Working with SASS

Demo - 1

edureka!

Problem Statement:

In this hands-on demo, you will be introduced to Sass (Syntactically Awesome Style Sheets) - a powerful preprocessor scripting language that enhances the capabilities of CSS. You will explore various features of Sass, including variables, operators, nesting, mixins, parameters, and functions. By the end of this exercise, you will have a solid understanding of how Sass works and how to leverage its functionalities to streamline your CSS development process.

Embark on a hands-on journey to discover the capabilities of Sass (Syntactically Awesome Style Sheets). This comprehensive demo will immerse you in the world of enhanced CSS development, as you delve into various Sass features. Through practical exercises, you will not only understand the core concepts but also witness how Sass can revolutionize your styling workflow.

Solution:

1. Installing Sass:

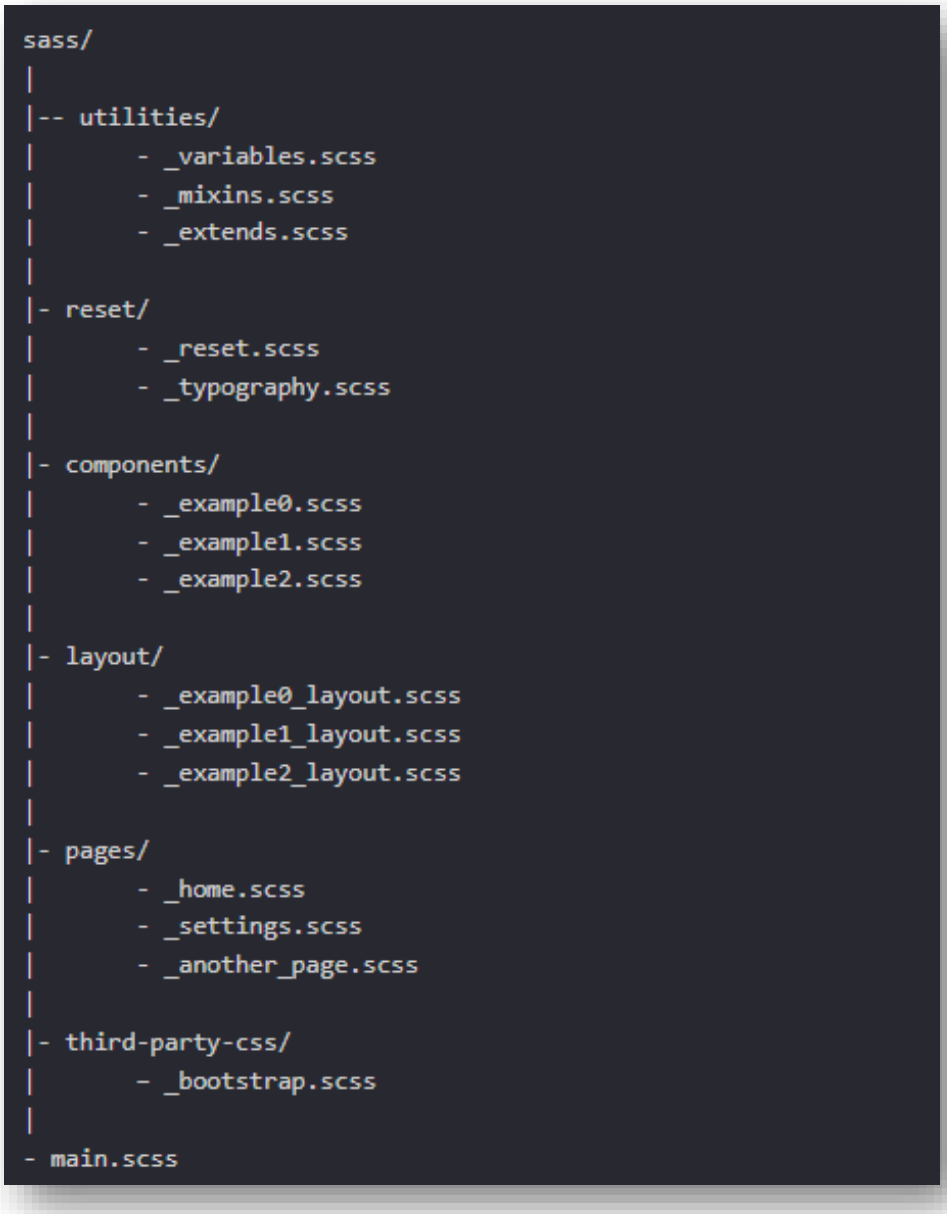
Begin by installing Sass on your development environment. You can use the command-line interface (CLI) to install Sass globally on your system using the following command:

```
npm install -g sass
```

```
//yarn
yarn global add sass
//npm
npm install -g sass
```

2. Creating a Sass File:

Create a new directory for this demo and inside it, create a file named styles.scss. This will be your Sass source file where you will write your Sass code.



```
sass/
|
|-- utilities/
|       - _variables.scss
|       - _mixins.scss
|       - _extends.scss
|
|- reset/
|       - _reset.scss
|       - _typography.scss
|
|- components/
|       - _example0.scss
|       - _example1.scss
|       - _example2.scss
|
|- layout/
|       - _example0_layout.scss
|       - _example1_layout.scss
|       - _example2_layout.scss
|
|- pages/
|       - _home.scss
|       - _settings.scss
|       - _another_page.scss
|
|- third-party-css/
|       - _bootstrap.scss
|
- main.scss
```

3. Using Variables: Define and use variables to store colors, font sizes, or any other values you commonly use in your stylesheets. Create variables for background color, text color, and font size. Use these variables to apply styles to elements in your styles.scss file.

```css
:root {
  --main-color: #000000;
  --main-bg: #FAFAFA;
}
body {
  background: var(--main-bg);
  color: var(--main-color);
}
```

```scss
$main-color: #000000;
$main-bg: #FAFAFA;
body {
  background: $main-bg;
  color: $main-color;
}
```

4. Applying Operators:

Demonstrate the use of operators within Sass to perform arithmetic operations. Calculate and assign a new variable for a width value that is a sum of two existing variables. Apply this calculated width to an element's style.

```
@use "sass:math"

.container
  display: flex

article[role="main"]
  width: math.div(600px, 960px) * 100%

aside[role="complementary"]
  width: math.div(300px, 960px) * 100%
  margin-left: auto
```

- **Arithmetic Operators**

These are the normal arithmetic operators we know.

Operator	Description
+	*Addition*
-	*Subtraction*
/	*Division*
*	*Multiplication*
%	*Remainder*

These operators function the same way as they do in normal arithmetics. However, it is important to note that you cannot perform arithmetic operations on values with different units. We'll use examples to explain this. Let's start with addition + and subtraction-.

Addition and Subtraction: The following are valid operations for addition and subtraction.

```
.header-small {
    /*addition*/
    font-size:  24px + 2px;
    width: 4em + 2;

    /*subtraction*/
    height: 12% - 2%;
    margin: 4rem - 1;
}
```

This is compiled to

```
.box-small {
    /*addition*/
    font-size: 26px;
    width: 6em;

    /*subtraction*/
    height: 10%;
    margin: 3rem;
 }
```

Notice that the values either have the same units or one has no unit at all. The value that has a unit should be on the left-hand side of the operator i.e. it should come first. Let's try to use different units and see the result.

```
.box-big {
    font-size:  22px + 4em;
    // Error: Incompatible units: 'em' and 'px'.
    width: 30% - 20px;
    // Error: Incompatible units: 'px' and '%'.
    }
```

We'll get errors in both cases. It is also possible to add and subtract colors.

Here's an example.

```
$primaryColor: #202020;

 .box {
     background-color: $primaryColor + #123456;
     color: $primaryColor - #100110;
 }
```

This is compiled to

```
.box {
    background-color: #325476;
    color: #101f10;
}
```

How does this work? Sass performs operations on each corresponding part of the RGB

color code. So in the addition part of our code above, here's what happened.

20+12=32(red color) 20+34=54(green color) 20+56=76(blue color)

If you add two color values and it's more than the color range, the result will be the last value on the color range which is #FFFFFF.

Similarly, if you subtract more than the color range, you'll get #000000. I think Sass is kind enough not to throw an error. :) The + can also be used for concatenation as we'll see soon.

Multiplication: In Sass, the multiplication operator is used in the same way as the addition and subtraction only that one value must not have a unit. So here, the code below results to valid CSS when compiled.

```
    .box-small {
        height: 16px * 4;
    }
Its CSS is
   .box-small {
      height: 64px; }

While this one results in an error.
    .box-small {
        height: 16px * 4px; //Error: 64px*px isn't a valid
CSS value.
    }
```

Division: If we use the / operator without putting the values in brackets, it is taken as the normal use of the forward slash in CSS and no operation is carried out. Look at the example below.

```
    .box-medium {
```

```
        font-size: 30px / 5px;
        width:  24px/ 4;
    }
```

This is compiled to

```
.box-medium {
    font-size: 30px / 5px;
    width: 24px/ 4;
}
```

No difference because it is taken as normal CSS. So, to make SCSS do the calculations, we put our values in brackets.

```
.box-medium {
    font-size: (30px / 5px);
    width:  (24px/ 4);


}
```

This will compile to

```
.box-medium {
    font-size: 6;
    width: 6px;
}
```

The operations are carried out. Note that if you are using the / without putting the values in brackets, you can use different units but when they are in brackets, you can only use similar units or no unit on one value.

The remainder operator is used in the same way as the addition and subtraction

operators. Next, let's consider another use of the + operator.

String Operations : The + operator can be used to concatenate strings i.e join two strings together. Some things to note are:

1. If a string with quotes comes before a string without quotes, the resulting string is quoted.
2. If a string without quotes comes before a string with quotes, the result is a string without quotes.

Let's use examples to prove this.

```
   p:before {
       content: "I am a string with" +quotes;
       font: Arial + ", sans-serif";
   }
This is compiled to
   p:before {
       content: "I am a string withquotes";
       font: Arial, sans-serif; }
```

When used with Mixins, it is quite interesting. Here's an example.

```
@mixin introduction($name) {
        &:before {
            content: "I'm a supercool person called " +$name;
        }
    }

    p {
        @include introduction(Sarah);
    }
```

This is compiled to

```
p:before {
    content: "I'm a supercool person called Sarah"; }
```

Just like you can do with other programming languages. Let's consider other operators next.

- **Comparison Operators:** There are operators that can be used to compare one value to another in Sass. They are:

Operator	Conditions	Description
==	x == y	*returns true if x and y are equal*
!=	x != y	*returns true if x and y are not equal*
>	x > y	*returns true if x is greater than y*
<	x < y	*returns true if x is less than y*
>=	x >= y	*returns true if x is greater than or equal to y*
<=	x <= y	*returns true if x is less than or equal to y*

This comparison can be used to help Sass make decisions.

Here's an example.

```
@mixin spacing($padding, $margin) {
    @if ($padding > $margin) {
        padding: $padding;
    } @else {
        padding: $margin;
    }
}

.box {
    @include spacing(10px, 20px);
```

```
    }
```

This is compiled to

```
    .box {
        padding: 20px; }
```

In the above example, we used the > operator to test if the given padding is greater than the margin. The value of the padding is then set based on the return value. The other comparison operators can be used in the same way. Let's move to the final set of operators.

- **Logical Operators:** The logical operators are:

Operator	Conditions	Description
and	x and y	returns true if x and y are true
or	x or y	returns true if x or y is true
not	not x	returns true if x is not true

Let's use an example to explain how they can be used. We are going to use a logical operator to decide which background color should be applied to a button.

```
@mixin button-color($height, $width) {
        @if(($height < $width) and ($width >=35px)) {
            background-color: blue;
        } @else {
            background-color: green;
        }
    }

    .button {
```

```
        @include button-color(20px, 30px)
    }
```

This is compiled to

```
   .button {
       background-color: green; }
```

The background color is set to green because both conditions are not met. If or was used, it would have been set to blue because at least one condition is met.

These operators are all valuable and, when used properly, can make CSS a lot of fun to work with.

5. Nesting for Readability: Utilize nesting in Sass to improve the readability and organization of your styles. Create styles for a navigation menu using nesting to group related styles together.

SASS Nested Rules enables us to write nested rules using the curly braces to enclose the style rules for nested elements just like in HTML. Nested rules make it easier to write CSS code and makes the style rules more readable.

In HTML, we have nested and visual hierarchy. For example, if we create a list, we use the <li> tag to create list items inside the <ul> or <ol> tag. But CSS doesn't support such nested syntax. So if we have to style a list and its items present in the sidebar of a webpage, with the following HTML code:

```
<div id="sidebar">
    <ul>
        <li>Tutorials</li>
        <li>Q & A Forum</li>
```

```
        <li>Flashcards</li>

        <li>Tests</li>

        <li>Collaborate</li>

    </ul>

</div>
```

We will write our CSS code as follows,

```
#sidebar {

    float:left;

    width:25%;

    background-color:lightblue;

    padding: 30px 10px;

}

#sidebar ul {

    list-style-type: none;

    margin: 0;

    padding: 0;

}

#sidebar li {

    padding: 6px;
```

```
    margin-bottom: 10px;

    background-color: #10A2FF;

    color: #ffffff;

}
```

But in SASS/SCSS we can use the nested rules to do so in a more readable way,

```
#sidebar {

    float:left;

    width:25%;

    background-color:lightblue;

    padding: 30px 10px;



    ul {

        list-style-type: none;

        margin: 0;

        padding: 0;

    }

    li {

        padding: 6px;

        margin-bottom: 10px;
```

```
        background-color: #10A2FF;

        color: #ffffff;

    }

}
```

SASS Nested Properties

This is used whenever there are CSS properties that come under same namespaces. Consider background namespace in which we have properties such as:

- background-color
- background-image
- background-repeat
- background-attachment
- background-position

In the case of CSS, we need to type all these properties separately if we want to use them. But SCSS provides us the shorthand nested way which makes work much easier where we write the namespace followed by all the properties i.e. in a nested way.

Let's see an example,

```
body

{

  background:

   {

     color: yellow;

     attachment: fixed;

     repeat: no-repeat;
```

```
    }

}
```

This will be compiled to:

```
body

{

  background-color: yellow;

  background-attachment: fixed;

  background-repeat: no-repeat;

}
```

Similarly in the case of font namespace,

```
.funky

{

  font:

    {

      family: fantasy;

      size: 30em;

      weight: bold;

    }
```

```
}
```

This will be compiled to the following CSS code,

```
.funky

{

  font-family: fantasy;

  font-size: 30em;

  font-weight: bold;

}
```

6. Creating Mixins: Define a mixin for creating box shadows with customizable parameters such as the horizontal offset, vertical offset, blur radius, and color. Apply this mixin to different elements to showcase its reusability.

Mixins are very similar to the extend feature, but allow for parameters to be passed as well. The syntax is a bit different.

```
@mixin flexContainer($direction, $justify, $align) {
  display: flex;
  flex-direction: $direction;
  justify-content: $justify;
  align-items: $align;
}
.main-container {
  @include flexContainer(column, center, center);
}
```

How to Write a Mixin: This is how you write a mixin in Sass:

```
@mixin name {
    properties;
```

```
}
```

And here's how to include it in your code:

```
div {
    @include name;
}
```

Here's an example of using a mixin in your code:

```
@mixin circle {
    width: 200px;
    height: 200px;
    background: red;
    border-radius: 50%;
}

div {
   @include circle;
}
```

Now let's see what's happening in the above code:

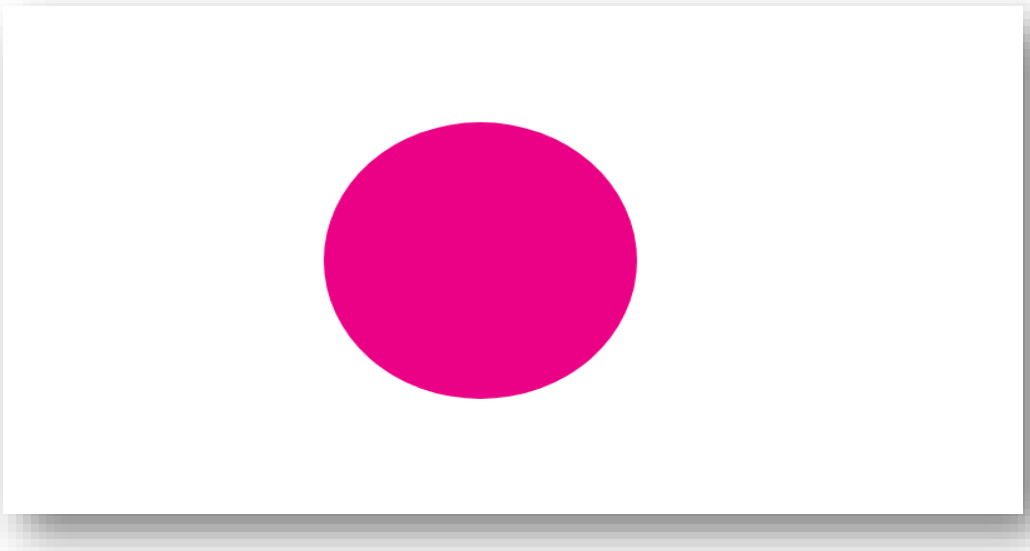
1. First we define a mixin using the @mixin at-rule.
2. Then we give it a name – choose whatever you think will fit what you're gonna be using it for.
3. Add your CSS properties.
4. By simply using @include you pass it to the mixin block.

Mixin Example: Now let's look at an example of a mixin in action.

Here's how to create a pink circle with a mixin:

```
@mixin circle {
    width: 200px;
```

```
    height: 200px;
    border-radius: 50%;
    background: #ea0185 ;
}
.circle {
    @include circle;
}
```



Now, you might ask, "Why should I use a mixin to create a pink circle? I could just give my element a class and style it."

Mixins are reusable, remember? We use them when we know we'll be repeating ourselves a lot. So, the whole point is to *avoid* repetition and keep the code clean.

Passing Arguments

Now that we've seen how to write a mixin, let's move on to the next section. I want to divide this section into smaller parts:

- What are mixin arguments?
- When to pass arguments?
- How to pass arguments? + Examples.

What Are Mixin Arguments?

An argument is the name of a variable which is separated by a comma.

When Should You Pass Arguments to a Mixin?

I'll start this section with an example:

What if you were to create two different circles? Like a green circle and a pink circle?

You could create two separate mixins, one for the green one and one for the pink one:

```
// a mixin for the green circle
@mixin green-circle {
    width: 200px;
    height: 200px;
    border-radius: 50%;
    background: green;
}

// and another mixin for the pink circle
@mixin pink-circle {
    width: 200px;
    height: 200px;
    border-radius: 50%;
    background: pink;
}
```

But this isn't great because you're repeating your code. And we should stick to the DRY (Don't Repeat Yourself) principle, remember?

And that's where mixin arguments come in.

In a regular mixin (and by regular I mean a mixin when no argument is passed) you define some certain styles. But an argument allows you to define different styles by turning them into variables. It's like customizing each style for each element. Let's move on to the next section and see some examples.

How to Pass Arguments to Mixins

We've seen what an argument is and when to use it. And now it's time to see how to pass the arguments:

```
@mixin name($argument,$argument) {
    property: $argument;
    property: $argument;

}
```

By passing arguments you can customize them

Here's an example:

```
@mixin circle2 ($width,$height,$color) {
    width: $width;
    height: $height;
    background: $color;
}
```

You can think of arguments as customizable variables that you can use in different situations to create different things without repeating yourself.

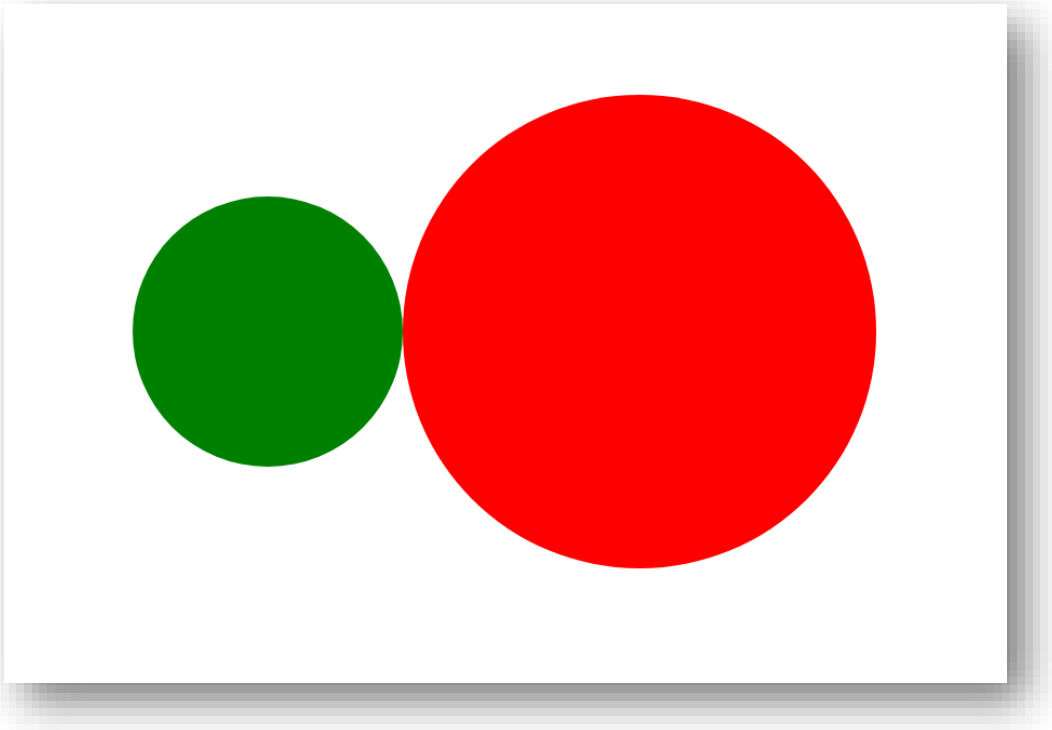
Like when you pass $width to the width property, you can define it in different situations. Maybe you need the width to be 50px in one place and 500px somewhere else.

Does that make sense? Let me break it down for you with another example.

Okay, back to our circles.

I want to make one big red circle and one small green circle (two different things) with just one mixin.

Now what properties do I need to make a circle?



Since we're building circles, the border-radius will be 50% in both situations. So I will leave it alone and won't pass any argument to it.

Now we're down to 3 properties:

1. **width**
2. **height**
3. **background-color**

That means we only need 3 arguments:

```
@mixin circle($width,$height,$color) {
    // We passed $width to the width property
    width: $width;

    // We passed $height to the height property
    height: $height;

    // And we passed $color to width background-color
    background: $color;
```

```
    // no argument for this property, beacuase it's gonna be
the
    // same in both circles
    border-radius: 50%;
}
```

This is how our mixin would look like

So now let's see how we can pass arguments to our mixin:

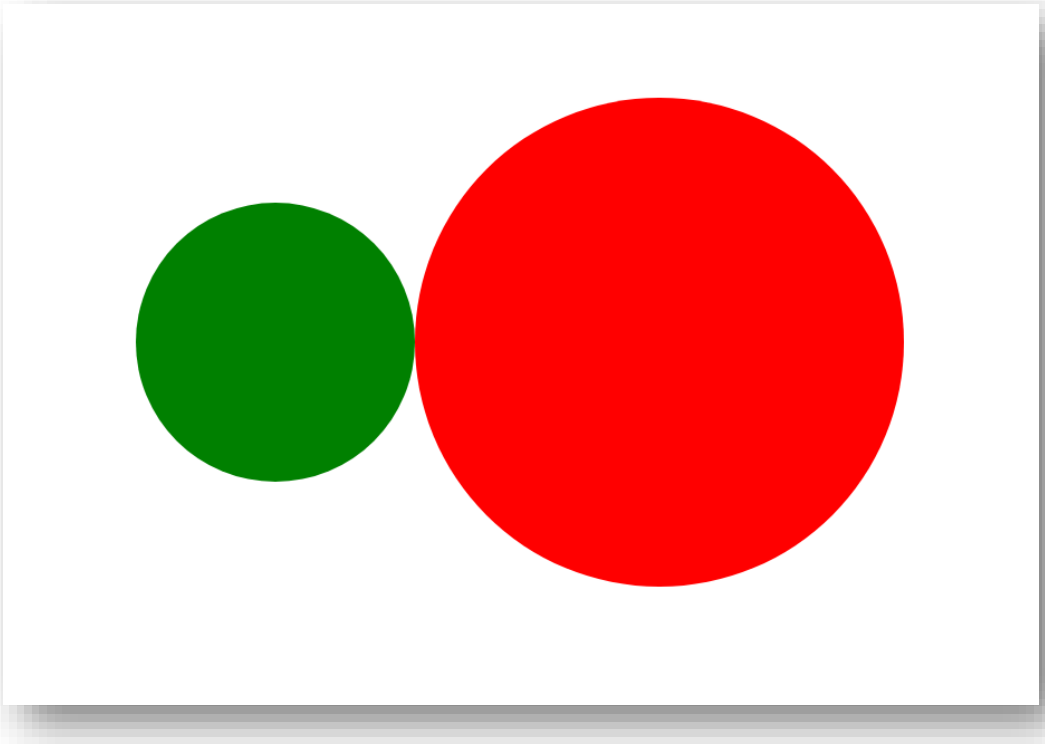
For the big red circle

```
.circle-red {

    // circle ($width,$height,$color);
   @include circle (350px,350px,red);
}
```

For the small green circle

```
.circle-green {

     // circle ($width,$height,$color);
    @include circle (200px,200px,green);
}
```

And here's the result:



8. Implementing Functions: Explore Sass functions by creating a function that calculates the width of a container based on a desired percentage width and a maximum width value. Apply this function to a container element's style.

```
style.scss
$first-width: 5px;
$second-width: 5px;

@function adjust_width($n) {
   @return $n * $first-width + ($n - 1) * $second-width;
}

#set_width { padding-left: adjust_width(10); }
```

You can tell SASS to watch the file and update the CSS whenever SASS file changes, by using the following command –

```
sass --watch C:\ruby\lib\sass\style.scss:style.css
```

Next, execute the above command; it will create the *style.css* file automatically

with the following code –

```
style.css
#set_width {
   padding-left: 95px;
}
```

Output:-

Let us carry out the following steps to see how the above given code works –

- Save the above given html code in **function_directive.html** file.
- Open this HTML file in a browser, an output is displayed as shown below.

In the output, you can see that the left-padding is being applied.

Just like mixin, function can also access globally defined variables and can also accept parameters. You should call the return value for the function by using **@return**. We can call the SASS-defined functions by using keyword parameters.

Call the above function as shown below.

```
#set_width { padding-left: adjust_width($n: 10); }
```

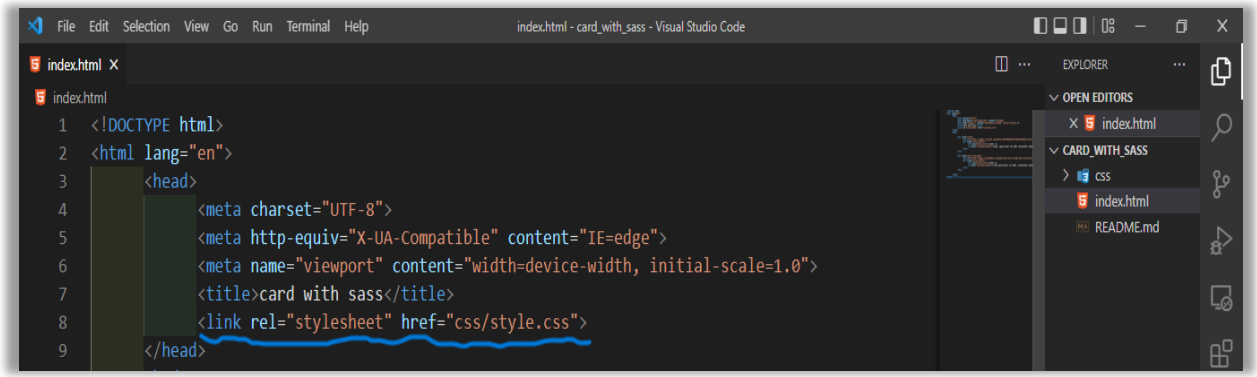
9. Compiling Sass to CSS: Use the Sass CLI to compile your styles.scss file into standard CSS. Run the following command to generate the CSS output:

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

How to Link the CSS File : It's really important to link the CSS file to index.html, to allow the CSS file to apply the CSS styles to the HTML. Otherwise there will be no styling applied and you will only see the code produced by the HTML.

So we will link our CSS file in the index.html file. In my case:

```
<link rel="stylesheet" href="css/style.css">
```



Compiling SCSS Using Command Line : To compile SCSS to CSS from your command line, you will first need to download the sass executable. There are two options you can choose from:

Installing Dart Sass: To install Dart Sass follow these steps:

- Go to Dart Sass GitHub repository and find the newest version at the top (Dart Sass 1.56.1 at the time of writing)

- Find and download the right archive for your operating system under Assets

Unpack the downloaded archive:

```
cd ~/Downloads
tar -xf dart-sass-1.56.1-linux-x64.tar.gz
```

Add the sass executable to your PATH

In my case, I keep my executable scripts inside .local/bin in my home directory.


```
cd ~/Downloads
tar -xf dart-sass-1.56.1-linux-x64.tar.gz
```


Conclusion: In this comprehensive hands-on demo assignment, you've embarked on an exciting journey into the world of advanced CSS development using Sass. Through a systematic exploration of various Sass features, understanding of how this powerful preprocessor can transform your styling workflow.

From setting up your Sass environment to mastering variables, arithmetic operations, nesting, mixins, and functions, you've harnessed the tools that Sass offers to create more efficient and maintainable styles. By compiling your Sass code into standard CSS and integrating it seamlessly with HTML, you've witnessed the tangible impact of your efforts in the browser.