SASS Variables and Mixins

Written by Margaret Rodgers

From Web Team

Contents

Variables

A variable in SASS works exactly the same as a variable in any programming language; it's an empty container waiting for something to be put into it. Once it holds a value, that variable can be called into use anywhere. If a global change needs to be made, that change only needs to be made in the appropriate variable and affects every location where that variable is called.

To create a new variable in your SASS document, you simply prefix the variable name with a dollar sign. The syntax is also more similar to CSS than a programming variable because a colon is used to assign value rather than an assignment operator:

$topLevel_font

To give it a value, add a colon after the name and then your value followed by a semi-colon:

$topLevel_font: 30px;

The idea is exactly the same as in other types of programming; you create your variable, give it a value and call it from within your script. In the case of SASS, of course, you're going to create a variable whose value is one of the properties of a style. So in the example of topLevel_font, if that were going to be the size of every h1 in the stylesheet you would use the variable in all classes containing an h1 element rather than retyping the property in all those classes. So this:

```
h1 {
  font-size: $topLevel_font;
}
```

Rather than how we would normally write it:

```
h1 {
  font-size: 30px;
}
```

If you later decide that all your first level headings should be 25 pixels instead of 30, you simply change that value in the variable declaration, recompile the stylesheet and the change is automatically reflected in every h1.

Nested Variables

In HTML you have a definite structure where child elements are indented so it's clear who belongs to whom. In CSS, however, everything is pretty much a square block; all at the same level. But SASS offers nesting so it's much easier to see what is subordinate to what. Here's an example using an HTML5 article that contains a header, a paragraph and an unordered list. All of these things have styles applied to them. In typical CSS your rules would look like this:

```
ul.contacts{
  background-color:green;
  list-style-type: none;
  padding:0;
}

ul.contacts li article {
  border: solid 5px;
  background-color:#ff5e00;
  margin-bottom: 10px;
  padding:10px;
}

ul.contacts li article header {
  font-style:italic;
}

ul.contacts li article p {
  color:white;
}
```

But with SASS we can nest all of those associated classes under one declaration.

```
ul.contacts{
  background-color:green;
  list-style-type: none;
  padding:0;

  li {
     article {
       border: solid 5px;
        background-color:#ff5e00;
        margin-bottom: 10px;
       padding:10px;
```

```
        header {
          font-style:italic;
        }

 p {
                    color:white;
                  }
              }
            }
}
```

When rendered in the stylesheet, the nested code looks just like the example we started out with. So the nesting does not affect the end product after compilation – it just makes it a little easier to read. CSS3 provides this same type of nesting but earlier versions of CSS do not.

Mixins

Mixins are a way to eliminate redundant styling and do a lot of other neat stuff. For example, let's say you had 15 classes. 10 of them have three properties that are exactly the same and two additional properties that are the same but have one of two different values. Each also has one additional property that has a different value in every class.

Normally you would have to write the same thing over and over again so it appears in all 10 of them. That's a lot of extra writing and if you decide to change something, you have to do it all over again!

However, if you use a mixin, you simply create a mixin which holds the three properties which are the same. Then you call the mixin in the 10 classes instead of rewriting everything.

For demo purposes, we'll say that our 10 classes have the same background color, font color and padding. They all have a font size property which has one of two values; 1em or 1.5em and a width setting that is different in every one. Because those font sizes and widths are different, we can't include that property in the mixin.

Or can we?

You can configure your mixin in a couple of different ways. You can either use individual CSS property statements and their values or you can create variables, give them the values and then use them in your mixin. I'm going to use variables in my example here.

So first up, create the variables and give them values.

```
$bgColor: #0072ff;
$fontColor: white;
$paddingAll: 10px 30px 10px 10px;
```

Since we know one of the other properties will have one of two font size values, we'll go ahead and create some variables to hold those while we're at it.

```
$fontSize: 1em;
$fontSizeBigger: 1.5em
```

Next step is to build the mixin using the three common variables. You'll notice that the mixin declaration really resembles a function declaration in programming languages – there is a name, an argument and opening and closing curly braces. However, unlike a language, SASS uses the @ sign to declare a mixin:

```
//Creating the mixin
@mixin tooMuchRedundancy(){
  background-color: $bgColor;
  color: $fontColor;
  padding: $paddingAll;
}
```

The mixin is called into the stylesheet through the use of a SASS '@include' directive. So you simply replace those three property declarations in all class rules that contain them with an include directive followed by the name of the mixin. Now the code in your SCSS file will look similar to this:

```
.blueButton{
  @include tooMuchRedundancy;
  font-size: 1.5em;
  width: 250px;
}
```

So we've pared down the class to three lines instead of six. Once compiled, the styles are rendered like this in the stylesheet:

```
.blueButton{
  background-color:#0072ff;
  color: white;
  padding: 10px 30px 10px 10px;
  font-size: 1.5em;
  width: 250px;
}
```

Well, this is really nice, but there are still a couple of properties that handle font size in all 10 of the classes. Can these be included in the mixin after all? You bet! To handle these, we'll use a 'parameterized mixin'.

Because we have the ability to give the mixin an argument as we would with any function, we will need to provide a parameter so it knows to expect something. There are a couple of ways to do this.

Let's look at the blueButton style for a second. Right now it looks like this in the SASS file:

```
.blueButton{
  @include tooMuchRedundancy;
  font-size:1.5em;
  width: 250px;
}
```

First we need to adjust the mixin called tooMuchRedunancy. When we declared it we gave it the ability to accept a parameter should we decide to give it one. So now we need to do that. Let's give it an empty variable that will be replaced with a value later on.

```
@mixin tooMuchRedundancy($newFont){
  background-color: $bgColor;
  color: $fontColor;
  padding: $paddingAll
  $newFont:$fontSize;
}
```

I've given it a parameter called $newFont which has no value, it's just an empty container that's acting as a placeholder. That empty container will be filled using one of the font size variables that we declared earlier in the argument, $fontSize for the smaller font or $fontSizeBigger, and with that addition we can eliminate another line of code from the SASS file.

```
.blueButton{
  @include tooMuchRedundancy($fontSize);
  width: 250px;
}
```

Once compiled the CSS looks as you would expect:

```
.blueButton{
  background-color:#0072ff;
  color: white;
  padding: 10px 30px 10px 10px;
  font-size: 1em;
  width: 250px;
}
```

But there is still one more property to deal with; width. Since the width is different for all 10 of the classes, it isn't really viable to create variables to hold the values. Instead, we can pass the

value of the width property to the mixin as part of the argument. To do this, we'll create another mixin and give it a parameter:

```
@mixin setWidth($bestSize){
  width:$bestSize;
}
```

Now, when we call the setWidth mixin, we'll just place the width in the argument instead of the empty variable $bestSize:

```
.blueButton{
  @include $setWidth(250px);
  @include tooMuchRedundancy($fontSize);
}
```

So in the SASS file, you now have two lines that define the style of blueButton instead of the original six. After SASS compiles the stylesheet, the resulting CSS looks as you would expect:

```
.blueButton{
  background-color: #0072ff;
  color: white;
  padding:10px 30px 10px 10px;
  font-size: 1em;
  width: 250px;
```

Variables and mixins can be used very effectively with media queries as well.

Inheritance

If there are classes that inherit properties from other rules, you can take advantage of SASS's ability to handle inheritance.

For example, we'll say that there are three buttons, all of them are exactly the same except the border around each is different. Because of that, we need to write a series of rules to assign the border property to each button.

```
.wideBorder{
  border: 5px solid;
}

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

```
.dottedBorder{
  border: 5px dotted;
}
```

Since they all inherit the style of the blue button that we defined earlier, they would need to be included in the blueButton class declaration. We could write them in or use @extend in SASS to have SASS do the work for us. The border definition classes now look like this:

```
.wideBorder{
  @extend .blueButton;
  border: 5px solid;
}

.narrowBorder{
  @extend .blueButton;
  border: 1px solid;
}

.dottedBorder{
  @extend .blueButton;
  border: 5px dotted;
}
```

When SASS compiles the stylesheet, the compiled CSS looks like this:

```
.blueButton, .wideBorder, .narrowBorder, .dottedBorder{
  background-color: #0072ff;
  color: white;
  padding:10px 30px 10px 10px;
  font-size: 1em;
  width: 250px;
}

.wideBorder{
  border: 5px solid;
}

.narrowBorder{
  border: 1px solid;
}

.dottedBorder{
  border: 5px dotted;
}
```

You might ask “So what’s the advantage? I could have written that myself with about the same amount of effort as I spent setting up the SASS stuff!” The difference is in the way you assign the style in the HTML. Where before the HTML would require assignment of both classes, .blueButton and one of the border classes:

<div class=”blueButton wideBorder”>

Using the @extend to cause each border class to inherit the properties of blueButton, only one class needs to be applied to the CSS.

There is a huge amount of functionality that I’ve not covered here. But these are a few of the basic things that can be done using SASS and the ones I felt would, most likely be of use on our site.