SASS and SCSS

CSS Preprocessors

A CSS preprocessor is a scripting language that extends CSS and compiles it into regular CSS. It allows you to use features that don't exist in pure CSS, such as variables, nested rules, functions, and mixins.

Popular CSS Preprocessors

- SASS (Syntactically Awesome Stylesheets)
- LESS (Leaner Style Sheets)
- Stylus

Benefits of Using CSS Preprocessors

- Maintainability: Write cleaner and more organized code.
- Reusable Code: Use variables, mixins, and functions to reuse code.
- Modularity: Split CSS into multiple files and import them.
- Advanced Features: Nesting, inheritance, and logical operations.

Differences Between SASS and SCSS

SASS (Syntactically Awesome Style Sheets) is a CSS preprocessor that adds power and elegance to the basic language. It has two syntax variations: the original SASS syntax and the newer SCSS (Sassy CSS) syntax. Both are used to write more maintainable and reusable CSS, but they have distinct syntaxes.

SASS Syntax

- Indentation-Based: Uses indentation to separate code blocks instead of braces {}.
- No Semicolons: Does not require semicolons at the end of statements.
- File Extension: Files using the SASS syntax have the .sass extension.

Example:

```
$primary-color: #3498db

body
  color: $primary-color

nav
  ul
    margin: 0
    padding: 0
    list-style: none

  li
    display: inline-block

  a
    text-decoration: none
    color: $primary-color
```

SCSS Syntax

- CSS-Like: Uses curly braces {} to denote code blocks and semicolons; to separate statements.
- Fully Compatible with CSS: Any valid CSS code is also valid SCSS.
- File Extension: Files using the SCSS syntax have the .scss extension.

Example:

```
$primary-color: #3498db;

body {
  color: $primary-color;
}

nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }
  li {
    display: inline-block;
  }

  a {
    text-decoration: none;
    color: $primary-color;
  }
}
```

Key Differences

1. Syntax Style
   - SASS: Uses indentation, no braces, and no semicolons. More concise but less familiar to those used to traditional CSS syntax.
   - SCSS: Uses braces and semicolons, making it more similar to regular CSS and easier to transition for those familiar with CSS.

2. Readability and Conciseness
   - SASS: More concise due to the lack of braces and semicolons, which can make it easier to write but harder to read for some.
   - SCSS: More verbose, but the syntax is clearer and more readable, especially for those used to CSS.

3. Compatibility
   - SASS: Requires a bit of a learning curve due to its unique syntax.
   - SCSS: Fully compatible with CSS, making it easy to integrate with existing CSS codebases and adopt incrementally.

4. Community and Adoption

   - SASS: Older syntax, less commonly used in new projects compared to SCSS.
   - SCSS: Preferred syntax in the modern SASS ecosystem due to its compatibility with CSS and readability.

Nesting

Nesting in SASS allows you to nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. This makes your stylesheet more readable and organized, especially for complex stylesheets with many nested elements.

Benefits of Nesting

- Improved Readability: Code structure mirrors HTML structure, making it easier to understand.
- Better Organization: Group related styles together.
- Reduced Redundancy: Avoid repeating selectors.

Example of Nesting
Consider the following HTML structure:

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
  </ul>
</nav>
```

In SASS, you can nest the styles like this:

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;

    li {
      display: inline-block;

      a {
        text-decoration: none;
        color: #3498db;
      }
    }
  }
}
```

The compiled CSS will be:

```css
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav ul li {
  display: inline-block;
}

nav ul li a {
  text-decoration: none;
  color: #3498db;
}
```

Key Points

- Use to reflect the HTML structure.
- Helps in grouping related styles.
- Can be overused; deeply nested selectors can become hard to read and maintain.

Parent Selector

The parent selector in SASS, denoted by the & symbol, refers to the parent selector in a nested rule. This is useful for pseudo-classes, pseudo-elements, and modifier classes.

Benefits of the Parent Selector

- Enhanced Control: Apply styles to parent elements or combine parent selectors with pseudo-classes/elements.
- Dynamic Nesting: Easily create more specific selectors.

Examples of Using the Parent Selector

1. Pseudo-Classes and Pseudo-Elements

HTML:

```
<a href="#">Hover over me</a>
```

SASS:

```
a {
  color: #3498db;

  &:hover {
    color: darken(#3498db, 10%);
  }
}
```

The compiled CSS will be:

```
a {
  color: #3498db;
}

a:hover {
  color: #2c80b4;
}
```

2. Modifier Classes

HTML:

```
<button class="button button--large">Large Button</button>
<button class="button button--small">Small Button</button>
```

SASS:

```
.button {
  background: #3498db;
  color: white;

  &--large {
    padding: 1em 2em;
  }

  &--small {
    padding: 0.5em 1em;
  }
}
```

The compiled CSS will be:

```
.button {
  background: #3498db;
  color: white;
}

.button--large {
  padding: 1em 2em;
}

.button--small {
  padding: 0.5em 1em;
}
```

3. Combining Parent Selectors

HTML:

```
<nav class="nav">
  <ul>
    <li class="nav__item"><a href="#"
class="nav__link">Home</a></li>
    <li class="nav__item"><a href="#"
class="nav__link">About</a></li>
  </ul>
</nav>
```

SASS:

```
.nav {
  &__item {
    display: inline-block;
  }

  &__link {
    text-decoration: none;

    &:hover {
      text-decoration: underline;
    }
  }
}
```

The compiled CSS will be:

```
.nav__item {
  display: inline-block;
}

.nav__link {
  text-decoration: none;
}

.nav__link:hover {
  text-decoration: underline;
}
```

Key Points

- Use & to reference the parent selector.
- Useful for pseudo-classes, pseudo-elements, and BEM (Block Element Modifier) methodology.
- Enhances the specificity and readability of your styles.

Variables

Variables in SASS allow you to store values (such as colors, fonts, or any CSS value) and reuse them throughout your stylesheet. This makes it easier to maintain and update your styles, as you only need to change the value in one place.

Benefits of Variables

- Reusability: Define a value once and use it multiple times.
- Maintainability: Easily update styles by changing the variable's value.
- Consistency: Ensure consistent use of values across your stylesheets.

Syntax and Example:
To define a variable, use the $ symbol followed by the variable name.
Example:

```
// Define variables
$primary-color: #3498db;
$font-stack: Helvetica, sans-serif;

body {
  color: $primary-color;
  font-family: $font-stack;
}
```

The compiled CSS will be:

```
body {
  color: #3498db;
  font-family: Helvetica, sans-serif;
}
```

Advanced Variable Usage

Variables can also be used within other SASS features like functions, mixins, and nesting.
Example with Nesting and Functions:

```
$base-font-size: 16px;

@mixin font-size($size) {
  font-size: $size;
  font-size: $size * 1px; // Example of using a variable in mixin
}

body {
  @include font-size($base-font-size);
}
```

The compiled CSS will be:

```
body {
  font-size: 16px;
  font-size: 16px;
}
```

Key Points

- Store values for reuse.
- Improve maintainability and consistency.
- Can be used within other SASS features.

Mixins

Mixins in SASS allow you to create reusable chunks of CSS that you can include in other selectors. Mixins can take arguments, making them extremely powerful for applying styles dynamically.

Benefits of Mixins

- Reusability: Define a set of styles once and reuse them multiple times.
- Maintainability: Update styles in one place.
- Dynamic Styles: Use arguments to create flexible and dynamic styles.

Syntax and Example
To define a mixin, use the @mixin directive followed by the mixin name.
Example:

```
// Define a mixin
@mixin border-radius($radius) {
  border-radius: $radius;
}

// Include the mixin
.box {
  @include border-radius(10px);
}
```

The compiled CSS will be:

```
.box {
  border-radius: 10px;
}
```

Advanced Mixin Usage

Mixins can also include nested rules, use arguments with default values, and include conditional logic.

Example with Nested Rules and Arguments:

```
@mixin button($color, $padding: 10px) {
  background-color: $color;
  padding: $padding;
  border: none;
  color: white;

  &:hover {
    background-color: darken($color, 10%);
  }
}

.button-primary {
  @include button(#3498db);
}

.button-secondary {
  @include button(#2ecc71, 15px);
}
```

The compiled CSS will be:

```
.button-primary {
  background-color: #3498db;
  padding: 10px;
  border: none;
  color: white;
}

.button-primary:hover {
  background-color: #2c80b4;
}

.button-secondary {
  background-color: #2ecc71;
  padding: 15px;
  border: none;
  color: white;
}

.button-secondary:hover {
  background-color: #28a745;}
```

Key Points

- Create reusable chunks of CSS.
- Can take arguments for dynamic styles.
- Support nested rules and conditional logic.

Partials

Partials in SASS are smaller SCSS files that you can import into other SCSS files. They help in organizing and modularizing your CSS, making it easier to maintain and manage large stylesheets.

Benefits of Partials

- Modularity: Break down your CSS into smaller, manageable files.
- Reusability: Use partials across multiple files.
- Maintainability: Easier to update and manage your styles.

Syntax and Example
Partials are named with a leading underscore (_) to indicate that they are partial files. The underscore prevents the file from being compiled into a separate CSS file.
Example:

Create a partial _variables.scss:

```scss
// _variables.scss
$primary-color: #3498db;
$font-stack: Helvetica, sans-serif;
```

Create another partial _mixins.scss:

```scss
// _mixins.scss
@mixin border-radius($radius) {
  border-radius: $radius;
}
```

Import these partials into your main stylesheet styles.scss:

```scss
// styles.scss
@import 'variables';
@import 'mixins';

body {
  color: $primary-color;
  font-family: $font-stack;
}

.box {
  @include border-radius(10px);
}
```

The compiled CSS will be:

```css
body {
  color: #3498db;
  font-family: Helvetica, sans-serif;
}

.box {
  border-radius: 10px;
}
```

Key Points

- Organize and modularize your stylesheets.
- Use the leading underscore (_) to indicate partial files.
- Import partials using the @import directive.

Operators

Operators in SASS allow you to perform calculations and operations on values. This includes mathematical operations (like addition, subtraction, multiplication, and division), as well as string operations and color manipulations.

Benefits of Operators

- Dynamic Calculations: Perform calculations directly in your stylesheets.

- Enhanced Functionality: Create more complex styles using operations.
- Flexibility: Easily adjust values based on calculations.

Syntax and Example

1. Mathematical Operations

You can use basic mathematical operators to perform calculations.

Example:

```
$base-margin: 10px;
$double-margin: $base-margin * 2;

.container {
  margin: $double-margin;
}
```

The compiled CSS will be:

```
.container {
  margin: 20px;
}
```

2. String Operations

You can concatenate strings using the + operator.

Example:

```
$base-url: 'https://example.com/';
$image-path: 'images/photo.jpg';

.background {
  background-image: url(#{$base-url + $image-path});
}
```

The compiled CSS will be:

```
.background {
  background-image: url(https://example.com/images/photo.jpg);
}
```

3. Color Operations

You can manipulate colours using operators.

Example:

```scss
$primary-color: #3498db;
$darken-primary: darken($primary-color, 10%);

.button {
  background-color: $primary-color;

  &:hover {
    background-color: $darken-primary;
  }
}
```

The compiled CSS will be:

```css
.button {
  background-color: #3498db;
}

.button:hover {
  background-color: #2c80b4;
}
```

Key Points

- Perform calculations and operations on values.
- Use mathematical, string, and color operations to create dynamic and flexible styles.
- Enhance the functionality and maintainability of your stylesheets.

Functions

Built-In Functions

SASS provides a wide range of built-in functions that can be used to manipulate and work with colors, numbers, strings, and more. Here are some common built-in functions:

Color Functions:

- lighten($color, $amount): Lightens a color by a specified amount.
- darken($color, $amount): Darkens a color by a specified amount.
- complement($color): Returns the complement of a color.

- mix($color1, $color2, [$weight]): Mixes two colors together.

Example:

```
$primary-color: #3498db;

.light-bg {
  background-color: lighten($primary-color, 20%);
}

.dark-bg {
  background-color: darken($primary-color, 20%);
}

.complement-bg {
  background-color: complement($primary-color);
}
```

The compiled CSS will be:

```
.light-bg {
  background-color: #5dade2;
}

.dark-bg {
  background-color: #2874a6;
}

.complement-bg {
  background-color: #db6234;
}
```

Number Functions:

- percentage($value): Converts a number to a percentage.
- round($value): Rounds a number to the nearest whole number.
- ceil($value): Rounds a number up to the next whole number.
- floor($value): Rounds a number down to the previous whole number.

Example:

```
$decimal: 0.75;

.percentage {
  width: percentage($decimal);
}

.rounded {
  width: round($decimal * 100px);
}
```

The compiled CSS will be:

```
.percentage {
  width: 75%;
}

.rounded {
  width: 75px;
}
```

String Functions:

- to-upper-case($string): Converts a string to uppercase.
- to-lower-case($string): Converts a string to lowercase.

Example:

```
$str: 'Hello World';

.uppercase {
  content: to-upper-case($str);
}

.lowercase {
  content: to-lower-case($str);
}
```

The compiled CSS will be:

```
.uppercase {
  content: 'HELLO WORLD';
}

.lowercase {
  content: 'hello world';
}
```

User-Defined Functions

You can also create your own functions in SASS using the @function directive. User-defined functions allow you to encapsulate reusable logic.
Syntax and Example:

```
@function calculate-spacing($base, $factor) {
  @return $base * $factor;
}

.container {
  margin: calculate-spacing(10px, 2);
}
```

The compiled CSS will be:

```
.container {
  margin: 20px;
}
```

Key Points

Built-In Functions:

- Use built-in functions for common operations on colors, numbers, strings, and more.
- Functions, like lighten, darken, complement, percentage and string manipulation functions, are very useful.

User-Defined Functions:

- Create reusable logic with @function.
- Encapsulate and reuse complex calculations and operations.

@for Loop

The @for loop in SASS allows you to iterate over a range of numbers and generate CSS rules dynamically. It can be used to apply styles to a sequence of elements, create grid systems, and more.
Syntax
The @for loop has two forms:

- @for $var from <start> through <end>: Includes the end value.
- @for $var from <start> to <end>: Excludes the end value.

Example

1. Generating Multiple Classes:

```
@for $i from 1 through 5 {
  .margin-#{$i} {
    margin: $i * 10px;
  }
}
```

The compiled CSS will be:

```
.margin-1 {
  margin: 10px;
}

.margin-2 {
  margin: 20px;
}

.margin-3 {
  margin: 30px;
}

.margin-4 {
  margin: 40px;
}

.margin-5 {
  margin: 50px;
}
```

2. Creating a Simple Grid System:

```
$columns: 12;

@for $i from 1 through $columns {
  .col-#{$i} {
    width: (100% / $columns) * $i;
  }
}
```

The compiled CSS will be:

```css
.col-1 {
  width: 8.33333%;
}

.col-2 {
  width: 16.66667%;
}

.col-3 {
  width: 25%;
}

/* ... */

.col-12 {
  width: 100%;
}
```

Key Points

- Iterate over a range of numbers.
- Dynamically generate CSS rules.
- Useful for creating grids, sequences, and repetitive styles.

@each Loop

The @each loop in SASS is used to iterate over a list or map, applying styles to each item. This loop is particularly useful for generating repetitive styles and handling collections of values efficiently.

Benefits of the @each Loop

- Reusability: Apply the same styles to multiple elements or values.
- Efficiency: Reduce redundant code by iterating over lists or maps.
- Flexibility: Easily manage collections of values and their associated styles.

Syntax and Example

The basic syntax for the @each loop is:

```
@each $item in $list {
  // Styles using $item
}
```

Example with a List:

```
$colors: red, green, blue;

@each $color in $colors {
  .text-#{$color} {
    color: $color;
  }
}
```

The compiled CSS will be:

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

.text-green {
  color: green;
}

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

@each Loop with Mapping Functionality

Mapping functionality allows you to iterate over a map (a collection of key-value pairs) using the @each loop. This is useful for scenarios where you need to handle paired values, such as color names and their corresponding hex values.

Benefits of Mapping Functionality

- Enhanced Organization: Group related data together.
- Simplified Styling: Easily apply styles based on key-value pairs.
- Dynamic Styling: Manage complex relationships between styles and values efficiently.

Syntax and Example

The basic syntax for the @each loop with maps is:

```
@each $key, $value in $map {
  // Styles using $key and $value
}
```

Example with a Map:

```
$theme-colors: (
  primary: #3498db,
  secondary: #2ecc71,
  danger: #e74c3c
);

@each $name, $color in $theme-colors {
  .bg-#{$name} {
    background-color: $color;
  }
}
```

The compiled CSS will be:

```
.bg-primary {
  background-color: #3498db;
}

.bg-secondary {
  background-color: #2ecc71;
}

.bg-danger {
  background-color: #e74c3c;
}
```

Advanced Example with Nested Maps:

```
$sizes: (
  small: (width: 50px, height: 50px),
  medium: (width: 100px, height: 100px),
  large: (width: 150px, height: 150px)
);
```

```
@each $size, $dimensions in $sizes {
  .box-#{$size} {
    @each $property, $value in $dimensions {
      #{$property}: $value;
    }
  }
}
```

The compiled CSS will be:

```
.box-small {
  width: 50px;
  height: 50px;
}

.box-medium {
  width: 100px;
  height: 100px;
}

.box-large {
  width: 150px;
  height: 150px;
}
```

Key Points

- Iterate over lists or maps.
- Apply styles to each item in the collection.
- Useful for generating repetitive styles efficiently.

Mapping Functionality:

- Handle key-value pairs with the @each loop.
- Apply styles based on keys and their associated values.
- Enhance organization and management of related data.

Nested Loops

Nested loops in SASS allow you to iterate over multiple collections within each other. This is useful for generating complex styles that involve multiple dimensions, such as creating a grid system with rows and columns.

Benefits of Nested Loops

- Complex Iterations: Handle multiple levels of iteration.
- Dynamic Styling: Create complex styles with nested structures.
- Efficiency: Reduce redundant code by managing nested collections.

Syntax and Example

You can nest loops within each other by placing one loop inside another.

Example with Two Lists:

```
$colors: red, green, blue;
$sizes: small, medium, large;

@each $color in $colors {
  @each $size in $sizes {
    .#{$color}-#{$size} {
      color: $color;
      font-size: $size;
    }
  }
}
```

The compiled CSS will be:

```
.red-small {
  color: red;
  font-size: small;
}

.red-medium {
  color: red;
  font-size: medium;
}

.red-large {
  color: red;
  font-size: large;
}

.green-small {
  color: green;
  font-size: small;
}
```

```
.green-medium {
  color: green;
  font-size: medium;
}

.green-large {
  color: green;
  font-size: large;
}

.blue-small {
  color: blue;
  font-size: small;
}

.blue-medium {
  color: blue;
  font-size: medium;
}

.blue-large {
  color: blue;
  font-size: large;
}
```

Key Points

- Iterate over multiple collections within each other.
- Create complex styles with nested structures.
- Reduce redundant code and manage nested collections efficiently.

Conditionals

Conditionals in SASS allow you to apply styles based on certain conditions. They use standard control flow statements like @if, @else if, and @else to evaluate expressions and apply styles accordingly.

Benefits of Conditionals

- Dynamic Styling: Apply styles based on conditions.
- Flexibility: Handle different scenarios within your stylesheets.

- Maintainability: Create more readable and logical stylesheets.

Syntax and Example

Basic Conditionals:

The @if statement checks a condition and applies styles if the condition is true.

Example:

```
$theme: dark;

body {
  @if $theme == light {
    background-color: white;
    color: black;
  } @else if $theme == dark {
    background-color: black;
    color: white;
  } @else {
    background-color: gray;
    color: black;
  }
}
```

The compiled CSS will be:

```
body {
  background-color: black;
  color: white;
}
```

Conditionals with Variables:

Conditionals can be combined with variables for more dynamic styling.

Example:

```
$primary-color: #3498db;
$secondary-color: #2ecc71;
$use-primary: true;

.button {
  background-color: if($use-primary, $primary-color,
$secondary-color);
}
```

The compiled CSS will be:

```
.button {
  background-color: #3498db;
}
```

Nested Loops with Conditionals

Combining Nested Loops and Conditionals

You can combine nested loops with conditionals to create even more dynamic and complex styles.

Example:

```
$colors: red, green, blue;
$sizes: small, medium, large;

@each $color in $colors {
  @each $size in $sizes {
    .#{$color}-#{$size} {
      color: $color;

      @if $size == small {
        font-size: 12px;
      } @else if $size == medium {
        font-size: 16px;
      } @else if $size == large {
        font-size: 20px;
      }
    }
  }
}
```

The compiled CSS will be:

```
.red-small {
  color: red;
  font-size: 12px;
}

.red-medium {
  color: red;
  font-size: 16px;
}
```

```
.red-large {
  color: red;
  font-size: 20px;
}

.green-small {
  color: green;
  font-size: 12px;
}

.green-medium {
  color: green;
  font-size: 16px;
}

.green-large {
  color: green;
  font-size: 20px;
}

.blue-small {
  color: blue;
  font-size: 12px;
}

.blue-medium {
  color: blue;
  font-size: 16px;
}

.blue-large {
  color: blue;
  font-size: 20px;
}
```

Key Points

- Apply styles based on conditions.
- Use @if, @else if, and @else for control flow.
- Combine with variables for dynamic styling.

Combining Nested Loops and Conditionals:

- Create even more dynamic and complex styles.
- Handle multiple levels of iteration and conditional logic together.

Conclusion

SASS (Syntactically Awesome Style Sheets) and SCSS (Sassy CSS) significantly enhance the capabilities of traditional CSS, providing developers with powerful tools to write more maintainable, scalable, and efficient stylesheets. Through the use of variables, mixins, nesting, partials, and operators, SASS introduces a level of flexibility and modularity that simplifies complex styling tasks. Variables ensure consistency and ease of updates, while mixins allow for reusable chunks of code, reducing redundancy. Nesting enhances the readability of the code by reflecting the HTML structure, and partials facilitate better organization and maintainability of large stylesheets. Operators enable dynamic calculations directly within the stylesheet, making it easier to manage responsive design and complex layouts.

Advanced features like functions, loops, and conditionals further extend the capabilities of SASS. Built-in functions for color manipulation, numeric operations, and string handling, along with user-defined functions, provide powerful ways to create dynamic styles. Loops, both @for and @each, simplify the generation of repetitive styles and the handling of lists and maps. Nesting these loops and combining them with conditionals allows for the creation of complex, multi-dimensional styles. Conditionals bring logical control flow into stylesheets, enabling styles to adapt based on specific conditions, enhancing flexibility and reducing manual coding efforts.

Overall, SASS and SCSS offer a comprehensive suite of tools that streamline the process of writing and managing CSS. By leveraging these features, developers can create more maintainable, efficient, and dynamic stylesheets, leading to improved development workflows and better-structured projects. The enhanced capabilities provided by SASS and SCSS ultimately contribute to more robust and scalable web applications, making them indispensable tools for modern web development.

References

- https://sass-lang.com/documentation/