Introduction to the Sass Challenges

Sass, or "Syntactically Awesome StyleSheets", is a language extension of CSS. It adds features that aren't available using basic CSS syntax. Sass makes it easier for developers to simplify and maintain the style sheets for their projects.

Sass can extend the CSS language because it is a preprocessor. It takes code written using Sass syntax, and converts it into basic CSS. This allows you to create variables, nest CSS rules into others, and import other Sass files, among other things. The result is more compact, easier to read code.

There are two syntaxes available for Sass. The first, known as SCSS (Sassy CSS) and used throughout these challenges, is an extension of the syntax of CSS. This means that every valid CSS stylesheet is a valid SCSS file with the same meaning. Files using this syntax have the .scss extension.

The second and older syntax, known as the indented syntax (or sometimes just "Sass"), uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties. Files using this syntax have the .sass extension.

This section introduces the basic features of Sass.

Sass: Store Data with Sass Variables

One feature of Sass that's different than CSS is it uses variables. They are declared and set to store data, similar to JavaScript.

In JavaScript, variables are defined using the letand constkeywords. In Sass, variables start with a $followed by the variable name.

Here are a couple examples:

$main-fonts: Arial, sans-serif;  
$headings-color: green;  
  
//To use variables:  
h1 {  
  font-family: $main-fonts;  
  color: $headings-color;  
}

One example where variables are useful is when a number of elements need to be the same color. If that color is changed, the only place to edit the code is the variable value.

Sass allows nestingof CSS rules, which is a useful way of organizing a style sheet.

Normally, each element is targeted on a different line to style it, like so:

nav {  
  background-color: red;  
}  
  
nav ul {  
  list-style: none;  
}  
  
nav ul li {  
  display: inline-block;  
}

For a large project, the CSS file will have many lines and rules. This is where nestingcan help organize your code by placing child style rules within the respective parent elements:

nav {  
  background-color: red;  
  
  ul {  
    list-style: none;  
  
    li {  
      display: inline-block;  
    }  
  }  
}

Sass: Create Reusable CSS with Mixins

In Sass, a mixinis a group of CSS declarations that can be reused throughout the style sheet.

Newer CSS features take time before they are fully adopted and ready to use in all browsers. As features are added to browsers, CSS rules using them may need vendor prefixes. Consider "box-shadow":

div {  
  -webkit-box-shadow: 0px 0px 4px #fff;  
  -moz-box-shadow: 0px 0px 4px #fff;  
  -ms-box-shadow: 0px 0px 4px #fff;  
  box-shadow: 0px 0px 4px #fff;  
}

It's a lot of typing to re-write this rule for all the elements that have a box-shadow, or to change each value to test different effects.

Mixinsare like functions for CSS. Here is how to write one:

@mixin box-shadow($x, $y, $blur, $c){   
  -webkit-box-shadow: $x, $y, $blur, $c;  
  -moz-box-shadow: $x, $y, $blur, $c;  
  -ms-box-shadow: $x, $y, $blur, $c;  
  box-shadow: $x, $y, $blur, $c;  
}

The definition starts with @mixinfollowed by a custom name. The parameters (the $x, $y, $blur, and $cin the example above) are optional.

Now any time a box-shadowrule is needed, only a single line calling the mixinreplaces having to type all the vendor prefixes. A mixinis called with the @includedirective:

div {  
  @include box-shadow(0px, 0px, 4px, #fff);  
}