in HTML or repetitively having to looking up hexadecimal color values in CSS.

These different tasks, while commonly small, do add up to quite a bit of inefficiency. Fortunately these, and a handful of other inefficiencies, have been recognized and preprocessor solutions have risen to the challenge.

A preprocessor is a program that takes one type of data and converts it to another type of data. In the case of HTML and CSS, some of the more popular preprocessor languages include Haml and Sass. Haml is processed into HTML and Sass is processed into CSS.

Upon setting out to solve some of the more common problems, Haml and Sass found many additional ways to empower HTML and CSS, not only by removing the inefficiencies but also in creating ways to make building websites easier and more logical. The popularity of preprocessors have also brought along different frameworks to support them, one of the more popular being Compass.

Haml

Haml, known as HTML abstraction markup language, is a markup language with the single goal of providing the ability to write beautiful markup. Serving as its own markup language, code written in Haml is later processed to HTML. Haml promotes DRY and well structured markup, providing a pleasing experience for anyone having to write or read it.

Installation

Haml requires Ruby to be compiled to HTML, so the first step to using it is to ensure that Ruby is installed. Fortunately for those on Mac OS X Ruby comes preinstalled, and those on a Windows machine may visit Windows Installer for directions. Upon confirming Ruby is installed the `gem install haml` command needs to be run from the command line, using Terminal or the alike command line program, to install Haml.

```
gem install haml
```

Files written in the Haml markup should be saved with the file extension of `.haml`. To then convert these files from Haml to HTML the `haml` command below needs to be run to compile each individual file.

```
haml index.haml index.html
```

In the example above, the file `index.haml` is converted to HTML and saved as `index.html` within the same directory. This command has to be run within the same directory the files reside in. Should the command be run outside this directory the path where the files reside need to be included within the command. At any time the command `haml --help` may be run to see a list of different available options.

Watching a File or Directory

Unfortunately Haml doesn't provide a way to watch a file, or directory, for changes without the use of another dependency.

Inside of a Rails application a Haml dependency may be added in the Gemfile, thus automatically compiling Haml files to HTML upon any changes. There are a few desktop applications available for those not using Rails, one of the more popular being CodeKit.

On top of Haml CodeKit also supports other preprocessors, which may also come in handy.

Doctype

The first part to writing a document in Haml is knowing what type of `doctype` is to be used. When working with HTML documents, the general document type is going to be the HTML5 `doctype`. In Haml document types are identified with three exclamation points, `!!!` followed by any specifics if necessary.

The default `doctype` in Haml is the HTML 1.0 Transitional document type so in order to make this the HTML5 `doctype` the number five has to be passed in after the exclamation points, `!!! 5`.

Haml

```
!!! 5
```

Compiled HTML

```
<!DOCTYPE html>
```

Declaring Elements

One of the defining features of Haml is its syntax, and how to declare and nest elements. HTML elements generally have opening and closing tags, however within Haml elements only have one tag, the opening. Elements are initialized with a percent sign, `%`, and then indented to identify nesting. Indentation with Haml can be accomplish with one or more spaces, however what is important is that the indentation remain consistent. Hard tabs or spaces cannot be mixes together, and the same number of tabs or spaces must be the same throughout an entire document.

Removing the need for both opening and closing tags, as well as mandating the structure with indentation creates an easy to follow outline. At any given time the markup can be scanned and changed without struggle.

Haml

```
%body
  %header
    %h1 Hello World
  %section
    %p Lorem ipsum dolor sit amet.
```

Compiled HTML

```
<body>
  <header>
    <h1>Hello World</h1>
  </header>
  <section>
    <p>Lorem ipsum dolor sit amet.</p>
  </section>
</body>
```

Handling Text

Text within Haml can be placed on the same line as the declared element, or indented below the element. Text cannot be both on the same line as the declared element and nested below it, it has to be either or. The example from above could be rewritten as the following:

```
%body
  %header
    %h1
      Hello World
  %section
    %p
      Lorem ipsum dolor sit amet.
```

Attributes

Attributes, as with elements, are declared a bit differently in Haml. Attributes are declared directly after the element in either `{}` or `()`, all depending if you wish to use Ruby or HTML syntax. Ruby style attributes will use the standard hash syntax inside of `{}`, while HTML style attributes will use standard HTML syntax inside of `()`.

Haml

```
%img{:src => "shay.jpg", :alt => "Shay Howe"}
%img{src: "shay.jpg", alt: "Shay Howe"}
%img(src="shay.jpg" alt="Shay Howe")
```

Compiled HTML

```
<img src="shay.jpg" alt="Shay Howe">
```

Classes & IDs

If you wish to, Class and ID attributes may be declared the same as all other attributes, however they may also be treated a bit differently. Rather than listing out the class or ID attribute name and value inside `{}` or `()` the value can be identified directly after the element. Using either a `.` for classes or a `#` for an ID the value can be added directly after the element.

Additionally, attributes may be mixed and matched, chaining them together in the appropriate format. Classes are to be separated with a `.` and other attributes may be added using one of the previously outlined formats.

Haml

```
%section.feature
%section.feature.special
%section#hello
%section#hello.feature(role="region")
```

Compiled HTML

```
<section class="feature"></section>
<section class="feature special"></section>
<section id="hello"></section>
<section class="feature" id="hello" role="region"></section>
```

Division Classes & IDs

In the event a class or ID is used on a `div` the `%div` may be omitted, and the class or ID value can be used outright. Again, classes are to be identified with a `.` and IDs are to be identified with a `#`.

Haml

```
.awesome
```

```
.awesome.lesson
#getting-started.lesson
```

Compiled HTML

```
<div class="awesome"></div>
<div class="awesome lesson"></div>
<div class="lesson" id="getting-started"></div>
```

Boolean Attributes

Boolean attributes are handled just as they would be within Ruby or HTML, all depending on the syntax being used.

Haml

```
%input{:type => "checkbox", :checked => true}
%input(type="checkbox" checked=true)
%input(type="checkbox" checked)
```

Compiled HTML

```
<input type="checkbox" checked>
```

Escaping Text

One of the benefits of Haml is the ability to evaluate and run Ruby, however this isn't always the desired action. Text, and lines of code, can be escaped by using a backslash, `\`, allowing the text to be rendered explicitly without being executed.

In the example below, the first instance of `= @author` is executed Ruby, pulling the authors name from the application. The second instance, starting with the backslash, is escaped text, printing it as is, without execution.

Haml

```
.author
  = @author
  \= @author
```

Compiled HTML

```
<div class="author">
  Shay Howe
  = @author
</div>
```

Text Escaping Alternatives

Occasionally escaping text doesn't quite do the job and Ruby is needed to generate the desired output. One popular instance of this is when trying to include a period directly after a link, but not as part of the anchor text. Putting the period on a new line isn't acceptable as it will be treated as an empty class value, causing a compiling error. Adding a backslash before the period will escape the character however it places a blank space between the last word and the period. Again, not producing the desired output.

In these cases a Ruby helper comes in handy. In the example below, the helper is used to place a period directly after the last word but still outside of the anchor text.

Haml

```
%p
  Shay is
  = succeed "." do
    %a{:href => "#"} awesome
```

Compiled HTML

```
<p>Shay is <a href="#">awesome</a>.</p>
```

Comments

As with elements and attributes, comments are handled a bit differently in Haml as well. Simply enough, code can be commented out with the use of a single forward slash, `/`. Individual lines may be commented out with the use of a forward slash at the beginning of the line, and blocks of code can be commented out by being nested underneath a forward slash.

Haml

```
%div
  / Commented line
  Actual line

/
  %div
    Commented block
```

Compiled HTML

```
<div>
  <!-- Commented line -->
  Actual line
</div>

<!--
  <div>
```

```
      Commented block
    </div>
  -->
```

Conditional Comments

Conditional comments are also handled differently in Haml. To create a conditional comment use square brackets, `[]`, around the condition. These square brackets need to be placed directly after the forward slash.

Haml

```
/[if lt IE 9]
  %script{:src  => "html5shiv.js"}
```

Compiled HTML

```
<!--[if lt IE 9]>
  <script src="html5shiv.js"></script>
<![endif]-->
```

Silent Comments

Haml also provides the ability to create Haml specific comments, or silent comments. Silent comments differ from general HTML comments in that upon being complied any content within a silent comment is completely removed from the page, and is not displayed in the output. Silent comments are initialized with a dash then the number sign, `-#`. As with other comments, silent comments may be used to remove one line or multiple lines with the use of nesting.

Haml

```
%div
  -# Removed line
  Actual line
```

Compiled HTML

```
<div>
  Actual line
</div>
```

Filters

Haml provides a handful of filters, allowing different types of input to be used inside of Haml. Filters are identified with a colon followed by the name of the filter, `:markdown` for example, with all of the content to be filtered nested underneath.

Common Filters

Below are some of the more common filters, with the more popular ones of the group being `:css` and `:javascript`.

- `:cdata`
- `:coffee`
- `:css`
- `:erb`
- `:escaped`
- `:javascript`
- `:less`
- `:markdown`
- `:maruku`
- `:plain`
- `:preserve`
- `:ruby`
- `:sass`
- `:scss`
- `:textile`

Javascript Filter

Haml

```
:javascript
  $('button').on('click', function(event) {
    $('p').hide('slow');
  });
```

Compiled HTML

```
<script>
  $('button').on('click', function(event) {
    $('p').hide('slow');
  });
</script>
```

CSS & Sass Filters

Haml

```
:css
  .container {
    margin: 0 auto;
    width: 960px;
  }

:sass
  .container
    margin: 0 auto
    width: 960px
```

Compiled HTML

```
<style>
  .container {
    margin: 0 auto;
    width: 960px;
  }
</style>
```

Ruby Interpolation

As previously mentioned Haml can evaluate Ruby, and there may occasionally be times where Ruby needs to be evaluated inside of plain text. In this event Ruby needs to be interpolated, accomplished by wrapping the necessary Ruby code inside .

Below is an example of Ruby being interpolated as part of a class name.

Haml

```
%div{:class => "student-#{@student.name}"}
```

Compiled HTML

```
<div class="student-shay">
```

SCSS & Sass

SCSS and Sass are preprocessing languages which are compiled to CSS, resembling Haml a bit in that they make writing code easier, and provide quite a bit of leverage in doing so. Individually SCSS and Sass come from the same origin however they are technical different syntaxes.

Sass, Syntactically Awesome Stylesheets, came first and is a strict indented syntax. SCSS, Sassy CSS, followed shortly after providing the same firing power of Sass but with a more flexible syntax, including the ability to write plain CSS.

Installation

As with Haml, SCSS and Sass are compiled using Ruby therefore Ruby needs to be installed to produce CSS files. Please follow the directions from before to install, or ensure Ruby is installed.

Once Ruby is installed the `gem install sass` command needs to be run from the command line to install SCSS and Sass.

```
gem install sass
```

Files written in SCSS or Sass need to have the `.scss` or `.sass` file extensions respectively. To convert either of these file types to `.css` the following `sass` command needs to be run.

```
sass styles.sass styles.css
```

The command above takes the `styles.sass` Sass file and compiles it to the `styles.css` file. As with Haml, these file names are paths and need to be executed respectively. The above command works when those files reside within the directory from which the command is run, should the files reside outside of the directory their path needs to be explicitly stated within the command.

Should changes to a file be ongoing Sass can watch the file and recompile the CSS every time a change takes place. To watch a Sass file the following `sass` command may be run.

```
sass --watch styles.sass:styles.css
```

Additionally, instead of compiling or watching individual files, Sass is capable of compiling and watching entire directories of files. For example, to watch an entire directory of Sass files and convert them to CSS the `sass` command below may be run.

```
sass --watch assets/sass:public/css
```

Converting Files from SCSS to Sass & Vice Versa

On top of being able to convert SCSS and Sass files to CSS you can also convert files from SCSS to Sass and vice versa. To do so the `sass` commands below may be used to convert a SCSS file to Sass, and then a Sass file to SCSS respectively.

```
# Convert Sass to SCSS
sass-convert styles.sass styles.scss

# Convert SCSS to Sass
sass-convert styles.scss styles.sass
```

Syntax

As previously mentioned the primary difference between SCSS and Sass is their syntax, and their difference in severity. The syntax of SCSS isn't much different than that of regular CSS. In fact, standard CSS will run inside of SCSS. Sass on the other hand is fairly strict, and any indenting or character errors will prohibit the styles from compiling. Sass omits all curly brackets, `{}`, and semicolons, `;`, relying on indentation and clear line breaks for formatting.

SCSS

```
.new {
  color: #ff7b29;
```

```
    font-weight: bold;
    span {
      text-transform: uppercase;
    }
  }
```

Sass

```
.new
  color: #ff7b29
  font-weight: bold
  span
    text-transform: uppercase
```

Compiled CSS

```
.new {
  color: #ff7b29;
  font-weight: bold;
}
.new span {
  text-transform: uppercase;
}
```

Using SCSS vs. Sass

Deciding on whether to use SCSS or Sass boils down to personal preference, and is a decision to be made based on what is best for a specific team and project. There are pros and cons to each syntax, all of which are fair.

Personally, I prefer the Sass syntax as it requires less characters and provides, I believe, a cleaner syntax. Sass will not allow straight CSS input as SCSS does, and will not put up with any composition errors. Sass has a bit more of a learning curve, however a learning curve I see well worth the ease of manageable styles.

Moving forward the examples in this lesson will use Sass, however they may also all be accomplished with SCSS.

Nesting

In the syntax example above you will notice how selectors may be nested inside of one another to create compound selectors. The nesting quickly outlines identifiable selectors, however it is important not to go overboard. Do **not** nest selectors for unapparent reasons or go overboard nesting one selector under the prior one. Using specific selectors without raising specificity is important.

Sass

```
```

```
.portfolio
  border: 1px solid #9799a7
  ul
    list-style: none
  li
    float: left
```

Compiled CSS

```
.portfolio {
  border: 1px solid #9799a7;
}
.portfolio ul {
  list-style: none;
}
.portfolio li {
  float: left;
}
```

Nesting Properties

On top of nesting selectors it is also possible to nest properties. Some of the most popular uses of this may be seen with `font`, `margin`, `padding`, and `border` properties. As with the decision of SCSS versus Sass, this is very much a personal decision. Many feel that shorthand values are fine and breaking out values in this longer format is unnecessary. Ultimately your decision is up to personal preference.

Sass

```
div
  font:
    family: Baskerville, Palatino, serif
    style: italic
    weight: normal
```

Compiled CSS

```
div {
  font-family: Baskerville, Palatino, serif;
  font-style: italic;
  font-weight: normal;
}
```

Nested Media Queries

Individual media queries may also be nested inside of a selector, changing property values based off a media condition.

Sass

```sass
.container
  width: 960px
  @media screen and (max-width: 960px)
    width: 100%
```

Compiled CSS

```css
.container {
  width: 960px;
}
@media screen and (max-width: 960px) {
  .container {
    width: 100%;
  }
}
```

Parent Selector

Sass provides a way to add styles to a previous selector with the use of the parent selector, implemented by using an ampersand, `&`. Most commonly the parent selector is used in conjunction with a pseudo class, such as `:hover`, however it doesn't have to be. Additionally the parent selector could be used to bind additional selectors to its parent, such as `&.featured`.

Sass

```sass
a
  color: #0087cc
  &:hover
    color: #ff7b29
```

Compiled CSS

```css
a {
  color: #0087cc;
}
 a:hover {
  color: #ff7b29;
}
```

Parent Key Selector

The parent selector may also be used as the key selector, adding qualifying selectors to make compound selectors. There are an abundance of ways to use the parent selector as the key selector but perhaps one of the most beneficial is inside of feature detection.

Sass

```sass
.btn
```

```
  background: linear-gradient(#fff, #9799a7)
  .no-cssgradients &
    background: url("gradient.png") 0 0 repeat-x
```

Compiled CSS

```
.btn {
  background: linear-gradient(#fff, #9799a7);
}
.no-cssgradients .btn {
  background: url("gradient.png") 0 0 repeat-x;
}
```

Comments

Sass handles comments very similar to that of Haml. The standard CSS syntax, `/* ... */`, for comments works as intended within Sass however there is also a syntax for silent comments to completely remove a comment or lines of code from being compiled.

The syntax for silent comments is two forward slashes, `//`, and any content on that line or nested below it will be omitted from computation. Notice in the example below how the `// Omitted comment` line is not rendered in the compiled CSS.

Sass

```
/* Normal comment */
div
  background: #333
// Omitted comment
strong
  display: block
```

Compiled CSS

```
/* Normal comment */
div {
  background: #333;
}

strong {
  display: block;
}
```

Variables

Variables are one of the more sought after features of CSS that Sass provides. With Sass you can define variables and then reuse them as necessary.

Variables are defined with a dollar sign, `$`, followed by the variable name. Between the variable name and value is a colon followed by an empty space, such as `$font-base: 1em`. As for the value of the variable, it may be a number, string, color, boolean, null, or a list of values separated by spaces or commas.

Sass

```
$font-base: 1em
$serif: "Helvetica Neue", Arial, "Lucida Grande", sans-serif

p
  font: $font-base $serif
```

Compiled CSS

```
p {
  font: 1em "Helvetica Neue", Arial, "Lucida Grande", sans-serif;
}
```

Variable Interpolation

For the most part variables may be used anywhere inside of a Sass document. However, they may occasionally need to be interpolated using the syntax. A few instances of where variables need to be interpolated include when being used in a class name, property name, or inside a string of plain text.

Sass

```
$location: chicago
$offset: left

.#{$location}
  #{$offset}: 20px
```

Compiled CSS

```
.chicago {
  left: 20px;
}
```

Calculations

Sass also has the ability to do calculations in a variety of different manners. Calculations can handle most problems, such as addition, subtraction, division, multiplication, and rounding.

Addition can be done by using the plus sign, `+`, and may be completed with or without units of measurement. When done with units, the unit tied to the first number in the equation is the unit that will be used in the computed value. For example, ten pixels plus

one inch will equal 106 pixels. Subtraction is handled the same way as addition but with the minus sign, `-`, instead.

Multiplication is completed with the asterisk sign, `*`, however only one of the numbers, if any, may include a unit of measurement. Using the percent sign, `%`, will return the remainder of the two numbers upon being divided, and as with multiplication, only allows one number, if any, to have a unit.

Sass

```
width: 40px + 6
width: 40px - 6
width: 40px * 6
width: 40px % 6
```

Compiled CSS

```
width: 46px;
width: 34px;
width: 240px;
width: 4px;
```

Division

Division is a bit trickier in Sass as the forward slash, `/`, used to perform division is already used in some CSS property values. Generally speaking, division will take place when any part of the value uses a variable, if the value is wrapped in parentheses, or if the value is used as part of another equation.

When using one unit of measurement in division the value will reside in that unit. When using two units of measurement, however, the resulting value will be unitless.

Sass

```
width: 100px / 10
width: (100px / 10)
width: (100px / 10px)
$width: 100px
width: $width / 10
width: 5px - 100px / 10
```

Compiled CSS

```
width: 100px/10;
width: 10px;
width: 10;
width: 10px;
width: -5px;
```

Detailed Math

As one may expect, Sass is also capable of combining multiple math operations. Sass also recognizes which operations to execute first based on the use of parentheses.

Sass

```
$grid: 16
$column: 40px
$gutter: 20px
$container: ($column * $grid) + ($gutter * $grid)

width: $container
```

Compiled CSS

```
width: 960px;
```

Number Functions

By default Sass includes a handful of built in functions, many of which are used to manipulate number values as wished.

The `percentage()` function turns a value into a percentage. The `round()` function rounds a value to the closest whole number, defaulting to rounding up where necessary. The `ceil()` function rounds a value up to the closest whole number, and the `floor()` function rounds a value down to the closest whole number. Lastly, the `abs()` function finds the absolute value of a given number.

- `percentage()`
- `round()`
- `ceil()`
- `floor()`
- `abs()`

Sass

```
width: percentage(2.5)
width: round(2.5px)
width: ceil(2.5px)
width: floor(2.5px)
width: abs(-2.5px)
```

Compiled CSS

```
width: 250%;
width: 3px;
width: 3px;
width: 2px;
width: 2.5px;
```

Color

Sass provides quite a bit of assistance in working with colors, providing a handful of different features to alter and manipulate colors. One of the more popular color features in Sass is the ability to change a hexadecimal color, or variable, and convert it into an RGBa value.

Sass

```
color: rgba(#8ec63f, .25)

$green: #8ec63f
color: rgba($green, .25)
```

Compiled CSS

```
color: rgba(142, 198, 63, .25);
```

Color Operations

On top of numbers, math may additionally be performed on colors using addition, subtraction, multiplication, and division. These computations are performed on the red, green, and blue components, changing them as intended.

Sass

```
color: #8ec63f + #666
color: #8ec63f * 2
color: rgba(142, 198, 63, .75) / rgba(255, 255, 255, .75)
```

Compiled CSS

```
color: #f4ffa5;
color: #ffff7e;
color: rgba(0, 0, 0, .75);
```

Color Alterations

Using color operators to perform calculations is helpful but can be a bit challenging as well. In this case color alterations may be a better option. Color alterations provide the ability to inverse colors, find complementary colors, mix colors together, or find the grayscale value of a color.

- `invert()`
- `complement()`
- `mix()`
- `grayscale()`

Sass

```
color: invert(#8ec63f)
color: complement(#8ec63f)
```

```
color: mix(#8ec63f, #fff)
color: mix(#8ec63f, #fff, 10%)
color: grayscale(#8ec63f)
```

Compiled CSS

```
color: #7139c0;
color: #773fc6;
color: #c6e29f;
color: #f3f9eb;
color: #838383;
```

HSLa Color Alterations

HSLa color alterations take things a step further, adding in even more alterations. Some of the more popular HSLa color alterations include `lighten()`, `darken()`, `saturate()`, and `desaturate()`.

- `lighten()`
- `darken()`
- `saturate()`
- `desaturate()`
- `adjust-hue()`
- `fade-in()`
- `fade-out()`

Sass

```
color: lighten(#8ec63f, 50%)
color: darken(#8ec63f, 30%)
color: saturate(#8ec63f, 75%)
color: desaturate(#8ec63f, 25%)
color: adjust-hue(#8ec63f, 30)
color: adjust-hue(#8ec63f, -30)
color: fade-in(rgba(142, 198, 63, 0), .4)
color: fade-out(#8ec63f, .4)
```

Compiled CSS

```
color: white;
color: #3b5319;
color: #98ff06;
color: #89a75e;
color: #4ac63f;
color: #c6bb3f;
color: rgba(142, 198, 63, 0.4);
color: rgba(142, 198, 63, 0.6);
```

Color Manipulation

Outside of altering colors Sass can also directly manipulate colors. Manipulating colors provides the most control over how to finely tune specific color properties. With this control also comes complexity, which is why color manipulations are a bit less common than color alterations.

- `change-color()` — Set any property of a color
  `$color, [$red], [$green], [$blue], [$hue], [$saturation], [$lightness], [$alpha]`

- `adjust-color()` — Incrementally manipulate any property of a color
  `$color, [$red], [$green], [$blue], [$hue], [$saturation], [$lightness], [$alpha]`

- `scale-color()` — Fluidly scale any percentage based on property of a color
  `$color, [$red], [$green], [$blue], [$saturation], [$lightness], [$alpha]`

Sass

```
color: change-color(#8ec63f, $red: 60, $green: 255)
color: adjust-color(#8ec63f, $hue: 300, $lightness: 50%)
color: scale-color(#8ec63f, $lightness: 25%, $alpha: 30%)
```

Compiled CSS

```
color: #3cff3f;
color: white;
color: #aad46f;
```

Extends

Extends provide a way to easily share and reuse styles without having to explicitly repeat code or use additional classes, providing a perfect way to keep code modular. Both elements and class selectors may be used as an extend, and there is even a placeholder selector built just for extends.

Extends are established by using the `@extend` rule followed by the selector to extend. Instead of duplicating the property and values, the original selector receives and additional selector, that of which is from the selector calling the extend.

In all, this provides a way to quickly reuse code without driving up code weight. Additionally, extends parley nicely with OOCSS and SMACSS.

Sass

```
.alert
  border-radius: 10px
  padding: 10px 20px

.alert-error
  @extend .alert
  background: #f2dede
  color: #b94a48
```

```
8
```

Compiled CSS

```
1  .alert,
2  .alert-error {
3    border-radius: 10px;
4    padding: 10px 20px;
5  }
6  .alert-error {
7    background: #f2dede;
8    color: #b94a48;
9  }
```

Placeholder Selector Extend

To avoid building a bunch of unused classes purely for extends we can use what is known as a placeholder selector. The placeholder selector is initialized with a percentage sign, `%`, and is never directly compiled into CSS. Instead, it is used to attach selectors to when it is called with an extend. In the refined example below notice how the `.alert` selector never makes its way into the CSS.

Sass

```
1  %alert
2    border-radius: 10px
3    padding: 10px 20px
4
5  .alert-error
6    @extend %alert
7    background: #f2dede
8    color: #b94a48
```

Compiled CSS

```
1  .alert-error {
2    border-radius: 10px;
3    padding: 10px 20px;
4  }
5  .alert-error {
6    background: #f2dede;
7    color: #b94a48;
8  }
```

Element Selector Extend

As with classes, extends also work with standard element selectors too.

Sass

```
```

```
h2
  color: #9c6
  span
    text-decoration: underline

.sub-heading
  @extend h2
```

Compiled CSS

```
h2, .sub-heading  {
  color: #9c6;
}
h2 span, .sub-heading span {
  text-decoration: underline;
}
```

Mixins

Mixins provide a way to easily template properties and values, then share them amongst different selectors. Mixins differ from extends as mixins allow arguments to be passed in where extends are fixed values.

Mixins are identified using the `@mixin` rule followed by any potential arguments, then any styles are outlined below the rule. To call a mixin from within a selector use the plus sign, `+`, followed by the name of the mixin and any desired argument values if needed.

It is worth nothing that SCSS handles mixins a bit different. Instead of using a plus sign to call a mixin SCSS use an `@include` rule.

Sass

```
@mixin btn($color, $color-hover)
  color: $color
  &:hover
    color: $color-hover

.btn
  +btn($color: #fff, $color-hover: #9799a7)
```

Compiled CSS

```
.btn {
  color: #fff;
}
.btn:hover {
  color: #9799a7;
}
```

Default Arguments

Using the same example from above we can also specify default argument values, which may be over written if wished.

Sass

```
@mixin btn($color: #fff, $color-hover: #9799a7)
  color: $color
  &:hover
    color: $color-hover

.btn
  +btn($color-hover: #9799a7)
```

Compiled CSS

```
.btn {
  color: #fff;
}
.btn:hover {
  color: #9799a7;
}
```

Variable Arguments

When one or more values need to be passed to an argument the variable name may end with `...` inside of the mixin. In the example below with box shadows we can pass in comma separated values to the mixin.

```
@mixin box-shadow($shadow...)
  -webkit-box-shadow: $shadow
  -moz-box-shadow: $shadow
  box-shadow: $shadow

.shadows
  +box-shadow(0 1px 2px #cecfd5, inset 0 0 5px #cecfd5)
```

Compiled CSS

```
.shadows {
  -moz-box-shadow: 0 1px 2px #cecfd5, inset 0 0 5px #cecfd5;
  -webkit-box-shadow: 0 1px 2px #cecfd5, inset 0 0 5px #cecfd5;
  box-shadow: 0 1px 2px #cecfd5, inset 0 0 5px #cecfd5;
}
```

Imports

One of nicest parts of Sass is its ability to import multiple `.scss` or `.sass` files and condense them into one single file. Condensing all of the files into one allows for multiple stylesheets to be used for better organization without the worry of numerous HTTP request.

Instead of referencing all of the different stylesheets within an HTML document only reference the one Sass file importing all of the other stylesheets.

In the following examples, all three files `_normalize.sass`, `_grid.sass`, and `_typography.sass` are all compiled into one file. In the event that the Sass file importing all the other files is named `styles.sass`, and it is compiled into `styles.css`, then only `styles.css` needs to be referenced within the HTML document.

Sass

```sass
@import "normalize"
@import "grid", "typography"
```

Compiled HTML

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

Loops & Conditionals

For a bit more intricate styling Sass supports different control directives. Its important to understand these directives are not intended for everyday styling but for creating detailed mixins and helpers. Many of these will look familiar as they are borrowed from other programming languages.

Operators

Some loops and conditionals will require operators to determine their behavior, of which can be broken down into relational and comparison operators. Relational operators looks at the relationship between two entities, while comparison operators determine equality or different between to entities.

- `<`
  Less than
- `>`
  Greater than
- `==`
  Equal to
- `<=`
  Less than or equal to
- `>=`
  Greather than or equal to
- `!=`
  Not equal to

```sass
// Relational Operators
6 < 10   // true
4 <= 60  // true
8 > 2    // true
```

```
10 >= 10 // true

// Comparison Operators
#fff == white  // true
10 + 30 == 40  // true
normal != bold // true
```

If Function

The `@if` rule test an expressions then loads the styles beneath that expression should it return anything other than `false` or `null`. The initial `if` statement may be proceeded by several `else if` statements and one `else` statement. Once a statement is successful identified the styles directly tied to it will be applied.

Sass

```
$shay: awesome

.shay
  @if $shay == awesome
    background: #ff7b29
  @else if $shay == cool
    background: #0087cc
  @else
    background: #333
```

Compiled CSS

```
.shay {
  background: #ff7b29;
}
```

For Loop

The `@for` rule outputs different sets of styles based off of a counter variable. There are two different forms available for `for` loops, those being `to` and `through`. The first, `@for $i from 1 to 3` for example, will output styles up to, but not including, `3`. The other form, `@for $i from 1 through 3`, will output styles up to, and including, `3`.

Sass

```
@for $col from 1 to 6
  .col-#{$col}
    width: 40px * $col
```

Compiled CSS

```
.col-1 {
  width: 40px;
```

```
  }
  .col-2 {
    width: 80px;
  }
  .col-3 {
    width: 120px;
  }
  .col-4 {
    width: 160px;
  }
  .col-5 {
    width: 200px;
  }
```

Each Loop

Simply enough, the `@each` rule returns styles for each item in a list. List may include multiple comma separated items.

Sass

```
@each $class in uxd, rails, html, css
  .#{$class}-logo
    background: url("/img/#{$class}.jpg")
```

Compiled CSS

```
.uxd-logo {
  background: url("/img/uxd.jpg");
}
.rails-logo {
  background: url("/img/rails.jpg");
}
.html-logo {
  background: url("/img/html.jpg");
}
.css-logo {
  background: url("/img/css.jpg");
}
```

While Loop

The `@while` rule repeatedly returns styles until the statement becomes false. The directive accepts a handful of different operators and the counter variable can be finely controlled allowing for precise looping.

Sass

```
$heading: 1
@while $heading <= 6
```

```
    h#{$heading}
      font-size: 2em - ($heading * .25em)
    $heading: $heading + 1
```

Compiled CSS

```
h1 {
  font-size: 1.75em;
}
h2 {
  font-size: 1.5em;
}
h3 {
  font-size: 1.25em;
}
h4 {
  font-size: 1em;
}
h5 {
  font-size: 0.75em;
}
h6 {
  font-size: 0.5em;
}
```

Other Preprocessors

Haml and Sass are far from the only preprocessing languages available, including JavaScript preprocessors as well. Some of the other popular preprocessors including Jade, Slim, LESS, Stylus, and CoffeeScript.

In the interest of brevity Haml and Sass were the only preprocessors covered in this lesson. They were also chosen because they are built using Ruby and fit right into Ruby on Rails applications. They've also got tremendous community support.

When it comes to choosing which, if any, preprocessor to use it is important to consider what is best for your team and project. Projects built in Node.js may likely better benefit from Jade and Stylus. The most important aspect to consider, though, is what your team is accustomed to using. Do your research for each project and make the most educated decision.

Resources & Links

- Haml — HTML Abstraction Markup Language
- Sass — Syntactically Awesome Stylesheets
- Haml Documentation Reference
- Sass Documentation Reference
- Sass Playground via SassMeister
- SassScript Function via Sass Documentation
- HSLa Color Function Visualization via SassMe