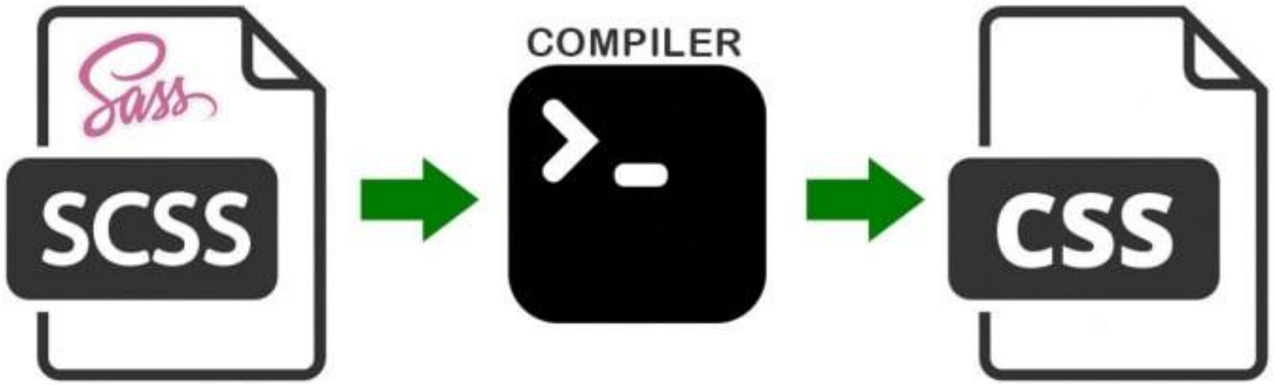
CSS preprocessor

SASS - Syntactically Awesome Stylesheets

What is a CSS preprocessor?

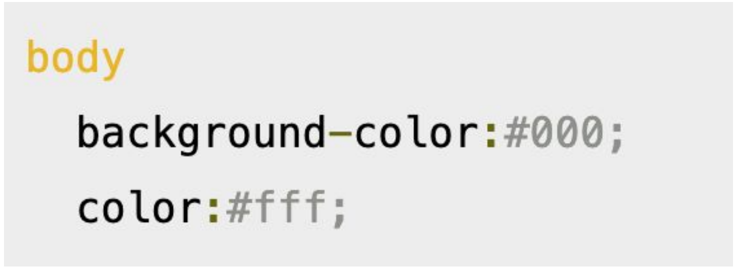
A CSS preprocessor is a scripting language that extends CSS by allowing developers to write code in one language and then compile it into CSS, other well-known examples include Less and Stylus.



What is SASS/SCSS

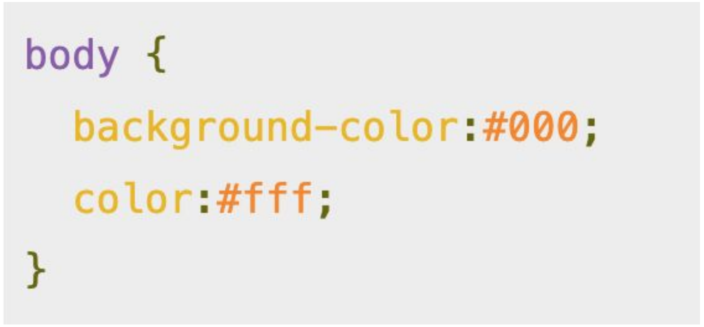
SASS Stands for Syntactically Awesome Style Sheets and it uses strict indentation (like Python)

SCSS known as Sassy CSS is a newer and most used version of Sass. It has a similar syntax to CSS because it's a language extension or a superset of CSS. Its file extension is .scss



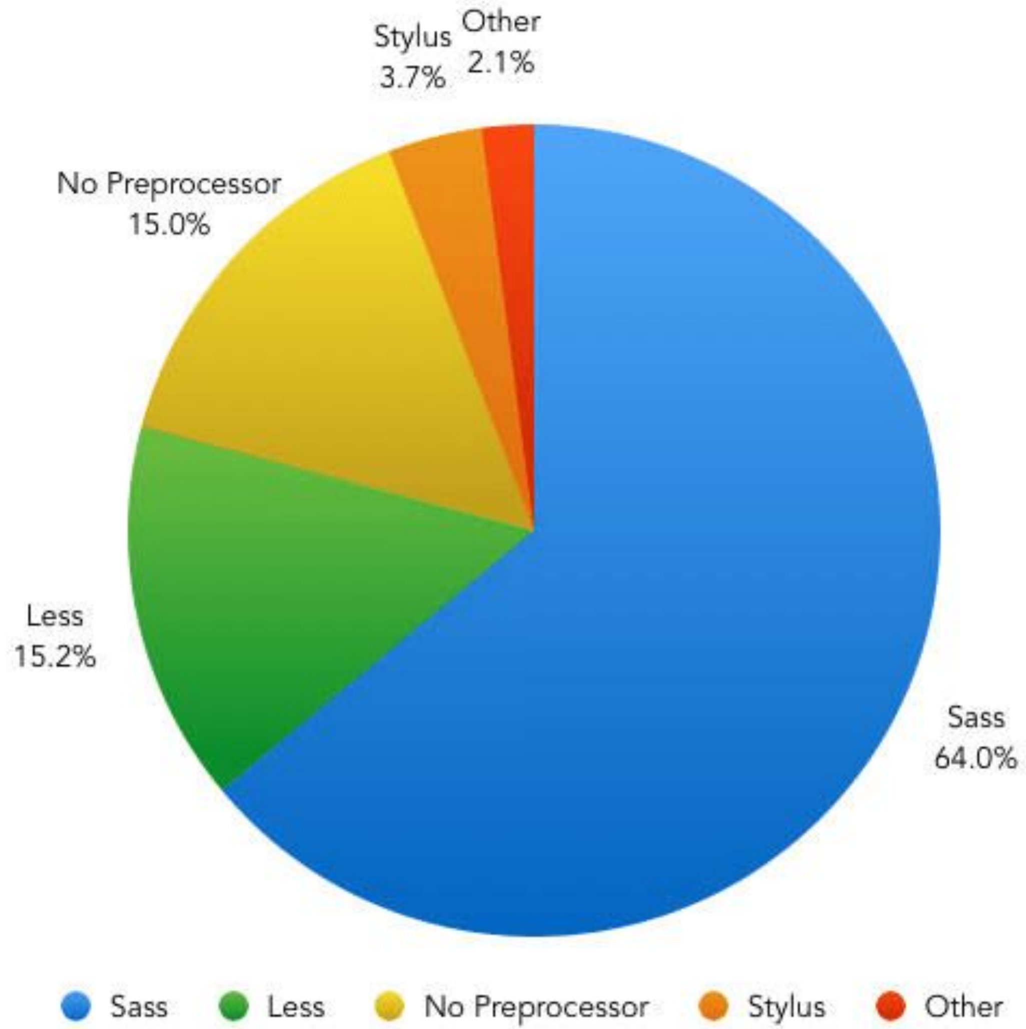
```
body
   background-color:#000;
   color:#fff;
```

SASS



```
body {
   background-color:#000;
   color:#fff;
}
```

SCSS


Stylus
3.7%
Other
2.1%
No Preprocessor
15.0%
Less
15.2%
Sass
64.0%
Sass
Less
No Preprocessor
Stylus
Other

Nesting

- Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML.
- The Inception Rule: don’t go more than four levels deep.
- More natural syntax and easy to read in most cases
- Prevents the need to rewrite selectors multiple times
- Better code organization and structure thanks to its visual hierarchy, which bring us to more maintainable code.

```
nav {
  background-color:#333;
  padding:1em;
}
nav ul {
  margin:0;
  padding:0;
  list-style:none;
}
nav ul li {
  display:inline-block;
}
```

```
nav {
  background-color:#333;
  padding:1em;
  ul {
    margin:0;
    padding:0;
    list-style:none;
    li {
      display:inline-block;
    }
  }
}
```

Using & in nesting

& always refers to the upper selection.

If I have a class at parent element called `.container`, and children with class `.childrenClass` then & will refer parent class `.container` and you don't need to write it again

Children selection => `& .childrenClass` similar to `.container .childrenClass`

Pseudo-class selection => `&:first-child` similar to `.container:first-child`

Concatenations => `&-main` similar to `.container-main`

Snippits

```
button {
  background-color: #535353;
  color: #000;
  &:hover {
    background-color: #000;
    color: #fff;
  }
}
```

```
.some-class {
  &:hover {
    /* when hovered */
  }
  &:focus {
    /* when focused */
  }
  & > button {
    /* selector equls to .some-class > button */
  }
  &-cool {
    /*** Notice this! ****/
    // selects .some-class-cool elements
  }
}
```

Exercises

```
.alert {
    &:hover {
        font-weight: bold;
    }

    button & {
        margin-left: 0;
        margin-right: 10px;
    }
}
```

&:hover similar to: … ?

Button & similar to: … ?

Variables

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse

$ symbol to make something a variable

- Set it as => $primary-color: #333;
- Use it as => color: $primary-color

This can be extremely powerful when working with brand colors and keeping them consistent throughout the site

Variable Declaration

```scss
$my-font: Nunito, sans-serif;
$my-font-color: #ffd969;
$content-width: 660px;
```

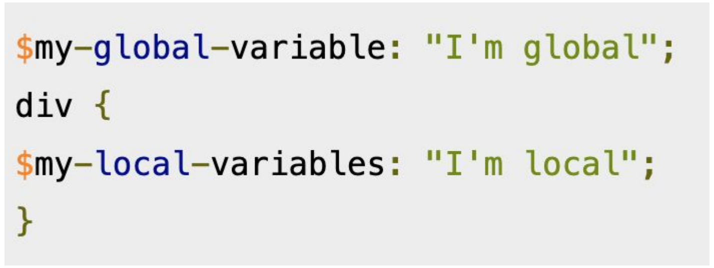
Variable Usage

```scss
body {
  font-family: $my-font;
  color: $my-font-color;
  content {
    width: $content-width;
  }
}
```

SCSS variable scope

All variables defined in the top level are global

All variables defined in blocks (i.e., inside curly braces) are local



```
$my-global-variable: "I'm global";
div {
$my-local-variables: "I'm local";
}
```

Exercises

```
$color: #fefefe;

.content {
      background-color: $color;
}

$color: #939393;

.footer {
      background-color: $color;
}
```

Content color is: #fefefe
Footer color is: #939393
Global color is: #939393

```
$color: #111;

.content {
       $color: #222;
        background-color: $color;
}

.test {
         $color: #333;
}

.footer {
         $color: #333 !global;
}
```

Content color is: #222;
Footer color is: #333;
Global at Test color level is: #111;
Global color is: #333;

Directives

- @mixin
- @import, @use
- @if, @else
- @for, @while.

@Mixins

Allow you to define styles that can be re-used throughout your stylesheet.

The syntax is similar to functions in JavaScript. Instead of the function keyword, use the @mixin directive.

Calling the mixin is done via the @include statement.

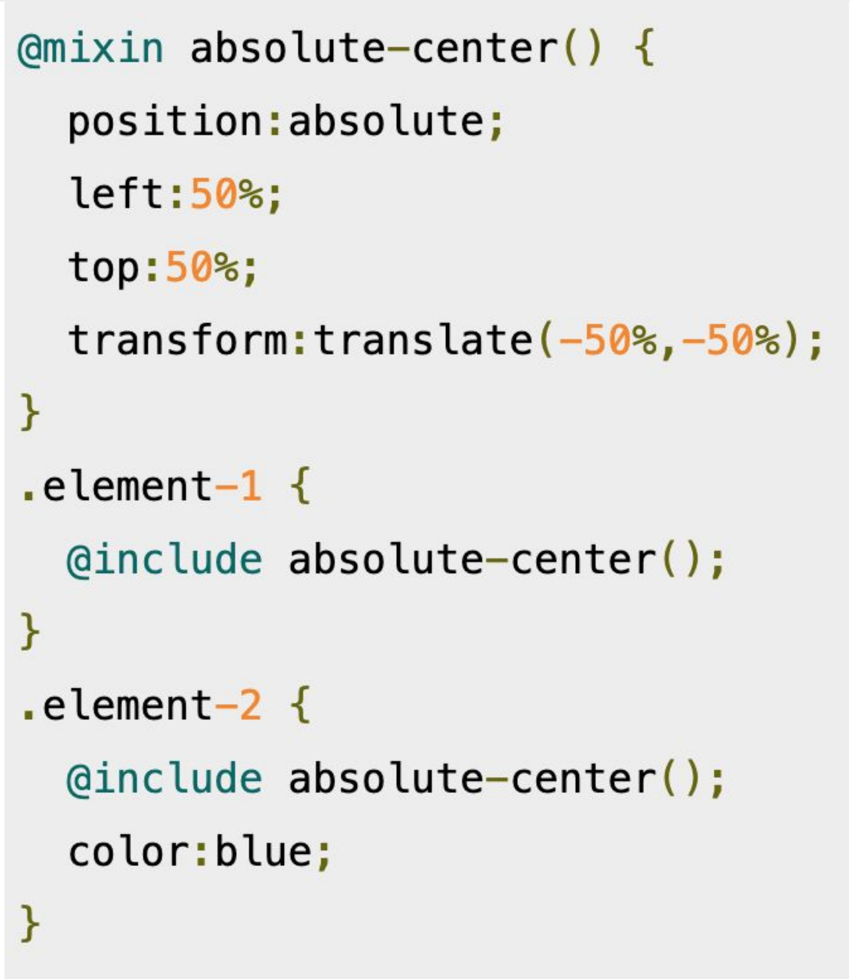
You can have arguments too.

Snippits

Here we defined absolute-center mixin without arguments and we reused it again using @include in element-1 and element-2

absolut-center() can also be written as object because we pass no arguments here and be like:

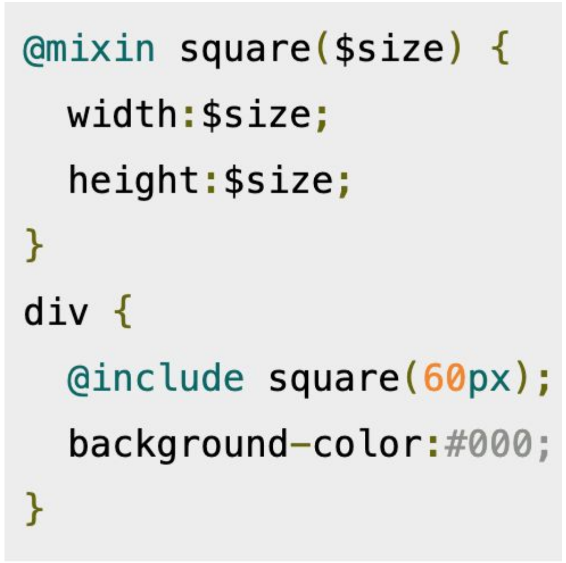
- @mixin absolute-center {....}
- @include absolute-center;



```scss
@mixin absolute-center() {
  position:absolute;
  left:50%;
  top:50%;
  transform:translate(-50%,-50%);
}
.element-1 {
  @include absolute-center();
}
.element-2 {
  @include absolute-center();
  color:blue;
}
```

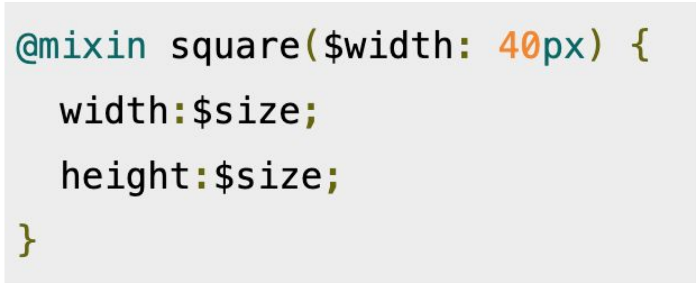
Mixins with arguments & optional arguments

With arguments



```
@mixin square($size) {
  width:$size;
  height:$size;
}
div {
  @include square(60px);
  background-color:#000;
}
```

With optional arguments



```
@mixin square($width: 40px) {
  width:$size;
  height:$size;
}
```

We pass default value to the arguments, then if user didn’t pass any value at calling it @include square(), it will use the default value of width

Mixins with content blocks

Instead of arguments, we can send CSS rules to the mixins. Those rules can be used in the mixin using **@content**

We write mixin as js object not function, then add the rule we want to apply to child, then add @content to be replaced with content at calling

This approach allows us to reduce the repetition of the `&:not([disabled]):hover` part.

```scss
@mixin hover-not-disabled {
  &:not([disabled]):hover {
    @content;
  }
}
.button {
  border: 1px solid black;
  @include hover-not-disabled {
    border-color: blue;
  }
}
```

Partials and @Import

The @import directive allows you to include the content of one file in another, like at styles.scss you can include normalize file as following `@import 'normalize';` if both on them on the same level in css folder.

If we also wanted to import customStyles.scss at styles.scss we import it like normalize `@import customStyles;` and all the variables, mixins, etc in customStyles will be globally accessible.

which is a problem when you have complex file structures and use libraries. For this reason, using @importis now officially discouraged.

When _ is prepended to a file name of a SCSS file, the parser knows that it is a partial file and it is there only for importing ex: `src/_colors.scss`

@use

The basic usage of @use is the same as that of @import, but files imported with @use are called modules, To use mixins or variables of these modules, we have to call them using namespaces, so no more global variables

src/_colors.scss

```scss
$accent-color: #535353;
@mixin dark-background {
  background-color:#000;
  color:#fff;
}
```

styles.scss

```scss
@use 'src/colors';
body {
  color: colors.$accent-color;
}
.dark-region {
  @include colors.dark-background;
}
```

styles.scss

```scss
@use 'src/colors' as c;
body  {
  color: c.$accent-color;
}
```

Custom namespace

Build in functions, check => https://sass-lang.com/documentation/modules

Sass provides the following built-in modules:

- **sass:math:** provides functions that operate on numbers.
- **sass:string:** makes it easy to combine, search, or split apart strings.
- **sass:color:** generates new colors based on existing ones, making it easy to build color themes.
- **sass:list:** lets you access and modify values in lists.
- **sass:map** makes it possible to look up the value associated with a key in a map, and much more.
- **sass:selector:** provides access to Sass's powerful selector engine.
- **sass:meta:** exposes the details of Sass's inner workings.

Usage:

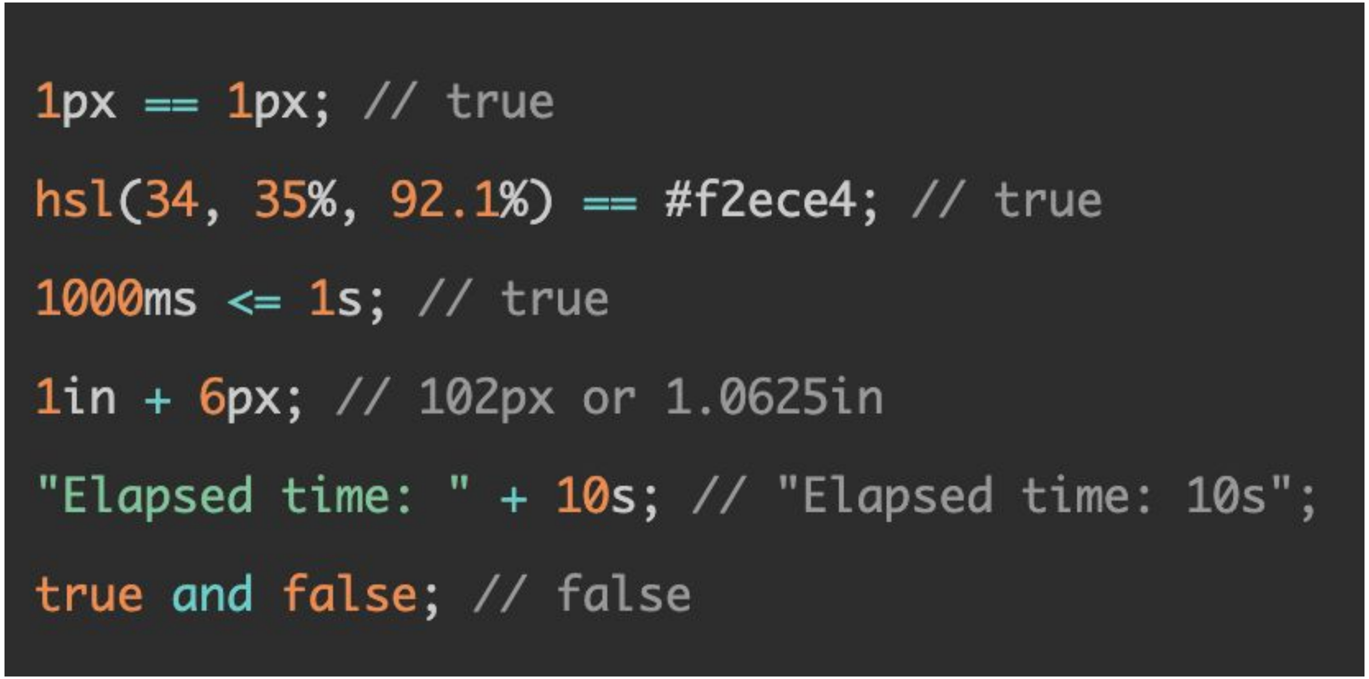
- Get it first => @use "sass:string";
- Use it => string.unquote(".widget:hover"); // .widget:hover
- Another example => string.quote(bold); // "bold"

Arithmetic operators

CSS calc() function allows us to perform basic calculation. But scss have more functionalities. You can do arithmetic (+, -, *, /, %), equality, boolean and conditional operations.

```scss
.inner-content {
  width: $content-width - 60px; // substraction
}
```

Fun fact: You can add two colors to get a new color

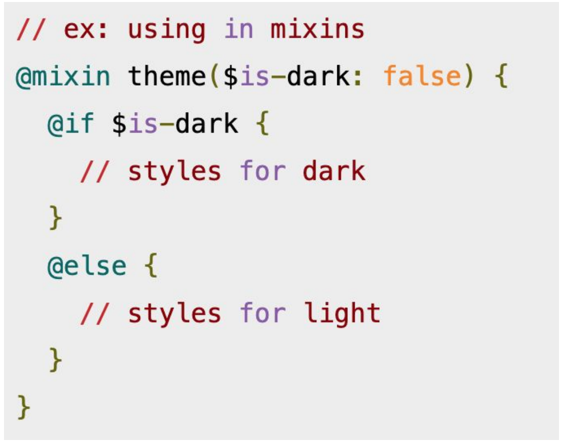


```
1px == 1px; // true
hsl(34, 35%, 92.1%) == #f2ece4; // true
1000ms <= 1s; // true
1in + 6px; // 102px or 1.0625in
"Elapsed time: " + 10s; // "Elapsed time: 10s";
true and false; // false
```

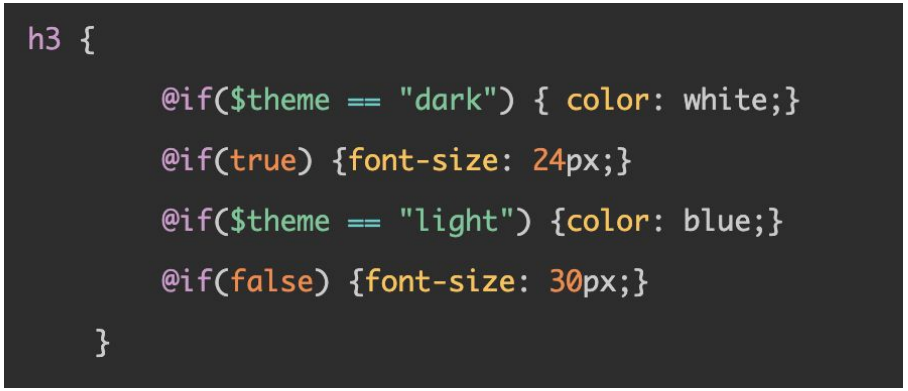
Flow control rules

There are four types of flow control rules: **@if /@else, @each, @for, and @while.**

1- **@if and @else**– are similar to if and else in JavaScript.

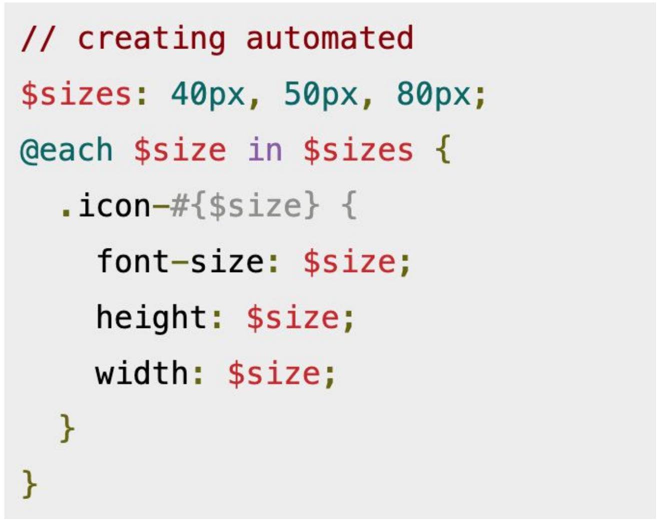


```scss
// ex: using in mixins
@mixin theme($is-dark: false) {
  @if $is-dark {
    // styles for dark
  }
  @else {
    // styles for light
  }
}
```



```scss
h3 {
        @if($theme == "dark") { color: white;}
        @if(true) {font-size: 24px;}
        @if($theme == "light") {color: blue;}
        @if(false) {font-size: 30px;}
    }
```

2- @each is similar to for of in JavaScript.

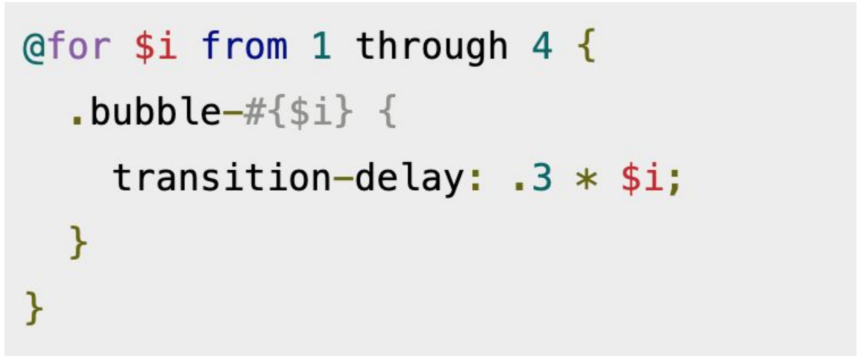


```scss
// creating automated
$sizes: 40px, 50px, 80px;
@each $size in $sizes {
  .icon-#{$size} {
    font-size: $size;
    height: $size;
    width: $size;
  }
}
```

The css of this example should be like: ... ?

Check this for more details about interpolation : https://sass-lang.com/documentation/interpolation

3- @for is similar to for loops in JavaScript.



```
@for $i from 1 through 4 {
  .bubble-#{$i} {
    transition-delay: .3 * $i;
  }
}
```

4- @while (not often used) is similar to while loops in JavaScript.

Adding Media Queries

CSS way

```
@media screen and (max-width:600px) {
  body > content {
    padding:0;
  }
}
```

SCSS way

```
body > content {
  @media screen and (max-width:600px) {
      padding:0;
  }
}
```

Both methods are valid; you can use whichever you prefer. Some developers save all media queries in a single file (e.g., Mobile.scss).

Using SCSS in real-world applications

1- Open terminal

- start
- search then cmd and open it

2- Download node.js if you don’t downloaded it before

- Write `node -v` in terminal to if you have node on your device, if node not found error then download it
- Visit this like https://nodejs.org/en/download/ and git suitable version for your PC
- After installation `node -v` should works fine now

3- Write `npm install -g sass` in terminal as per documentation https://sass-lang.com/install

4- you need to install an extension in your editor that watches your SCSS files and converts them to regular CSS each time you save “Live SASS Compiler” extension in VSCode for example.

5- Create a styles.scss file. When you create a .scss file, you will see a pink ‘s’ logo on the side of the file in your explorer window indicating that it’s a SASS file if you are using vscode editor

6- Click the ‘watch sass’ button on the bottom right of your VSCode screen., or Alternatively to the extension you can use the following command `sass --watch <input folder>:<output folder>` like that `sass --watch demo/scss/style.scss:demo/css/style.css`

7- The compiler will automatically create two additional files in your directory. styles.css and styles.css.map

8- the next step is to link your CSS to your HTML. For this step the only file you need to worry about and link is the styles.css file. <link rel=”stylesheet” href=”style.css” />

9- Now we are ready to start writing SCSS code in the styles.scss document which will be converted to css by live sass compiler each time we hit save.

10- Now we are ready to rock!

Questions

References & resources

https://smazee.com/blog/complete-guide-to-sass

https://blog.logrocket.com/the-definitive-guide-to-scss/

https://sass-lang.com/documentation

https://www.sassmeister.com/

https://devhints.io/sass