%%

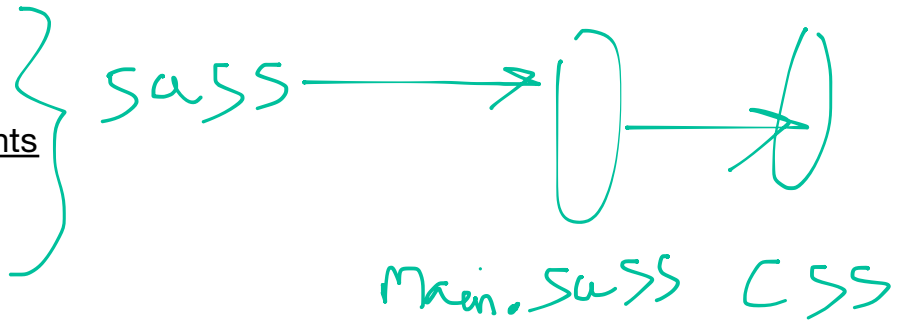
7iii Part #3 - Streamline your code using advanced Sass techniques

%%

1. Use the 7-1 pattern for a manageable codebase

- **video notes**
  - -> clean code
  - -> moving all over the place -> variables which are at the top of the Sass file
  - -> <u>putting mixins and variables in different places</u>
- **notes on the text below the video**
  - A place for everything and everything in its place
    - -> clean code
    - -> how to organise all of the different components
    - -> <u>you can split up sections of the Saas into different documents</u>
      - -> a load of Saas files for the same project
      - -> each containing a different type of code -> mixins, variables etc
  - The Seven Dwarfs: the 7-1 file system
    - about the 7-1 file system

- -> you have a css style sheet
- -> there is one Sass file which comes off of that
- -> that one Sass file <- if you change the code in there, then it compiles into the css
- -> one css file can have one Sass file linking to it o.e
- -> you take the main css file and link a master Sass file to it
- -> then that one Sass file linked to the one css file
- -> 7-1 <- that Sass file - it has 7 smaller Sass files which funnel into it
  - 1 Base
  - 2 Utils
  - 3 Layout
  - 4 Components
  - 5 Pages
  - 6 Themes
  - 7 Vendors



- how to make it
  - -> command shift p in VSCode
  - -> then terminal
  - -> then use linux to mkdir them
  - -> you can "mkdir base utils layout components pages themes vendors" <- you make multiple directories using mkdir in one line
- what each directory is for <- directory, aka base.scss etc
  - 1 Base
    - -> typography
    - -> things which are basic
    - -> they form the base of the site <- styles which form site wide
  - 2 Utils
    - -> everything which goes into extensions, it's utilities
      - variables
      - functions
      - mixins
      - -> %placeholders <- mixins / classes which aren't being used
  - 3 Layouts
    - -> things which can be used for layouts
    - -> using the BEM naming convention
    - -> e.g forms / headers
  - 4 Components
    - -> BEM blocks which are more self contained, e.g buttons
    - -> entire components rather than site-wide layouts
    - components vs layouts
      - -> a component of a webpage is like one part of it, more specific
        - -> a component of it
        - -> and then a layout is more to do with the entire thing
      - -> e.g if you have a search bar <- layout
        - -> the components are - a form, divs and button
        - -> components are the smaller parts and layouts are the smaller parts coming together to create one of the elements on the page
  - 5 Pages
    - -> some webpages on the site use elements which aren't present anywhere else
    - -> e.g a homepage having a part which isn't used elsewhere on the site, styles for those parts
    - -> unlike buttons e.g
    - -> styles which are only used on a single page

- 6 Themes
  - -> theme.scss files for how the site should look e.g on Christmas / Halloween - which aren't there all of the time, but which would be applied on occasion
- 7 Vendors
  - -> where code from elsewhere which you have taken goes
    - -> code you didn't write
  - -> Bootstrap, jquery UI,

- Cleaning house
  - the 7:1 file structure for Saas
    - -> **it's not that you have seven .scss files linking to one main scss file which links to the css file it generates**
    - **-> it's that you have seven folders -> 7 file directories created using the mkdir in the terminal using VSCode**
      - **-> then each of those folders contains .scss file**
      - -> then all of those scss files link back to the main one
      - -> then the main one links to the .css file
      - -> which then gets compiled with the index. html files
      - -> which makes the webpage
  - partials
    - -> in each of the directories -> each of those seven folders - you have the .scss files
    - -> then all of those partial scss files are combined into one larger one
    - -> a partial scss file is named _variables.scss e.g <- with an - one - underscore before it's name, it's called "a partial"
      - so each of the directories can have a _variables.scss file -> they are all partial, and there can be seven of them
      - -> then they are combined / imported into the main scss file -> which compiles them into the main css file for the project
    - to create one (a 'partial')
      - -> he _names-it.scss, then opens in in VSCode
      - -> then he pastes lines of Sass into it from the main one
      - -> you have to
        - -> partials are all named in a specific way
        - -> you then have to take the partials (the _partial-name.scss files) and import them all back into the main scss file <- that one file, the codebase then links back into the css file for the project, to compile it using the Saas
    - to import a partial (smaller Sass file) back to the main .scss file
      - -> in the main scss file
        - @import "./"; <- inside the ""'s, it's the name of the file where the partial is stored
          - the . is a a relative file path to the partial
          - -> in a browser then the partial name will begin with an _
          - -> but in the IDE it's only showing -> in that line of code where it's been imported, you can't see the _ (but it's still there)
          - -> separating out the styles from the main Saas file like this -> into a separate partial which is then imported back into the main Saas file produces the main webpage in this case
    - he repeats the process to create a partial
      - -> he makes a new file -> _name-of-file.scss
        - the folder which that file is in depends on the nature of the code which we're planning on putting into it -> and taking out of the main.scss document
      - -> then he takes Sass out of the main.scss file and pastes it into the partial

- -> then he imports the partial (the name of it) back into the main scss document
  - you have to be careful about the order which you import a given partial
  - -> you are importing the partials into the saas file at the top of it
  - -> the order you import them in matters, or you can get compilation errors
  - ***-> you have to import saas files into other sass files according to the order in which the partials were defined***
    - ***-> the order which you import sass files into it is***
      - ***1 Utils <- variables, functions, mixins, placeholders***
      - ***2 Vendor sheets <- sheets which come from elsewhere that you import in, they come first because you don't know how they respond***
      - ***3 Base***
      - ***4 Components***
      - ***5 Layout***
      - ***6 Pages***
      - ***7 Themes***
  - ***-> the main.scss file should only contain these imports, like this ->***
    - ***the main.scss file is like the contents of the scss partials***
    - ***-> and there are seven smaller file directories (the 7 dwarfs)***

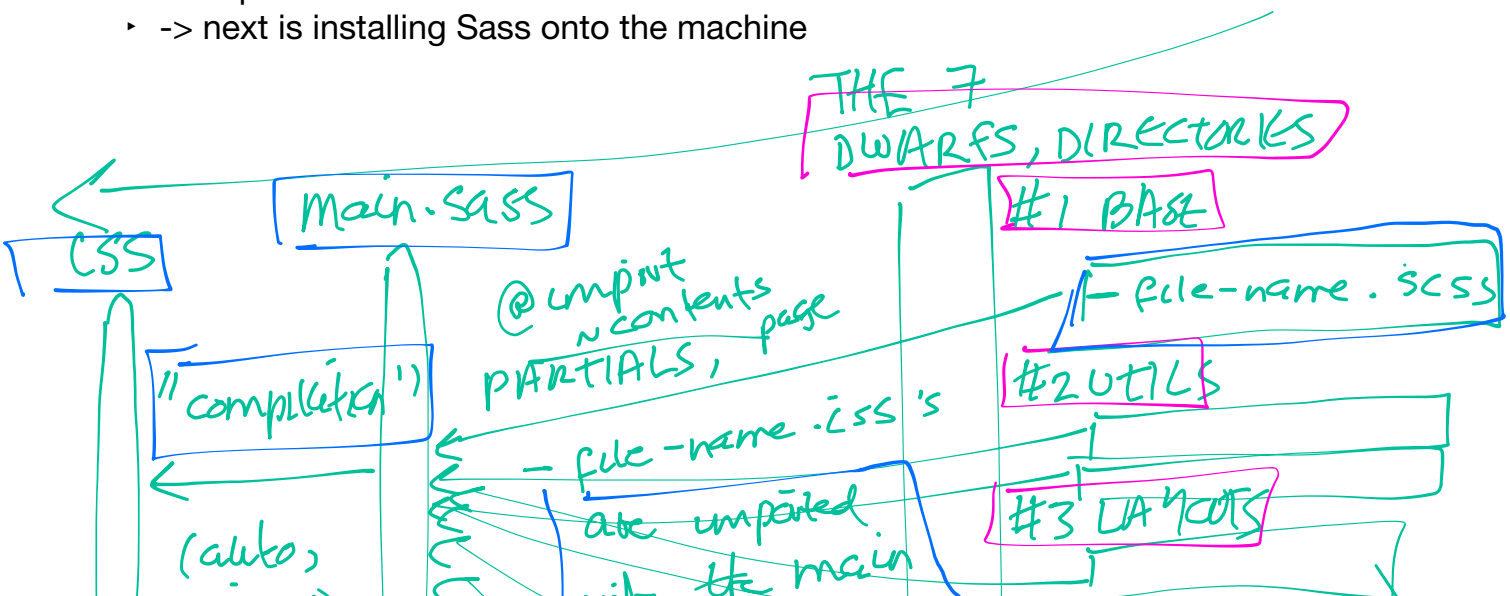
```
@import "./utils/variables";
@import "./utils/functions";
@import "./utils/mixins";
@import "./utils/extensions";

@import "./base/base";
@import "./base/typography";

@import "./components/buttons";

@import "./layouts/header";
@import "./layouts/nav";
@import "./layouts/container";
@import "./layouts/form";

@import "./pages/work";
@import "./pages/about";
@import "./pages/project";
```

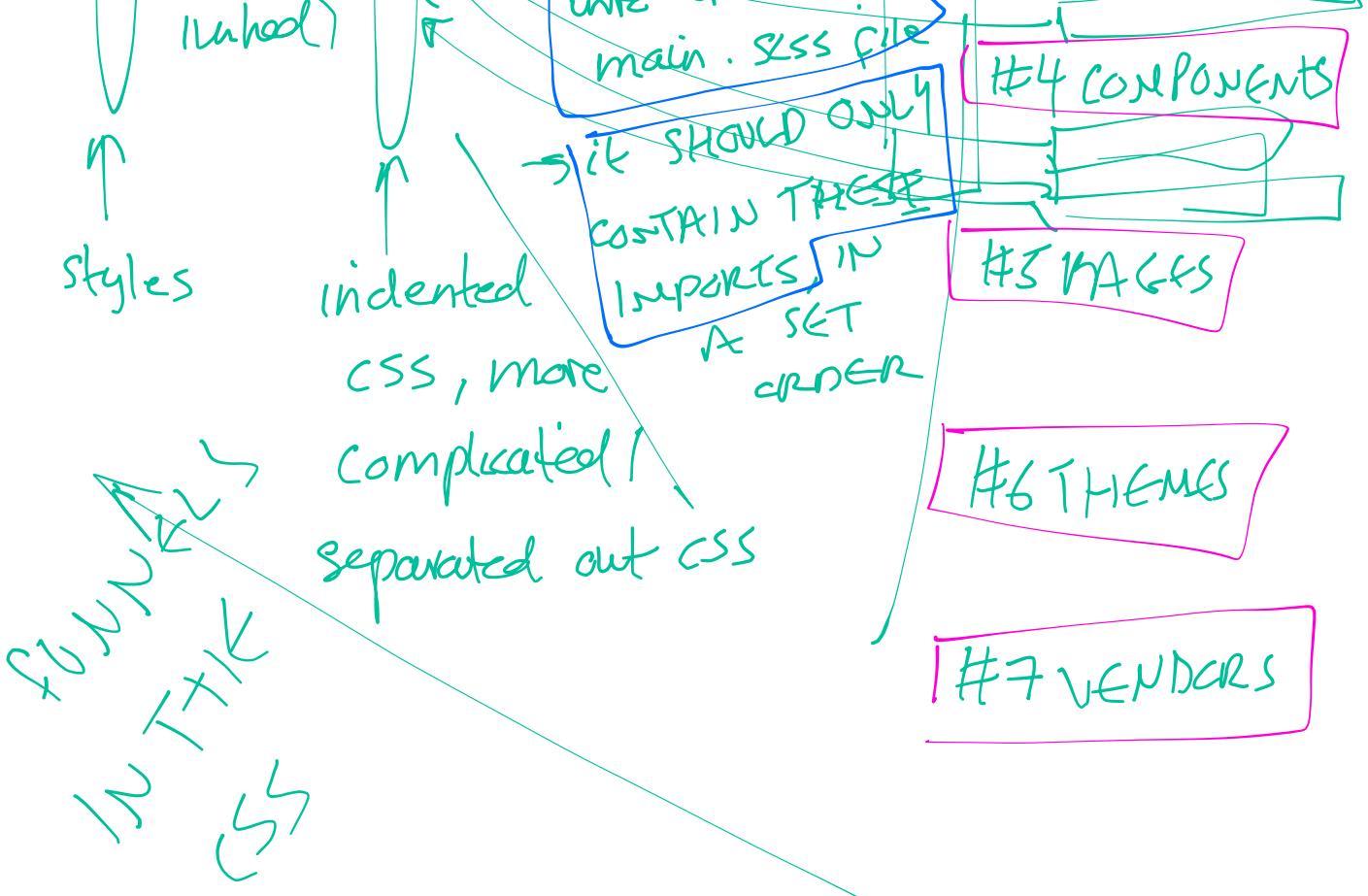
- breaking down this example Sass (parts of it which I don't understand)
  - .form {
  - width: 100%;
  - padding-bottom: $grid-gutter;
  - &__heading { <- **#1 why the & is required, #2 what the name of the css after the indentations is**
  - width: 100%;
  - color: $colour-white;
  - @include heading-shadow; <- **#3 why the @include was written here, what it meant again**
  - background: $colour-primary;
  - line-height: 5rem;
  - padding: $grid-gutter;
  - }
  - &__field {
  - & label {
  - color: $colour-accent;
  - display: block;
  - font-size: map-get($txt-size, label);
  - line-height: map-get($txt-size, label);
  - padding-top: $grid-gutter;
  - }
  - & input {
  - width: 100%;
  - background: $colour-secondary;
  - @include border;
  - padding: $grid-gutter;

```
            color: $colour-accent;
        }
        & textarea {
            width: 100%;
            color: $colour-primary;
            background: $colour-secondary;
            @include border;
            outline: none;
            padding: $grid-gutter;
            margin-bottom: $grid-gutter*0.5;
            &:active, &:focus {
                color: $colour-accent;
            }
        }
    }
}
```

- -> asking my questions about the code into OpenAI
  - why the &__ is required
    - The &__ syntax in CSS and preprocessors like Sass is used to denote a BEM (Block Element Modifier) naming convention. BEM is a methodology for naming and organizing CSS classes in a way that makes it clear and easy to understand the structure of your HTML.
    - Here's a breakdown of the BEM syntax:
      - Block (&): Represents the higher-level component or module. In your case, it's likely the form itself.
      - Element (__): Represents a component or part of the block. For example, the heading within the form is considered an element of the form block.
    - -> So, when you see &__heading, it indicates that the heading is an element of the block represented by the & (which refers to the current selector, in this case, .form).
    - -> This naming convention helps in creating more maintainable and modular styles, especially in larger projects where it's important to avoid conflicts and have a clear structure. It also makes it easier to understand the relationship between different parts of your HTML structure and their corresponding styles.
      - -> BEM <- the E is an element of html
  - indented elements
    - The indented elements in your code are part of a nested structure and follow a specific selector hierarchy. In CSS and Sass, these indented elements are often referred to as "nested selectors" or "nested rules." Each level of indentation signifies a deeper level of specificity within the styling structure.
    - In your provided code:
      - .form is the outer block or component.
      - &__heading is an element nested inside .form.
      - &__field is another element nested inside .form.
      - & label, & input, and & textarea are elements nested inside .form__field.
    - So, you can say that &__heading, &__field, & label, & input, and & textarea are nested selectors within the overall .form selector. This nested structure helps in organizing styles and understanding the

relationships between different parts of your HTML structure.
    - -> nested selectors
- why the @include was used
    - In Sass, the @include directive is used to include a mixin in a selector. Mixins are a way to reuse and bundle together a set of CSS properties and rules, making your styles more modular and maintainable.
    - In your provided code snippet, you have the following line:
        - scss
        - @include heading-shadow;
    - This line is including a mixin called heading-shadow into the styles for the &__heading element. The heading-shadow mixin likely contains a set of CSS properties that apply a shadow effect to the heading
    - Here's a simplified example to illustrate the concept:
    - @mixin heading-shadow {
    - text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.5);
    - // other properties...
    - }
    - .form {
    - &__heading {
    - // other styles...
    - @include heading-shadow;
    - }
    - // other styles...
    - }
        - -> to define a mixin, it's @mixin mixin-name
        - -> to call a mixin, it's @include mixin-name
    - In this example, the heading-shadow mixin defines a text-shadow property, and it's included in the styles for the .form__heading element. This allows you to reuse the shadow effect across different parts of your styles.
    - Using mixins with @include helps keep your code DRY (Don't Repeat Yourself) and allows you to make global changes easily by modifying the mixin definition.
        - -> you can reuse mixins across different sass files, if you import the name of the css file which contains that mixin into the partial you are using at the top of it
        - -> then call it in the code by using @include
- Let's recap!
    - -> next is installing Sass onto the machine





→ You have to import partials in a very specific order, ∵ they are compiled in the order you import them

→ main.scss is like a contents page full of partials.scss imports - each of these is a sass file - so when you import them in → they then get compiled into css in the order you import them into the main.scss file - which converts to css

→ and the further up an el is in that document, THE HIGHER THE SPECIFICITY IT HAS

2. Install Sass locally

- **video notes**
  - -> installing Sass onto a computer
  - -> you can run it from a codepen - you need it locally in case there is no internet
- **notes on the text below the video**
  - Downloading the project files
    - -> you need locally it in case
      - -> there is no internet and you want to compile Sass into css
      - -> how the project files are organised
        - **it's one website -> each html file is a page on that website**
        - **-> and then there is another directory, which is specifically for sass**
  - Getting Sassy in Three Easy Steps!
    - open a terminal in VScode with the project files - aka, the Sass file directory already setup, with the html for the different webpages and all of the files linked together - when you have that done, then open a terminal in VSCode
      - -> open the project files in VSCode
      - -> open a terminal there -> command shift p for the colour palette
    - -> then install - he's going through a Saas installation - type this into the terminal in the project files under VSCode
      - npm init
        - -> initiating an npm package.json file
        - -> this file stores information about the project
        - -> scripts <- json scripts for npm to run the site
        - -> this installs libraries, aka dependancies
        - -> npm to reuse other peoples' code
        - -> then the terminal is asking for information, if you enter through it then you have the default values
        - -> you just have to enter through all of them
        - -> that's how you create a package.json file in the project file directory, which you can see in VSCode
        - this is inside the json file that creates ->
          - so, this is for using other peoples' code / dependencies with npm
          - -> when you install packages -> they appear there within the dependancies objects

    - -> then you install scss
      - -> in the terminal in VSCode
      - -> npm install sass -g
        - -> -g is called the flag
        - -> it's installing it globally
        - -> node js is a server framework

        - installs packages
        - executes scripts
        - manages dependencies
    - -> sass is a package which it's installing
        - -> you can also get chatgpt to help you with the installation
- -> sass --version <- to see if it's been installed
- How do you turn this thing on?
- -> after Sass has been installed with npm
    - node package manager

nothing would happen. Within the quotes type the following:

```
sass --watch ./sass/main.scss:./css/style.css
```

json

```
{
  "name": "writting-sass",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "sass": "sass --watch ./sass/main.scss:./css/style.css"
  },
  "author": "",
  "license": "ISC",
}

```

    - -> what this line does
        - this json file is what npm created when sass was being installed in terminal
        - -> npm is the tool which allows us to use external packages in the project
        - -> inside the json file it created <- that's where we've added this line
        - -> what it does
            - -> sass <- it's telling npm to look in the sass package which was installed - to find the file to compile with it
            - -> --watch <- a flag, - is an option, --is a flag
                - to watch for changes in the Sass file
                - -> it automatically updates
                - -> it's like running latex in overleaf and turning on the auto-compile
                - -> it checks for updates to the file without you having to do anything
            - -> the next one is the directory of the Saas file
            - -> : <- from the source to the destination paths
            - -> then the last one is telling it where to compile the CSS to
    - -> after editing the json file which npm created
        - save it and back out of it
        - -> command shift p -> open a terminal
        - -> then npm run sass
            - this is running Sass in the terminal for VSCode
            - -> don't put comments in json files, it will mess them up
            - -> this is into the terminal
                - execute an npm command
                    - -> run is the command in npm which you want it to run
                    - -> then sass is the name of the script which you want to run
                    - -> then when it runs, its compiling the main.scss into css
    - -> when scss is running in the VSCode terminal with all the project files
        - -> control C in the terminal stops the process from running
        - -> it's running in the terminal and is watching for changes

        - whenever those changes are made, they show in the command line - running at the bottom of the IDE
        - -> it's showing errors when they run
        - -> the answer to my earlier question -> Sass is verifying itself
            - but you have to verify the html and css using the w3c validators
            - -> because it's automatically recompiling the Sass -> into css
- Compiler Modes
    - -> **when you go from Sass to css -> there are four different modes it uses to compile this, these are called compiler modes**

```scss
.form {
  width: 100%;
  padding-bottom: 1.5rem; }
  .form__heading {
    width: 100%;
    color: #fff;
    text-shadow: 0.55rem 0.55rem #11af82;
    background: #15DEA5;
    line-height: 5rem;
    padding: 1.5rem; }
  .form__field label {
    color: #D6FFF5;
    display: block;
    font-size: 2rem;
    line-height: 2rem;
    padding-top: 1.5rem; }
  .form__field input {
    width: 100%;
    background: #001534;
    border: 0.1rem solid #15DEA5;
    padding: 1.5rem;
    color: #D6FFF5;
    font-weight: 900;
    font-style: italic;
    font-size: 2.75rem; }
  .form__field textarea {
    width: 100%;
    color: #15DEA5;
    background: #001534;
    border: 0.1rem solid #15DEA5;
    outline: none;
```

Nested mode

```scss
.form {
  width: 100%;
  padding-bottom: 1.5rem;
}
.form__heading {
  width: 100%;
  color: #fff;
  text-shadow: 0.55rem 0.55rem #11af82;
  background: #15DEA5;
  line-height: 5rem;
  padding: 1.5rem;
}
.form__field label {
  color: #D6FFF5;
  display: block;
  font-size: 2rem;
  line-height: 2rem;
  padding-top: 1.5rem;
}
.form__field input {
  width: 100%;
  background: #001534;
  border: 0.1rem solid #15DEA5;
  padding: 1.5rem;
  color: #D6FFF5;
  font-weight: 900;
  font-style: italic;
  font-size: 2.75rem;
}
.form__field textarea {
  width: 100%;
```

Expanded mode - looks like someone wrote the css, rather than it having ben compiled from mail.saas

        - -> it's taking the Sass and converting it into css
        - -> this is what it's doing when it's open like that in the terminal
        - -> so you can look through the css file -> you know it came from the main.saas file <- and you know the order of imports from that file
        - -> so you know where about on the compiled css file the

```scss
.form { width: 100%; padding-bottom: 1.5rem; }
.form__heading { width: 100%; color: #fff; text-shadow: 0.55rem 0.55rem #11af82; background: #15DEA5; line-height: 5rem; padding: 1.5rem; }
.form__field label { color: #D6FFF5; display: block; font-size: 2rem; line-height: 2rem; padding-top: 1.5rem; }
.form__field input { width: 100%; background: #001534; border: 0.1rem solid #15DEA5; padding: 1.5rem; color: #D6FFF5; font-weight: 900; font-style: italic; font-size: 2.75rem; }
.form__field textarea { width: 100%; color: #15DEA5; background: #001534; border: 0.1rem solid #15DEA5; outline: none; padding: 1.5rem; margin-bottom: 0.75rem; }
```

Compact mode -> one css ruleset per line

```scss
.form{width:100%;padding-bottom:1.5rem}.form__heading{width:100%;color:#fff;text-shadow:.55rem .55rem #11af82;background:#15DEA5;line-height:5rem;padding:1.5rem}.form__field label{color:#D6FFF5;display:block;font-size:2rem;line-height:2rem;padding-top:1.5rem}.form__field input{width:100%;background:#001534;border:0.1rem solid #15DEA5;padding:1.5rem;color:#D6FFF5;font-weight:900;font-style:italic;font-size:2.75rem}.form__field textarea{width:100%;color:#15DEA5;background:#001534;border:0.1rem solid #15DEA5;outline:none;padding:1.5rem;margin-bottom:.75rem}
```

Compressed mode <- no whitespace / line breaks at all. This is a minified css file (common for faster final webpages)

    - code for what you want will be
    - -> because one literally came from the other -> one if like the contents page of the other
- Flying our flags: compiling in compressed mode
    - -> **you can change the sass compile mode in the json file which links the saas to the css file -> in this example, the line of code which they've used has a new style flag to do this:**
        - **"sass": "sass --watch ./sass/main.scss:./css/style.css --style compressed"**

        - -> then in the terminal running the Sass
            - -> crtl C <- ends the current process
            - -> npm run sass <- to restart the sass up in the terminal with the new json
    - -> he's changed the compile mode into compressed <- and the entire css is visible in this case
    - -> if you change the compiler mode (i.e how you get from sass to css), this has to be done in the json file which links them
        - -> you have to restart the terminal for this -> not for changing the Sass files themselves, but when you change the json file then you need to restart the terminal to see the changes
  - You made it!
    - -> installing packages
    - -> more time is spent reading code than writing it, it needs to be organised
    - -> next chapters is
        - -> advanced Sass datatypes
        - -> how to use them

3. Integrate advanced Sass data types

- **video notes**
  - -> variables <- datatypes in Saas
  - -> lists and maps
  - -> this chapter is lists and maps
- **notes on the text below the video**
  - List-making and cartography

Produce Maintainable CSS With Sass

15 hours Medium

Last updated on 3/13/23

Integrate advanced Sass data types

00:30

List-making and cartography

Remember way back when you were learning about variables? We briefly covered the data types available within Sass, and I told you not to worry too much about lists and maps because we'd be covering them in depth in just a bit? Well, it's been a bit, so that must mean it's list and map time!

The Sass variables we've covered so far only store single values: a color, a size, etc. That's fine for a lot of uses, but what about something like padding? You can assign four different values in a single rule: `scss`

```scss
.block {
    padding: 1rem 2rem 3rem 4rem;
}
```

Creating a variable for each side's padding size would be way more work than it's worth. But, what about a variable that stored all four dimensions in a single variable?

scss

```scss
$padding-dimensions: 1rem 2rem 3rem 4rem;

.block {
    padding: $padding-dimensions;
}
```

Wait! Does that actually work?! Let's check out the compiled CSS:

css

```css
.block {
  padding: 1rem 2rem 3rem 4rem;
}
```

What devilry is this?!

`$padding-dimensions` is what Sass calls a **list,** which is a list of values. They allow you to group values together in a single variable. In this case, we've made a list of dimensions to use as values for a padding property.

To make a list, you just need to define a variable and fill it with values. The syntax for writing them is extremely flexible. You can separate list items with spaces, like we've done with `$padding-dimesions`, or use commas to separate them:

scss

```scss
$syntax-01: 1rem 2rem 3rem 4rem;

$syntax-02: 1rem, 2rem, 3rem, 4rem;
```

And you can write them with or without parentheses:

scss

```scss
$syntax-01: 1rem 2rem 3rem 4rem;

$syntax-02: 1rem, 2rem, 3rem, 4rem;

$syntax-03: (1rem 2rem 3rem 4rem);

$syntax-04: (1rem, 2rem, 3rem, 4rem);
```

All of them will compile identically:

scss

```scss
$syntax-01: 1rem 2rem 3rem 4rem;

```

css

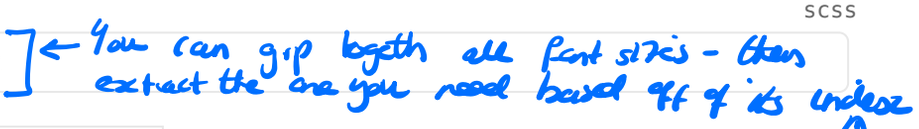
```css
.syntax-01 {
  padding: 1rem 2rem 3rem 4rem;
```

```scss
$syntax-02: 1rem, 2rem, 3rem, 4rem;

$syntax-03: (1rem 2rem 3rem 4rem);

$syntax-04: (1rem, 2rem, 3rem, 4rem);

.syntax-01 {
    padding: $syntax-01;
}
.syntax-02 {
    padding: $syntax-02;
}
.syntax-03 {
    padding: $syntax-03;
}
.syntax-04 {
    padding: $syntax-04;
}
```

```scss
}

.syntax-02 {
  padding: 1rem, 2rem, 3rem, 4rem;
}

.syntax-03 {
  padding: 1rem, 2rem, 3rem, 4rem;
}

.syntax-04 {
  padding: 1rem 2rem 3rem 4rem;
}
```

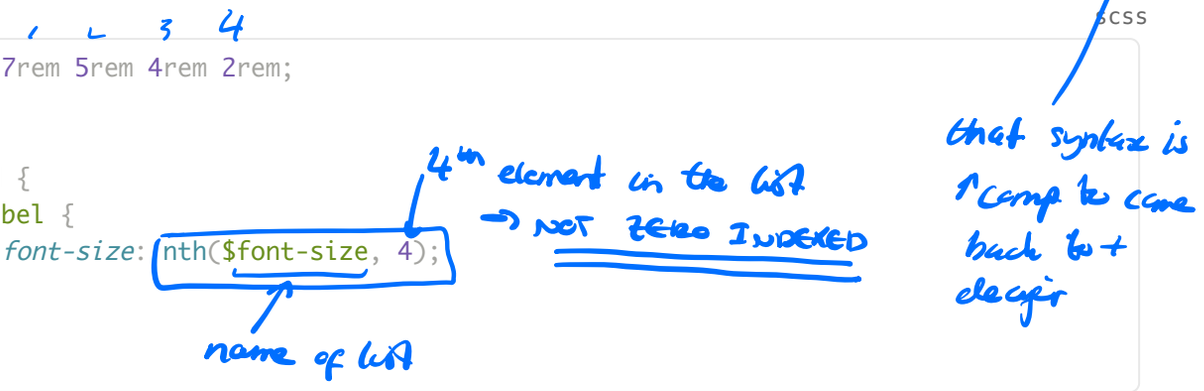
Just because you choose to store a group of values in a list, doesn't mean that you want to use them all in the same instance. You can also use the values from list items individually. Let's make a list of the various font sizes we are using throughout the site:

```scss
$font-size: 7rem 5rem 4rem 2rem;
```



Now let's use the `2rem` value from `$font-size` to populate the value of our labels' font size. To access individual values from a list, call the `nth()` function with the name of the list and the index of list item:

```scss
$font-size: 7rem 5rem 4rem 2rem;

.form{
    &__field {
        & label {
            font-size: nth($font-size, 4);
        }
    }
}
```



We want to use the `2rem` value from `$font-size` on our form's label's `font-size`, so call `nth($font-size, 4)`, since `2rem` is the fourth value in `$font-size`.

> Most other programming languages have lists that start with an index of zero. If you're accustomed to other programming languages, it might feel a bit weird. Be forewarned!

Lists make more maintainable code by allowing you to group associated values together. But they can be a bit tough to read, especially coming back after being away for a while. Looking at a list and trying to

remember the uses of its items can take some head-scratching and scrolling. In a few months or even a few minutes, you will probably forget the indices of the different values in `$font-size`.

Map making

Lists can be difficult to read and remember because there's no real context to list contents; just values grouped together. That's why you have Sass maps! They are a lot like lists, but give each value a name in the form of **key/value pairs:**

scss

```scss
$font-size: (logo:7rem, heading:5rem, project-heading:4rem, label:2rem);
```

By giving each value a **key,** or name, it makes figuring out and remembering its purpose a lot clearer. And clearer is easier. And easier is good. 👍

The rules for writing maps are much stricter than those for writing lists. With lists, just about everything is optional. But maps must be **surrounded by a set of parentheses,** and must **use commas to separate the key/value pairs:**

scss

```scss
$map: (
    key-01: value-01,
    key-02: value-02,
    key-03: value-03
);
```

Accessing the value of a map is a bit different than the `nth()` function for lists. With maps, you need to call the `map-get()` function:

scss

```scss
$font-size: (logo:7rem, heading:5rem, project-heading:4rem, label:2rem);

.form{
    &__field {
        & label {
            font-size: map-get($font-size, label);
        }
    }
}
```

`map-get()` requires two **arguments:** the first is the **name of the map,** and the second is the **name of the key.** The end result will still be 2rem compiled as the font size in the CSS:

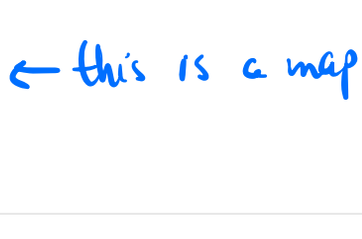
css

```css
.form__field label {
  font-size: 2rem;
}
```

Let's put our newly found map-making skills to the test and create a map named `$input-txt-palettes`, which contains the color palettes for our text inputs. Group the palettes by their state, and create a key for each pseudo-class: `active`, `focus`, and `invalid`:

scss

```scss
$colour-primary: #15DEA5;
$colour-secondary: #001534;
$colour-accent: #D6FFF5;
$colour-white: #fff;
$colour-invalid: #DB464B;

$txt-input-palette: (
    active: ,
    focus: ,
    invalid:
);
```



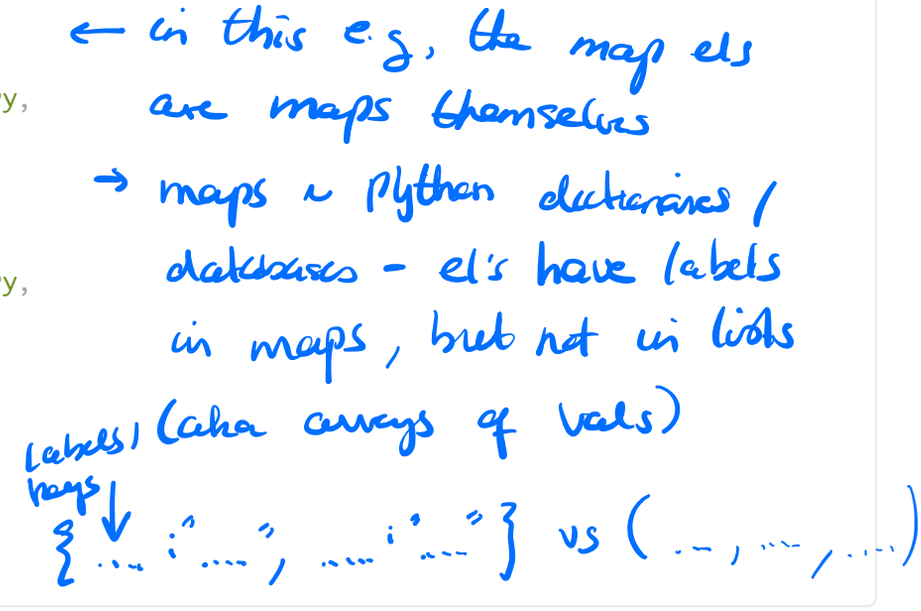
Now you have keys, but they're empty; they need values assigned. Fill the keys with your color palettes, which will require nesting another map for the colors within each of the keys.

That's right, maps inside of maps! 🗺️🗺️🗺️

The values in maps (and lists) can be any valid Sass data type, maps included. Your text input states need to store three separate colors, so each key will contain a map with three values: the state's border, background, and text colors:

scss

```scss
$colour-primary: #15DEA5;
$colour-secondary: #001534;
$colour-accent: #D6FFF5;
$colour-white: #fff;
$colour-invalid: #DB464B;

$txt-input-palette: (
    active: (
        bg: $colour-primary,
        border: $colour-primary,
        txt: $colour-white,
    ),
    focus: (
        bg: $colour-primary,
        border: $colour-primary,
        txt: $colour-white,
    ),
    invalid: (
        bg: $colour-invalid,
        border: $colour-white,
        txt: $colour-white,
    )
);
```



Now you have a variable that contains all of the color information for the various states of your text inputs, in a format that is easily read and remembered. And should you need to update any of the schemes or values, they're all centralized in a single, easy-to-locate block.

Mix'n'map

You have all of the palette info stored waiting to be used. To make things a bit easier on you, let's make a mixin to deploy the palettes. You need to assign rules for the `border` , `background-color` , and

text `color` properties, with the appropriate values from `$txt-input-palette` .

scss

```scss
@mixin txt-input-palette {
    border: .1rem solid $border;
    background-color: $bg;
    color: $txt;
}
```

To declare which state (hover, etc.) to pull the map from, you need to assign an argument for the state that you can pass into the mixin:

scss

```scss
@mixin txt-input-palette($state) {
    border: .1rem solid $border;
    background-color: $bg;
    color: $txt;
}
```

With the name of the state, you can store its palette map in a variable name `$palette` :

scss

```scss
@mixin txt-input-palette($state) {
    $palette: map-get($txt-input-palette, $state);
    border: .1rem solid $border;
    background-color: $bg;
    color: $txt;
}
```

Now `$palette` contains a map of the `bg` , `border` , and `txt` color values of the assigned state. You can use it with the `map-get()` function to populate the color values of your ruleset:

scss

```scss
@mixin txt-input-palette($state) {
    $palette: map-get($txt-input-palette, $state);
    border: .1rem solid map-get($palette, border);
    background-color: map-get($palette, bg);
    color: map-get($palette, txt);
}
```

Now you have a mixin that you can use to style all of the text input elements and pseudo-selectors by simply passing the state when you apply the mixin! This helps keep your code maintainable and easily modifiable in the future.

Let's apply the mixin to your default, inactive `form__txt input` selector:

scss

```scss
@mixin txt-input-palette($state) {
    $palette: map-get($txt-input-palette, $state);
    border: .1rem solid map-get($palette, border);
    background-color: map-get($palette, bg);
    color: map-get($palette, txt);
}
```

```
.form {
    &__field {
        & input {
            @include txt-input-palette(focus);
        }
    }
}
```

Now you should see the selector `.form__field input`, populated with rules for the `border`, `background-color` and text `color` in the compiled CSS:

css

```
.form__field input {
  border: 0.1rem solid #15DEA5;
  background-color: #001534;
  color: #15DEA5;
}
```

Perfect!

Try it out for yourself!

In this interactive exercise, let's give our CSS some structure by using maps! Currently, you have a .btn selector and three buttons in different colors.

Generate our buttons' selectors with a map!

- Declare a new variable with the name of `$btn-mods` and place a set of parentheses after the colon to contain the map keys and values.
- Create a key for the modified pink button called “pink” and give it a value of `$color-secondary`
- Create a key for the modified blue button called “blue” and give it a value of `$color-tertiary`
- Replace the uses of `$color-secondary` and `$color-tertiary` with the values from the map, using the `map-get()` function

Review the rendered page to ensure the buttons still appear as they should - you can check my solution as well using this CodePen.

Now you have a mixin that you can use to create the properties for all the different states of your text inputs; or, you could automate it. Because that’d be a lot less tedious - and a whole lot cooler.

And that’s just what we’ll be doing next chapter!

Let's recap!

- **Lists** and **maps** are collections of values.
- Lists have a really **flexible syntax:** use commas, or skip them. Same with parentheses!
- List values are accessed by calling their **index** within the `nth()` function:

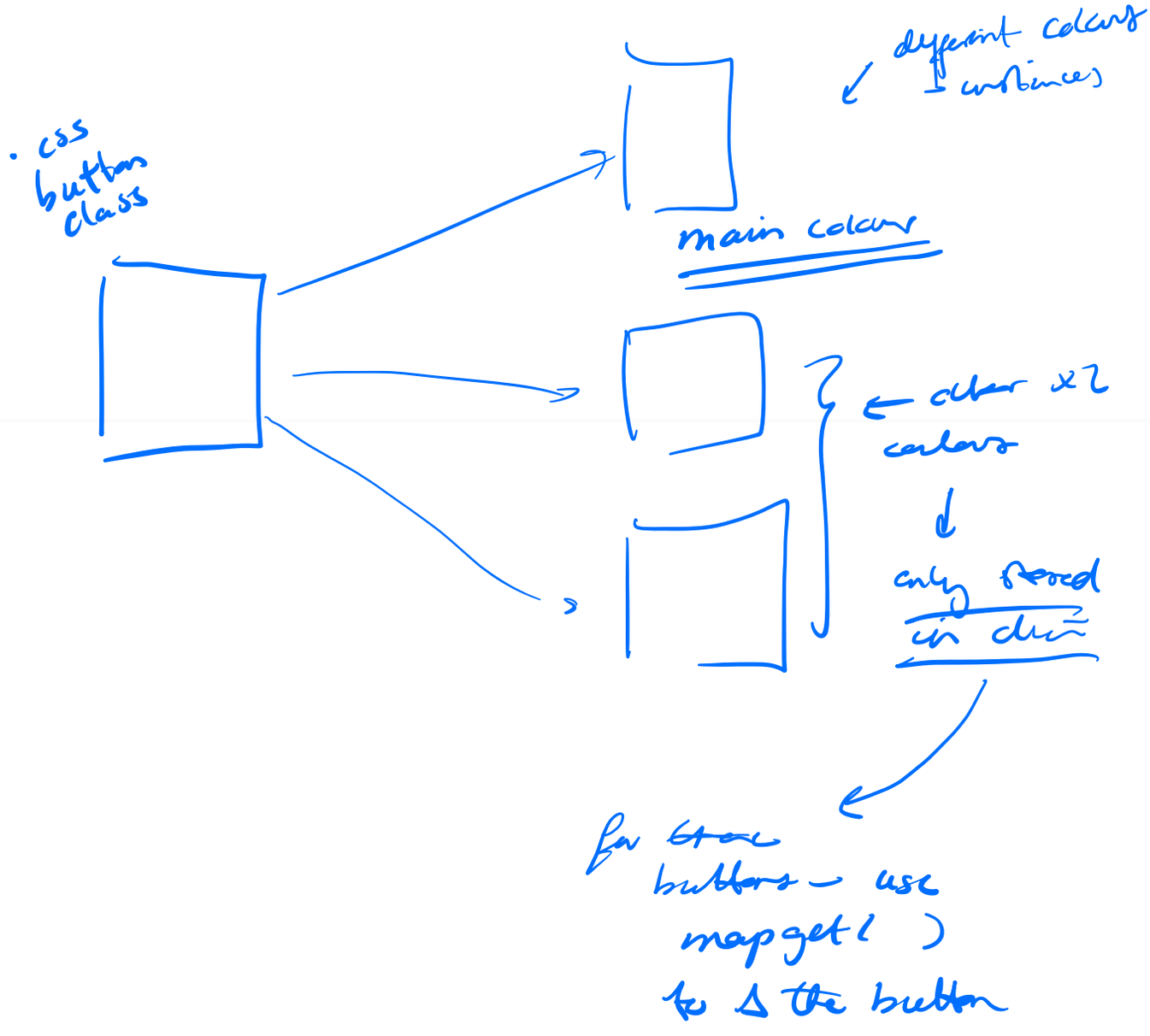
```
nth($list, index)
```

- Lists indexes start at 1.
- Maps are similar to lists, but each value is given a name, called a **key:** `$map(key: value)`
- Maps have a much more **rigid syntax** compared to lists. Maps **must** use parentheses and commas.
- Map values are accessed via the `map-get()` function: `map-get($map, key)`
- Maps and lists can contain any valid Sass **data type,** including other lists and maps.

I finished this chapter. Onto the next!

Install Sass locally

Use loops in Sass to streamline your code

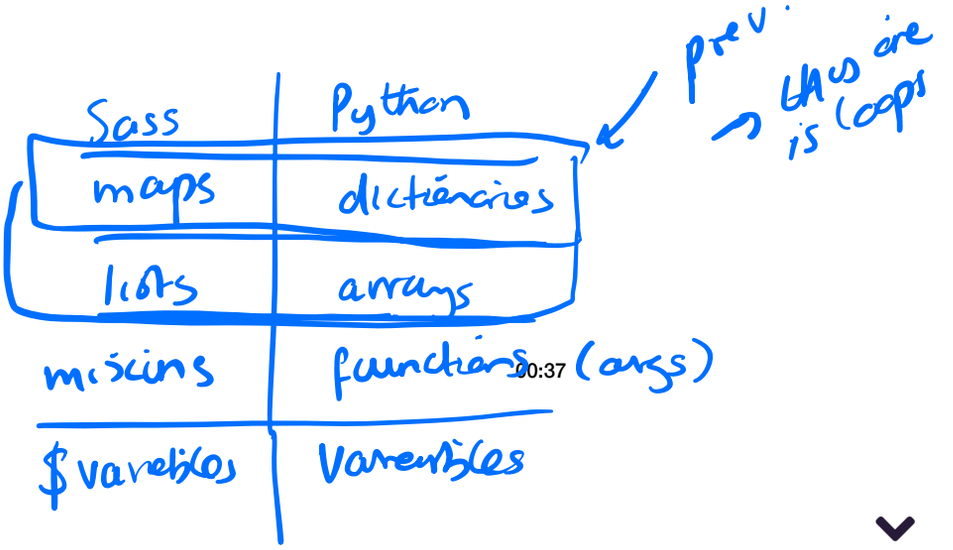


Produce Maintainable CSS With Sass

Use loops in Sass to streamline your code

4. Use loops in Sass to streamline your code

- **video notes**
  - -> lists and maps <- automatic code block creation
    - You can automate button creation e.g
    - -> based off of dictionaries / maps which create them for you
    - -> in Python they're dictionaries, and the equivalent for Sass is maps
      - The elements have keys and then it's their values
- **notes on the text below the video**
  - Loop de loop
  - Each and every last one
  - Let's recap!



Loop de loop

In the last chapter, we created a mixin to expedite making various text input modifiers:

```scss
$txt-input-palette: (
    active: (
        bg: $colour-primary,
        border: $colour-primary,
```

@ mixin ... for dynamic

@ include ... to call it

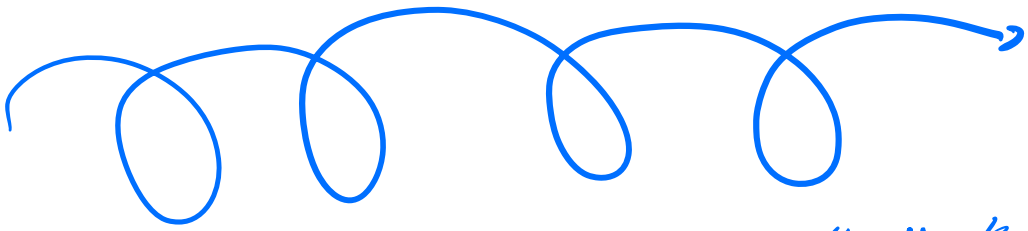
mixin → Sass class

"button" ~ function

+

list of colours

[... , ... , ... , ... , ... , ...]



iterate the list → return "button" with that colour

CONCEPT OF A LOOP

list of the x3 button states

in Sass, = mixin

```
        txt: $colour-white,
    ),
    focus: (
        bg: $colour-primary,
        border: $colour-primary,
        txt: $colour-white,
    ),
    invalid: (
        bg: $colour-invalid,
        border: $colour-white,
        txt: $colour-white,
    )
);

@mixin txt-input-palette($state) {
    $palette: map-get($txt-input-palette, $state);
    border: .1rem solid map-get($palette, border);
    background-color: map-get($palette, bg);
    color: map-get($palette, txt);
}
```

We still need to create the selector for each state and apply the mixin with the appropriate argument. This is both repetitive and tedious, and the exact thing that Sass is so great at saving us from.

In programming, there's a concept called **looping,** where you can iterate through a dataset, and execute code for each item in the set.

> Say what? 😨

Take a box of cookies for example. Within the packaging, there are 45 individual, delicious, chocolate chip cookies. 🍪 You're in the kitchen and can hear them calling to you from the cupboard, and the following routine commences:

- You crack open the packaging.
- You debate whether you really need to eat a cookie.
- You decide "no" but you've already bought them, so you might as well enjoy one, right?
- You transport yourself to nirvana as we eat a cookie.
- You feel a little guilty.
- You close the package, and tell yourself it was just one cookie.

Then you open the packaging. Again. And go through the same routine, over and over again until you've eaten all of the cookies. Some might refer to this as a vicious cycle. I prefer to call it a **loop!** A guilt-ridden loop, but a loop all the same. A loop is just **repeating a set of actions on items in a collection** until you run out of items.

By implementing a loop with your text input mixin, Sass can automatically create the relevant selectors and rulesets for each item in the map you supply it. That sounds a lot cleaner and less laborious than manually creating a block for each of the text input states, right?

And no, I promise, this isn't what gives rise to Skynet (I hope)!

There are several types of loops available in Sass, but we're only going to worry about one: the `@each` loop. It's the simplest and cleanest to set up, and works seamlessly with maps. Since all of the different types of loops that Sass has to offer are going to produce the same result in the end, why not go with the most straightforward option?

When you write an `@each` loop in Sass, you are saying that you want to perform a task for each key/value pair in a `$map`, and that is exactly how you define one:

```scss
@each $key, $value in $map {

}
```

To kick things off, define the loop with the `@each` keyword. Then declare each of *what* exactly: each `$key` and `$value`. Then from where: each `$key` and `$value` in this `$map`.

Sass goes to the map and and creates temporary `$key` and `$value` variables for each set that it finds. These variables exist only within that iteration of the loop. They don't exist before or after the iteration, and are invisible to outside code.

Each and every last one

We're going to use the `@each` loop to create all of the pseudo-selector rulesets for all of our `.form__input label` states, and fill them with their ruleset in one fell swoop! Let's tweak our `txt-input-palette` to do our bidding.

The first thing that you need to do is pass the entire `$txt-input-pallete` map rather than the specific `$state` key as the argument. Your loop will iterate through each of the keys in the map, so there's no need to name them manually:

```scss
@mixin txt-input-palette($palettes) {
    $palette: map-get($txt-input-palette, $state);
    border: .1rem solid map-get($palette, border);
    background-color: map-get($palette, bg);
    color: map-get($palette, txt);
}
```

You can also remove the `$palette` variable and its `map-get()` function. The `@each` loop will automatically get the values for each key in the map as well:

```scss
@mixin txt-input-palette($palettes) {

    border: .1rem solid map-get($palette, border);
    background-color: map-get($palette, bg);
    color: map-get($palette, txt);
```

```
6 }
```

Now you can start using `@each` to automate your workflow!

scss

```scss
@mixin txt-input-palette($palettes) {
    @each $state, $palette in $palettes{
        border: .1rem solid map-get($palette, border);
        background-color: map-get($palette, bg);
        color: map-get($palette, txt);
    }
}
```

We've added an `@each` loop that will iterate over our map, `$palettes`, with the variables `$state` and `$palette` storing the key and value, respectively. Then, within each loop, it creates rules for the `border`, `background-color`, and text `color` for each state it finds.

Only it's putting them all in a single selector:

scss

```scss
@mixin txt-input-palette($palettes) {
    @each $state, $palette in $palettes{
        border: .1rem solid map-get($palette, border);
        background-color: map-get($palette, bg);
        color: map-get($palette, txt);
    }
}

.form {
    &__field {
        & input {
            @include txt-input-palette($txt-input-palette);
        }
    }
}
```

css

```css
.form__field input {
  border: 0.1rem solid #15DEA5;
  background-color: #001534;
  color: #15DEA5;
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
  border: 0.1rem solid #fff;
  background-color: #DB464B;
  color: #fff;
}
```

Not *quite* what we were going for.

We also wanted to create the selector for each state as we looped through the map. Let's use the ampersand, `&`, and a colon, `:`, to join the name of each `$state` as pseudo-selectors:

scss

```scss
@mixin txt-input-palette($palettes) {
    @each $state, $palette in $palettes{
```

```
        &:#{$state}{
            border: .1rem solid map-get($palette, border);
            background-color: map-get($palette, bg);
            color: map-get($palette, txt);
        }
    }
}
```

That hashtag/curly brace/$variable thing is new, but don't worry! It's just Sass's **interpolation syntax.** It allows you to use a variable's value within a string. In this case, it's allowing us to use the `$state` variable's value as the name of the pseudo-selector.

Now, when you check out the compiled CSS, you should see a pseudo-selector for each state with the appropriate color values populating the ruleset:

css

```
.form__field input:active {
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
}
.form__field input:focus {
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
}
.form__field input:invalid {
  border: 0.1rem solid #fff;
  background-color: #DB464B;
  color: #fff;
}
```

Now `txt-input-palette` will generate a pseudo-selector for the state **if** the state isn't "inactive." Let's check and see if it's working properly:

css

```
.form__field input:active {
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
}
.form__field input:focus {
  border: 0.1rem solid #15DEA5;
  background-color: #15DEA5;
  color: #fff;
}
.form__field input:invalid {
  border: 0.1rem solid #fff;
  background-color: #DB464B;
  color: #fff;
}
```

There you go! Now if you change any of the color values or palette combinations, the text input rules will update automatically. And not only that but if you were to add a new state to `$txt-input-palette`,

Sass would add it to your compiled CSS without any extra work from you. How nice is that?

If you added 20 more color palettes to your map, or removed all but one, your CSS would update itself automatically. No hassle. No tedious cutting/pasting/modifying. No more looking for the hidden typo. With loops, creating and maintaining your codebase is as easy as updating the values in a map and hitting save. It doesn't get much cleaner and easier than that!

Try it out for yourself!

Now that we have the palette for our button modifiers stored in map, let's save ourselves a bit of work updating and maintaining them but putting Sass to work and have it generate the modifier selectors automatically based on the contents of the `$btn-mods` map in this interactive exercise:

- Store our loop logic in a mixin called `btn-mods` with an argument for `$map`
- Inside of the mixin, set up an `@each` loop with the variables `$mod` and `$val` for the contents of `$map`
- Inside of the `@each` loop, use the interpolation syntax to fill in the selector after an ampersand and set of dashes: `&--#{$mod}`
- Inside of the modifier selector, create a background property and give it the value of `$val`
- Remove the modifier `&--pink {...}` and `&--blue {...}` selectors and their properties
- Include the `btn-mods` mixin at the end of the `.btn` block, passing `$btn-mods` as the argument

Check the rendered page that the buttons display with the proper colors - you'll find the solution in this CodePen!

Our site is starting to look pretty snazzy on a monitor! But in this day and age, it also needs to look good on screens of all shapes and sizes including phones and tablets!

Coming up next, we'll take a look at how Sass can help make our site responsive.

Let's recap!

- A **loop** is a repeating a set of actions on items in a collection.
- **Define a loop** with the `@each` keyword, followed by the `$key`, the `$value`, and the associated `$map`.
- **Interpolation syntax** allows you to use a variable's value within a string.

I finished this chapter. Onto the next!

Integrate advanced Sass data types

Add breakpoints for responsive layouts

→ ex.

→ to gen. CSS auto'ly

→ points

#1 DEFINE THE MIXIN W/ A LOOP IN IT

- loops are given by @ each statements, defined within @ mixins, aka function def's
- → under @ each → get map func to extract val. for that iteration
- → then use &-- ← aka, BEM
- → set the props you want in CSS / Sass (compiled CSS)

#2 USE THE MIXIN

- → @ include mixin-name
- → w/ avg. under use of Sass which it's been indented / used in

#3 INTERPRET RESULTS TO SEE IF MAKE SENSE

- → codepen ← commonly used
- → check webpage / site appearance

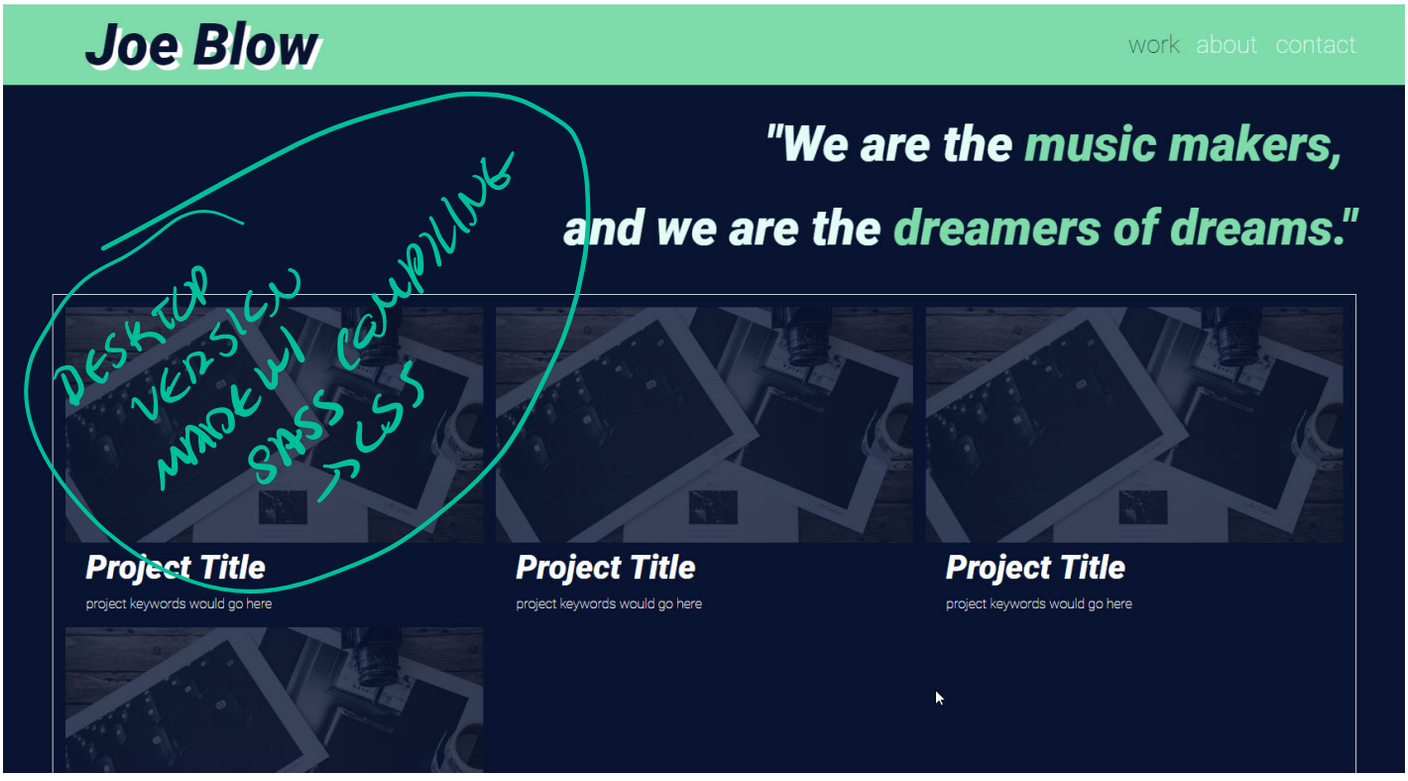
5. Add breakpoints for responsive layouts

- **video notes**
  - -> <u>different screen sizes</u>
  - <u>-> this entire chapter is Sass on e.g mobiles</u>
    - This entire thing is responsive layouts <- the screen is responding to different sizes of screen -> how the webpage changes to respond to them
  - -> changing the styling in Sass based off of the screen resolution
    - To make the same webpage responsive to e.g mobile layouts
- **notes on the text below the video**
  - Responsive layouts
  - Breakpoints done sassily
  - Even Sassier breakpoints
  - The @content directive
  - Let's recap!

CONTENTS OF CHAPTER

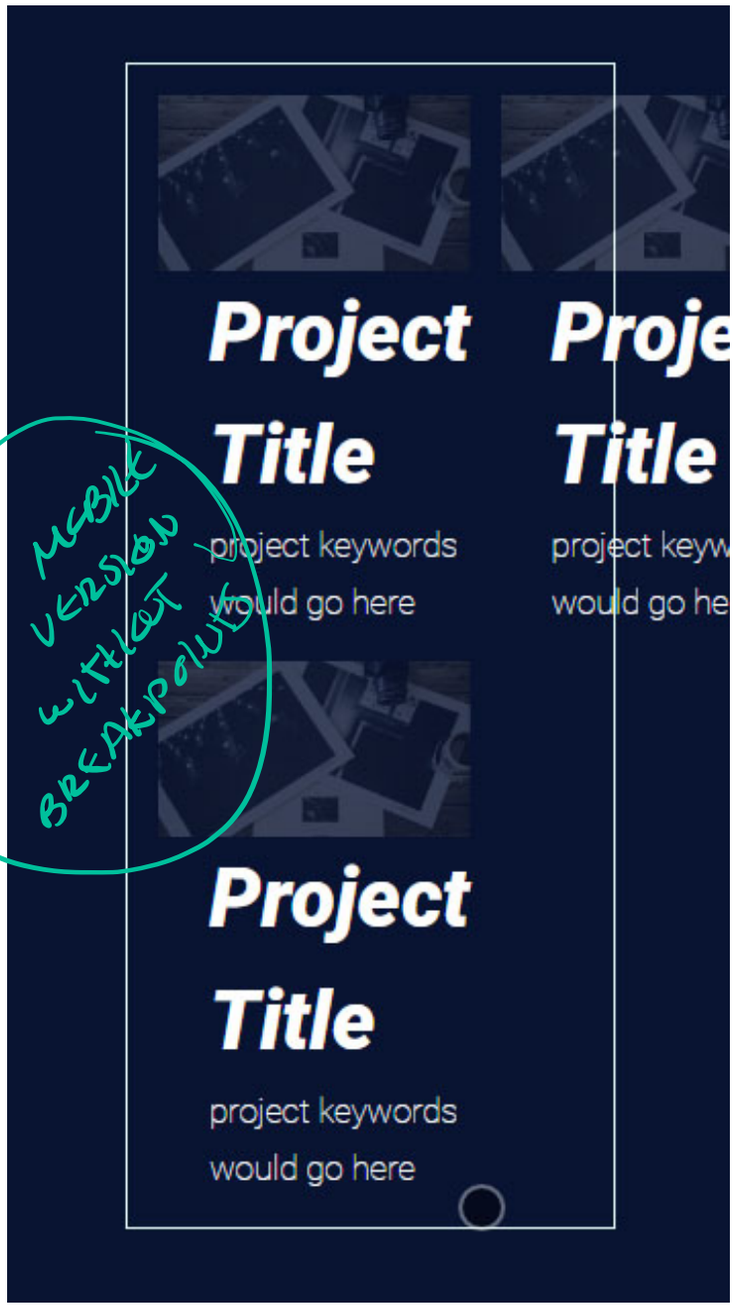
1. CONCEPT / WHY HAVE MEDIA QUERIES IN SASS
2. HOW TO DO BREAKPOINTS / MEDIA QUERIES IN SASS ← MOST BULK
   - 2A ← Media queries in Sass (Syntax)
   - 2B ← Media queries w/ maps in Sass
   - 2C ← The @ content directive
3. RECAP

1. CONCEPT / WHY HAVE MEDIA QUERIES IN SASS



But it looks bad when you view it on a mobile device!

-> the picture above is the webpage for a desktop browser and then the one below it is the same webpage, but for a mobile browser
-> the entire thing is saying - therefore we need media queries
-> the CSS for that page was compiled using Sass
-> normally you would write media queries in CSS
-> the CSS - in this project, doesn't require writing CSS
-> it requires writing Sass -> which is then automatically compiled into the CSS in the terminal by the IDE
-> you can't just write media queries in CSS
-> because the CSS is being written from the main.scss, which is compiling into it
-> so in this case - we're looking at writing the media queries for the breakpoints in scss, which which then compile (are translated into by the IDE in the terminal) into css
-> this entire chapter is about the Sass in the scss file which would compile to media queries / for the breakpoints in the css
-> so now not only do we have to do the breakpoints / media queries, but we have to write them into Sass, which then compiles into css



Ooof...😬 It looks awful on mobile devices because our page isn't yet **responsive;** the layout stays the same no matter the screen resolution. To make our site display properly on different devices, we need to implement what are called **media queries.**

Media queries tell the browser to **use an alternative ruleset** if certain circumstances are true, such as whether the medium is a screen or for print, or if the browser resolution is that of a large monitor or tiny cell phone. To execute a media query, you deploy the CSS `@media` rule, followed by the query list and a set of curly braces to contain the alternative rulesets:

scss

```scss
@media (max-width: 599px) {

}
```

With this media query, the browser will apply whatever rulesets you place within the curly braces if the browser width is less than 600 pixels.

The resolutions that you use for media queries are referred to as **breakpoints;** they're the limits for screen resolution-specific rulesets to be applied. In this case, we've created a media query with a breakpoint that will apply rulesets specifically tailored to mobile screens.

The standard CSS syntax for for media queries is to place a selector and its ruleset within the queries curly braces. When the screen resolution matches the breakpoint, it's ruleset will override the default's:

```scss
@media (max-width: 599px) {
    .proj-grid {
        grid-template-columns: 1fr;
    }
}
```

We've used the breakpoint to change the layout of the project preview grid on our work page to look better on mobile. On the desktop version, it's three columns wide, no matter the resolution of the window. But for mobile, we've put all of the project previews to be **in a single column.** Now that section of our site is much easier to read on a mobile screen:

- **Review of media queries**
  - -> reduce the screen size below a certain point -> then content starts to stack
  - -> this is the breakpoint, e.g for mobile versions
  - -> so when the screen size reduces -> this is when - in order to get the specific content to e.g stack, there are an entirely new set of css rules which need to be applied / defined
  - -> these are done using @media queries
  - -> they set the screen size which e.g corresponds to mobile versions, and within them they set the rules for how certain sections of the content are supposed to behave
  - -> this is done using indented css -> which looks like Sass indentations - but it's css
  - -> this chapter is how to optimise media queries using Sass
  - -> which then compiles into that css

New and improved mobile view

We're on our way to making our site mobile-friendly! And I have a hunch that this whole smartphone thing might be more than just a trend, so that's a good thing. 😉 But having to place the selectors within the media query means that **they won't be part of their nested BEM blocks.** This makes things harder to find, maintain, and more work to write.

Luckily, Sass, as always, is there to make your life easier. It's getting hard to imagine writing CSS with a preprocessor, right?

Breakpoints done sassily

Where standard CSS media queries require you to place the selectors within the query, Sass lets you put media queries within selectors:

scss

```scss
.proj-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    @media (max-width: 599px) {
        grid-template-columns: 1fr;
    }
}
```

Rather than having to split things up, the media query and its rules are **neatly nested in its BEM block,** making it easier to find and therefore read and update. When Sass compiles media queries, it looks to see which selector it's nested within, and renders a standard media query with the selector nested inside:

css

```css
.proj-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}

@media (max-width: 599px) {
  .proj-grid {
    grid-template-columns: 1fr;
  }
}
```

While Sass helps with media queries right out of the box, you can leverage the Sass tools we've already covered to to make it more maintainable and easier to write.

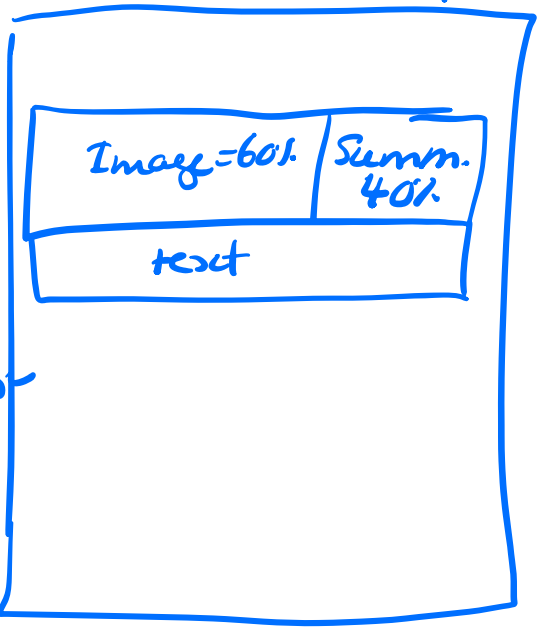
Try it out for yourself!

Let's take a look at a wireframe mockup of a page, where we have an image block, summary block, and text block. The image and summary blocks share a row, where the image takes up 60% of the width and the summary occupies the remaining space via the flex-grow property. Below those blocks sits the the text, which occupies `100%` of the width.

- **Media queries in Sass vs css**
  - -> media queries in css
    - -> these have the styles for the main page listed at the top of the styles.css file
    - -> and then below a certain breakpoint -> they list the styles for the webpage below this point in a different section
    - -> so they're grouping the styles according to the size of the webpage
    - -> you can have two styles for the same element depending on the webpage screen size
      - -> those styles for the same element are separated out according to the screen size
      - -> styles for the main screen size are at the top of the webpage
      - -> and then the styles for the element below the breakpoint are listed int he media queries section of the webpage, with the @media query
  - -> media queries in Sass
    - -> rather than separating out the styling for the same element
    - -> e.g, the same element has two styles, the styling when the screen size is main (desktop e.g), and then the styling of the element below the breakpoint in the media queries section
    - -> what's doing in Sass is putting the styling for that element in css, and then included in the definition of those styles it its own media query
    - -> so rather than having main css for the webpage and then media queries for the css of the webpage in a separate section listing how it behaves below the breakpoint, it's put the media query for each element next to its styling for the main webpage
    - -> each element has a bunch of different styles in its definition, depending on the screen size
    - -> the media queries in Sass are listed under the definition of that element's stying, rather than all of them together in a media queries section at the end of the styles.css file

EXERCISE

wireframe = diag. of webpage



-> You take that + shrink it to a mob. breakpt

-> Image size doesn't behave - enter media q's

Bridges

BEM -- syntax

-> embed . -- ing - a @ media query bit

-> that has a breakpt - set, + styles

THOUGHT PROCESS / APPROACH IN SASS MEDIA QUERIES

① tgt. el. want
② stick a media query bit's dep. in Sass

It looks fine on a large monitor. But if we take the browser window and scale it down to a more mobile-size width, things start looking a bit funky. The summary wraps to its own line, which is what we want, but the image still only occupies `60%` of the width, which isn't what we want. To fix the layout for mobile, let's use a media query to change the image width to `100%` when the browser is less than `600px` wide in this interactive exercise:

- Create a media query within the `.article__image` selector via the `@media` keyword.
- Set the query to be the `max-width` property with a value of `599px`
- After the query list place a set of curly brackets: `{ }`
- Inside of the curly brackets, place the properties that we want applied when the width of the browser window is `599px` or less, which is to set the `width` to `100%`

Review the rendered page. Resize it and ensure that the image changes relative width when the browser is less than `600px` wide. Check the solution on this CodePen.

Even Sassier breakpoints

To make things a bit more maintainable, let's create a `$breakpoints` map to store our breakpoints in. Let's add our mobile breakpoint value while we're at it:

scss

```scss
$breakpoints: (
    mobile: 599px
);
```

Writing out `@media screen and (max-width: map-get($breakpoints, mobile)` each time you need to use a media query strikes me as a bit long-winded. Let's use a Sass mixin to **abstract** away all of that verbose syntax, and create something more semantic:

scss

```scss
@mixin mobile-only {
    @media screen and (max-width: map-get($breakpoints, mobile)){
        grid-template-columns: 1fr;
    }
}
```

By naming the mixin `mobile-only`, you know at a glance that the rules it contains will **only be applied to mobile resolutions,** and it's easier to remember and type out as well. Within the mixin, we've moved the mobile version of the project preview `grid-template-columns` rule.

So let's plug our `mobile-only` mixin into the `.proj-prev` block:

scss

```scss
@mixin mobile-only {
    @media screen and (max-width: map-get($breakpoints, mobile)){
        grid-template-columns: 1fr;
    }
}

```

```
7  .proj-grid {
8      display: grid;
9      grid-template-columns: repeat(3, 1fr);
10     @include mobile-only;
11 }
```

That's a lot cleaner to look at! And when you check out the compiled CSS, you see a media query for resolutions under 600px with a ruleset for the `.proj-prev` block, just like before:

css

```
1 .proj-grid {
2   display: grid;
3   grid-template-columns: repeat(3, 1fr);
4 }
5 @media screen and (max-width: 599px) {
6   .proj-grid {
7     grid-template-columns: 1fr;
8   }
9 }
```

~~Perfect!~~

~~Except that we can only really use the mixin this one time. It's pretty useless for just about any other situation;~~ its ruleset is specific to the layout of the `.proj-prev` block's grid, and is hard coded into the mixin. That means we'll have to **write a whole new mixin for each of our media queries**. ~~And that seems like far more trouble than it's worth. Maybe Sass isn't so great after all!~~

😎 ~~Of *course*, Sass has a better way to do it. It's Sass after all!~~

The @content directive

Rather than having to hard code the content of a mixin, Sass gives the option of deploying the `@content` directive instead.

scss

```
1 @mixin mobile-only {
2     @media screen and (max-width: map-get($breakpoints, mobile)){
3         @content;
4     }
5 }
```

When Sass compiles instances of the mixin, it will replace `@content` with whatever code you place inside of the instance of the mixin.

And just how do we add content to an instance of a mixin? 🤔

When using the `@content` directive, you can add a set of curly braces to instances of the mixin to contain the content:

scss

```
1 @mixin mobile-only {
```

```
    @media screen and (max-width: map-get($breakpoints, mobile)){
        @content;
    }
}

.proj-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    @include mobile-only{
        grid-template-columns: 1fr;
    }
}
```

#2 USE THE MIXIN, AKA FUNC.

we are calling it

Sass

Now Sass will replace `@content` with `grid-template-columns: 1fr` when it compiles:

compilation

CSS

```
.proj-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}
@media screen and (max-width: 599px) {
  .proj-grid {
    grid-template-columns: 1fr;
  }
}
```

q/o

In essence, `@content` is a placeholder for code that will **be swapped in at compile time on an instance by instance basis.** By implementing it, we've created a highly flexible but succinct mixin for our media queries.

We can now use our `mobile-only` mixin throughout the site to style it for mobile devices. Let's use it to adjust the `font-size` of the `.quote` block while we're at it:

Passing the mixin as another e.g

scss

```
$font-size: (
    logo:7rem,
    quote: 6rem,
    heading:5rem,
    project-heading:4rem,
    label:2rem);

.quote {
    font-size: map-get(font-size, quote);
    @include mobile-only {
        font-size: map-get(font-size, quote)*0.4;
    }
}
```

calling the mixin

Sass compile

CSS

```
.quote {
  font-size: 6rem;
}
@media screen and (max-width: 599px) {
  .quote {
    font-size: 2.4rem;
  }
}
```

- **What @content means in Saas**
  - -> we're looking at media queries for Sass
  - -> defining a function (mixin) in Sass which can be applied to given elements and change their behaviour under different breakpoints
  - Thought process / approach that they've used
    - -> it's like a function, and the function allows you to add styles to certain elements of the webpage below given breakpoints - the function is adding media queries to the elements it's being called on
    - -> it's called by using @include under each of the elements its being reused on
    - -> so you first have to define it, then to use it
    - -> to define the function (called a mixin) for the media queries in Sass
      - @mixin name-of-function {
        - @media type and (media-feature) {
          - @content
        - }
      - }
        - -> under media-feature, this is where the breakpoints go
        - -> these are done in terms of min-width, max-width
        - -> they sometimes define these values in terms of maps (equivalent to dictionaries in Python, which are storing the breakpoints)
          - And then it's -> mapget, to return the specific from the dictionary which we want
        - -> @content
          - You literally have -> in the definition of the function aka Sass mixin, @content -> it literally just says @ content
          - -> this is a placeholder (when you call the function, that's where css styles are supposed to go)
          - -> if you were to put actual css there in the definition of the function aka Sass mixin, then the media query would only apply to one instance where it was used
            - <u>-> you want to be able to control what each of the elements is doing where that code is being applied</u>
            - -> so @content allows you to write the content whenever that mixin is being called
            - -> rather than defining the content in the definition of the function itself
            - -> because then each time the function was called, the media query would be the same and the code wouldn't be as reusable
            - -> it's basically a function which adds media queries whenever it's used
  - -> then when using the mixin
    - -> in other words, the mixin (Saas function) which adds the media queries to the elements it's applied to
      - -> with the @content (@ is called a directive)
    - <u>-> @content in the definition of the mixin means when you call the mixin inside the styling for an element in Saas, then you are setting the styles for the element there</u>
      - -> @include name-of-mixin {
        - <- the @include keyword is used whenever the mixin is called
        - Then here you are inserting the styles for the element
          - -> in other word, how it responds below the given breakpoints
          - -> the styling you want it to have inside it's media query
          - -> because the mixin is setting the media queries (in this case)

        - }
    - -> so you’re essentially defining a function which adds media queries below some mobile breakpoint for example
    - -> then you can apply that function to whatever you want in Sass, under the styling for it
    - -> then there is the generic @content directive (@) which was used in the definition of that mixin, which means whenever the mixin is called, then you can set the styles which that element obeys
    - -> so the breakpoint is set in the definition of the mixin (Sass function)
    - -> and then the styles are set wherever that mixin is used -> for the case where it’s used - in whichever class etc that might be
        - I think he prefers to call them selectors

Sass has given us the tools to create responsive and graphically consistent websites with clean, maintainable CSS, all while reducing the amount of tedious and repetitive code that we need to write.

Try it out for yourself!

We've fixed the image width of our articles using a breakpoint, but there's more that we can do to fix the responsiveness of our article wireframe. The article itself is a good place to start. On the desktop, it has a width of `75%`, but, just like the image, we'd like it to occupy the full width of the container on a mobile screen. And let's make the summary font size a bit bigger while we're at it.

We could create another media query, but a DRY-er way of doing things would be to port the media query we made for the image width into a mixin, using Sass' `@content` directive to make it more versatile and reusable. In this interactive exercise:

- Create a mixin named `mobile-only`
- Cut the media query from the `.article__selector` and paste it inside of the `mobile-only` mixin
- Replace the `width` property with the Sass `@content` directive
- Include the mobile-only mixin within the `.article__selector`, using a set of curly braces to include the rules that we want placed within the mixin's media query: `width: 100%;`
- Do the same for the `.article` selector, setting it's `width` to `100%` as well.
- Include the `mobile-only` mixin within the `.article__summary` mixin, setting the `font-size` to `1.5rem`

Review the rendered page. When the browser is scaled down to a mobile-size width, the article block should occupy 100% of the width of the window, as should the image. The summary font size should increase 50% to `1.5rem`. Check your solution with this CodePen.

Coming up next, we'll put the cherry on top by automatically adding vendor prefixes to our compiled CSS, ensuring our site displays properly across the major browsers.

Let's recap!

- **Media queries** allow you to adapt content to different screens by telling the browser to use an alternative ruleset if certain circumstances are true.
- To **execute a media query,** deploy the CSS `@media` rule, followed by the query list, and a set of curly braces containing the necessary alternative rulesets.
- **Breakpoints** are the resolutions that you use for media queries.
- In Sass, you can put media queries within selectors, which allows you to nest them in the appropriate BEM block.
- Use `@content` as a code placeholder for specific media queries.

Ex → x2 Δ's made
a) desktops 75% page, mobile width = 100%
b) font size bigger in screenmany section

Problem solving approach

a) → take prev code, @content to make it ↑ generic
→ then apply that for use on all content on page / body
↑ - to force it to take up 75% of the middle of the page

@include name-of-mixin {

← css styles
}

@mixin {

}

MIXINS a) DEFINE THE FUNC., "MIXIN" IN SASS
b) USE THE " " " "
VIA @include

→ you can reuse the mixin as many times as you want

b) → for this you reuse the same mixin - just with different styling

---

Then you analyse the results to see if they make sense

a) codepen
b) compiled CSS from Sass
c) appearance of webpage

← don't think there's a W3C Sass Validator

- you need to know what Δ's you're looking for
- Console inspection
- look @ the spec.els

- **Extra points which came up from doing the exercise**
  - The names of different brackets
    - -> {} braces
    - -> () parentheses
    - -> [] brackets
    - -> <| bra
    - -> |> ket
  - -> so we have the mixin (Saas function) which sets media queries for certain breakpoints
    - And the styling for that media query, in the Saas definition is @content <- this is called the content directive
    - -> and then you can use that mixin function as many times as you want on different css selectors
    - -> so for example - if you were to use it on a given Saas selector using @include, then when you did, you would have to set the specific css / sass rules which apply for that situation
    - -> every time you use that Sass mixin (function, in this case which sets the media queries) you need to tell it what the styles are for that element its being used on
      - In each case where it's being used, you can change the styles to be anything you want
      - -> each of the times it's being used, the thing which they have in common are the breakpoints
      - -> you have one mixin function per breakpoint you want to target -> that it what's being reused across the different mixins
      - -> the things which are different are the styles
      - -> so here for example, the styles for two different sections on the webpage were being targeted for the mobile breakpoints
        - The same mixin was being used to target both
        - -> but the styling was different for each one
        - **-> the styling was specific to the elements, and the mixin (Sass function) being used across both was specific to the breakpoint which was being targeted**
          - **-> reduce the webpage past a certain point - and that's what's being targeted**
  - -> to check whether the function has worked - for media queries
    - -> the entire thing is having a responsive design
    - -> you want the page to respond well below certain breakpoints
    - -> there is a responsive design mode in the console <- to check how it responds to different device sizes
    - -> to check how it's responding to different screen sizes -> you don't just open the inspector in the console, you have to open the responsive design mode in the browser
      - -> you are checking how it responds to screen widths specific to those devices
      - -> and with questions involving breakpoints / media queries / screen widths / responsive design, this is how you check it's worked
      - -> you are looking for the specific elements - how they behave below the breakpoint in that responsive design mode which you open in the console

Produce Maintainable CSS With Sass

Use Autoprefixer for browser compliant code

6. Use Autoprefixer for browser compliant code

- **video notes**
  - -> looking at the behaviour of the site across different browsers
  - -> media queries are about the behaviour of the content across multiple different screen sizes
  - -> this chapter is about the behaviour of the page across different browsers
  - -> **this is a tool called autoprefixer**
- **notes on the text below the video**
  - Different displays for different browsers
  - Installing Autoprefixer
  - Let's recap!
  - Bonus end of part challenge

CONTENTS

1) Vendor prefixes
2) The autoprefixer extension installation / use
3) Recap
4) Exercise

1) VENDOR PREFIXES

Different displays for different browsers

Our site looks good when you check it out in your browser but not necessarily everyone else's. We've developed it using Chrome but someone else might use Firefox, Opera...or *Internet Explorer* (gasp). 😱

The point is, some **CSS properties will display differently on different browsers**, and you want

One chapter done, nice! Close

properties, such as `border` or `margin`, there's nothing to worry about; they have been standardized and work uniformly across browsers. But newer properties, such as `grid` or `flexbox`, aren't as straightforward.

The vendors who make your browsers, such as Google and Mozilla, don't want to wait for the cool new stuff to be standardized before they implement it. Instead, they make their own version of it, adding a prefix to differentiate it from other browser's implementations. Eventually, the new property will be standardized and work the same across all browsers, removing the need for prefixes. Until then, you need to use vendor prefixes in your code.

When using flexbox, rather than just defining the `display` property as `flex`, you should be writing out all of the **vendor prefixes** as well:

```scss
.header {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
}
```

Now our header will display uniformly on Chrome, Safari, Firefox, Opera, and Microsoft Edge! I know, we haven't done that *at all* so far. And we've written a lot of code. And now we have to go back and update all of it. And that seems like a lot of work.

But I don't ***want to!***

Don't worry, nobody does. Vendor prefixing is tedious and constantly changing. Since the standards are constantly changing, you need to look them up every time you write CSS to ensure you're writing compliant code.

② THE AUTOPREFIXER EXTENSION INSTALLATION / USE

"**Autoprefixer**" is a plugin that can save you from the monotony of `-webkit-` and `-moz-`. It does exactly what it sounds like: automatically add prefixes to your CSS. All you do is supply it with a CSS sheet, and it will read through it and add vendor prefixes where needed.

Installing Autoprefixer

So, how do you get this sanity-saving plugin up and running? Open the terminal in VS Code and use **npm** to install it, as well as `postcss` and its companion `postcss-cli`, a command line tool that you'll use to run Autoprefixer. The process is pretty much the same as installing Sass, only you're installing three packages at once. You can install as many at a time as you'd like, each separated by a space:

```
npm install autoprefixer postcss postcss-cli -g
```

Once npm has downloaded and installed the packages, you need to jump back into `package.json`

One chapter done, nice!

```json
{
  "name": "joeblow",
  "version": "1.0.0",
  "description": "Joe Blow's web portfolio",
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix":
  },
  "author": "",
  "license": "ISC",
}
```

And inside the script, you need to tell npm to use the new `postcss` package you've just installed, as well as where to find your compiled CSS file:

json

```json
{
  "name": "joeblow",
  "version": "1.0.0",
  "description": "Joe Blow's web portfolio",
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix": "postcss ./public/css/style.css"
  },
  "author": "",
  "license": "ISC",
}
```

After you've told npm which package to use and where to find the CSS file, you need to tell the `postcss` package to use Autoprefixer by implementing its `--use` flag followed by `autoprefixer` :

json

```json
{
  "name": "joeblow",
  "version": "1.0.0",
  "description": "Joe Blow's web portfolio",
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix": "postcss ./public/css/style.css --use autoprefixer"
  },
  "author": "",
  "license": "ISC",
}
```

And then, finally, you need to tell it where to put your new, prefixed, CSS sheet:

json

```json
{
```

One chapter done, nice!

```
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix": "postcss ./public/css/style.css --use autoprefixer -d ./public/css/prefixed/"
  },
  "author": "",
  "license": "ISC",
}
```

Now your prefixing script is complete! All that's left to do is tell Autoprefixer how far back you want it to cover in your browser compatibility. By default, Autoprefixer **will only look at the previous version** of the major browsers to figure out which vendor prefixes to add to your sheets. But you want to make sure that people with slightly older versions of browsers will be covered as well. Let's tell Autoprefixer to look at the last four versions of the common browsers when it checks the compliance of our CSS sheets.

Right after our scripts, let's add a new key named `browserslist` :

json

```
{
  "name": "joeblow",
  "version": "1.0.0",
  "description": "Joe Blow's web portfolio",
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix": "postcss ./public/css/style.css --use autoprefixer -d ./public/css/prefixed/"
  },
  "author": "",
  "license": "ISC",
  "browserslist":
}
```

And then give `browserslist` a value of `last 4 versions` :

json

```
{
  "name": "joeblow",
  "version": "1.0.0",
  "description": "Joe Blow's web portfolio",
  "main": "index.js",
  "scripts": {
    "sass": "sass ./sass/main.scss:./public/css/style.css -w --style compressed",
    "prefix": "postcss ./public/css/style.css --use autoprefixer -d ./public/css/prefixed/"
  },
  "author": "",
  "license": "ISC",
  "browserslist": "last 4 versions"
}
```

Now you're ready to auto-prefix your CSS! 🤘

One chapter done, nice!

- -> checking for browser releases and versions
- -> say we're using a new feature, e.g flex -> which hasn't been standardised across browsers yet
- -> then, vendors of these browsers release different prefixes for the new features in the css, before they become standardised
- -> we are checking for these prefixes by putting a line of code into the json file
  - -> using the auto prefix plugin
  - -> an extension is for example, a Sass class extending to another one
  - -> what this is, is us using someone else's code
    - -> then importing it in using npm manager in the terminal in the IDE
    - -> then doing that creates the json file
    - -> then the line in there - that's what we're changing
    - -> then once we have that setup, we're running it in the terminal which compiles the prefixes onto the css from the Saas
- -> then -
  - You can't just check for the prefixes across different browser types
  - -> you have to check for them from e.g four releases of those browsers back
  - -> in case the users haven't updated their browsers, or they're operating from phones for example, which aren't compatible with the newest releases of that browser
  - -> so we're not just checking for the different browsers (e.g Chrome / Firefox etc), we're checking for the prefixes across the different releases of those browsers
    - -> and that's done by changing one of the lines on the json file -> to check for prefixes from x number of browser releases back, for each browser checked

```
npm run prefix
```

The script pops up in the command line interface, and, a moment later, it's all done!

```
PS Z:\Cloud\Dropbox (Personal)\2017\__code__\joeblow> npm run prefix

> joeblow@1.0.0 prefix Z:\Cloud\Dropbox (Personal)\2017\__code__\joeblow
> postcss ./public/css/style.css --use autoprefixer -d ./public/css/prefixed/
```

Take a look at the new, prefixed, CSS and check out the `.header` block:

CSS

```css
.header {
  background: #15DEA5;
  height: 10rem;
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-align: center;
      -ms-flex-align: center;
          align-items: center;
  -ms-flex-wrap: wrap;
      flex-wrap: wrap;
  width: 100%;
}
```

Now our website will display properly and uniformly across all **compatible** browsers.

> Note the word "compatible." Flexbox, for example, isn't supported on Internet Explorer 9 or earlier. Adding a prefix won't change that. **Prefixes ensure that different implementations of the same property are invoked in their respective browsers.** If you need a site to display properly on something like IE8 (😬), you should use different properties for your layout.

Our website is ready for the general public! It's responsive and compliant. And maintaining it in the future will be so much simpler using Sass.

Let's recap!

- Some CSS properties require vendor prefixes and will display differently on different browsers.
- Use the plugin **Autoprefixer** to automatically add prefixes to your CSS.
- Install Autoprefixer using the command line, and be sure to **indicate how many older browser versions** you would like it to cover!

Bonus end of part challenge

Use loops and maps to create modifiers

One chapter done, nice!

The following is completely optional, but is great practice at creating maps and using them with `@each` loops, which can help save time and keep your code maintainable.

The challenge is this: automate the creation of modifiers that change the font color of `.article__headline` to the following color values:

1. #cc8624 (orange)
2. #cc2475 (pink)
3. #2469cc (blue)
4. #24cc24 (green)

Put the color values into a map, giving each a key, then use an `@each` loop to generate selectors with the following structure:

```css
.article__heading--/*colour name goes here*/ {
    color:/*colour values goes here*/;
}

```

If you're feeling a bit fuzzy on maps, you can brush up on them in the previous chapters! 😀

Ex

→ told - maps, @each loops ← loops

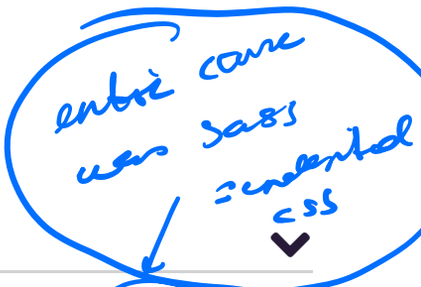
↑ ~Python dict's

→ modifiers = ? → we want to gen. ↗ font color for the heading → we don't have as is

Course summary

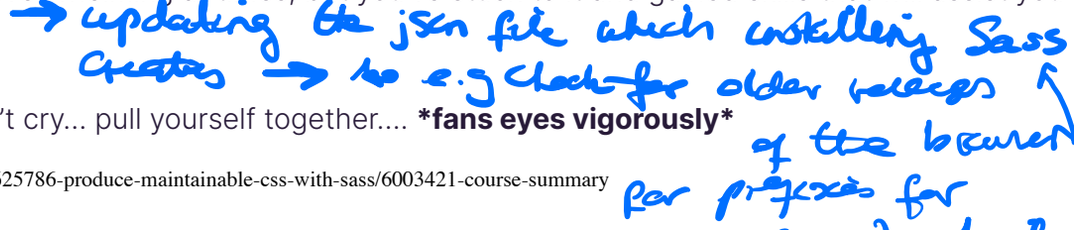
7. Course summary

- **video notes**
  - egfbef
- **notes on the text below the video**
  - Mission accomplished



Mission accomplished

You did it! You've taken your CSS skills to another level, learned to use powerful tools, like Sass, to improve your code, and laid a solid foundation as programmers to boot! I know that there was a lot to take in, and it was a bit overwhelming at times, but you've stuck to it and gained skills that will assist you throughout your career.

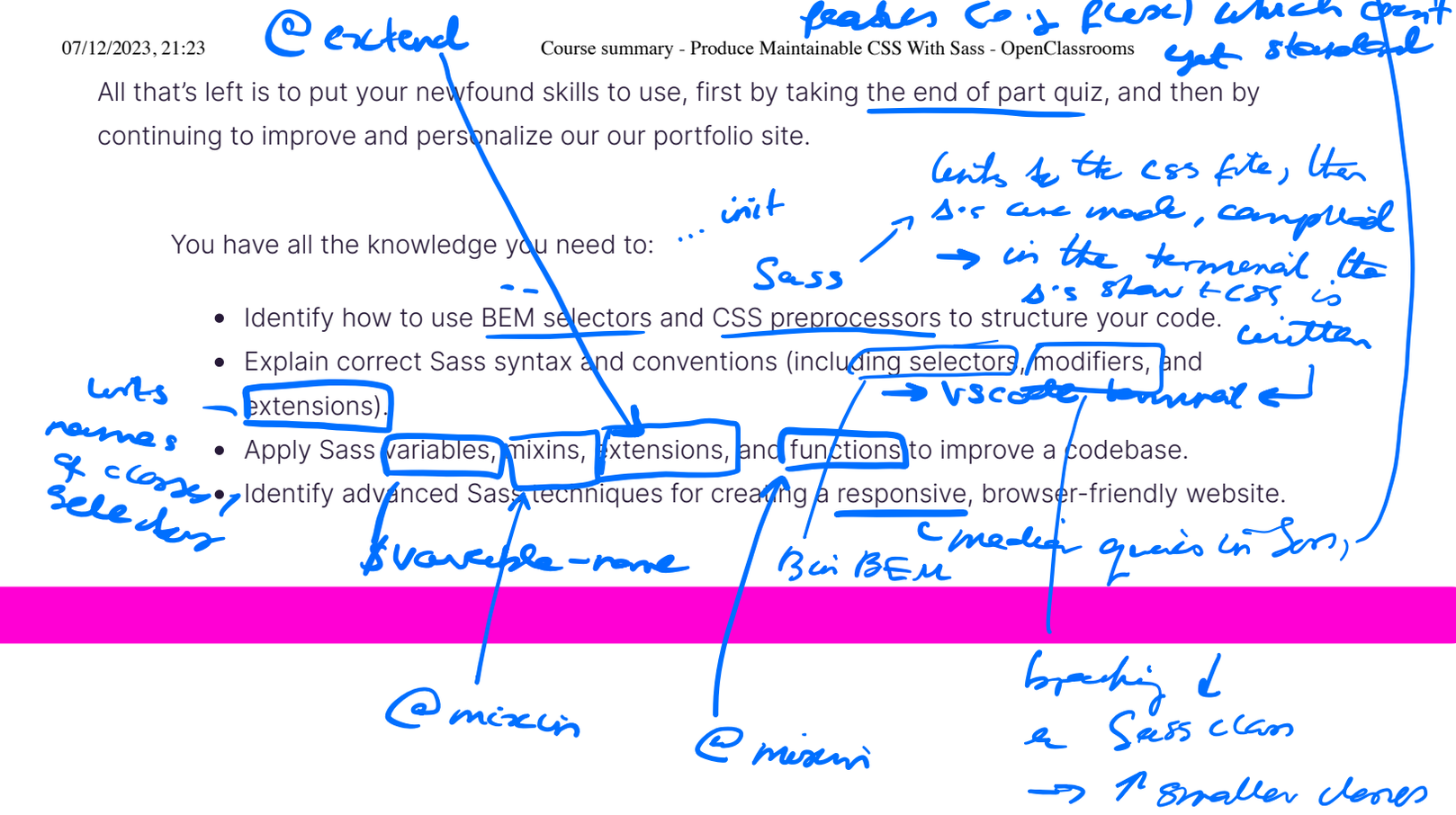


😢 I told myself I wouldn't cry... pull yourself together.... ***fans eyes vigorously***

All that's left is to put your newfound skills to use, first by taking the end of part quiz, and then by continuing to improve and personalize our our portfolio site.

You have all the knowledge you need to:

- Identify how to use BEM selectors and CSS preprocessors to structure your code.
- Explain correct Sass syntax and conventions (including selectors, modifiers, and extensions).
- Apply Sass variables, mixins, extensions, and functions to improve a codebase.
- Identify advanced Sass techniques for creating a responsive, browser-friendly website.



Quiz: Apply advanced Sass techniques to your code

LANGUAGE

Produce Maintainable CSS With Sass

Quiz: Apply advanced Sass techniques to your code

Apply advanced Sass techniques to your code

Evaluated skills

Identify advanced SASS techniques for creating responsive, browser-friendly websites

Question 1

Which of the following directory names are not part of the 7-1 file system?

- Base
- _variables.scss
- Themes
- Utils

Question 2

Which of the following are properly formatted Sass maps?

scss

```
$map: (key1,$value1,key2,$value2,key3,$value3);
```

scss

```scss
$map: (key1:$value1, key2:$value2, key3:$value3);
```

scss

```scss
$map: value1, value2, value3;
```

scss

```scss
$map: $value1 $value2 $value3;
```

Question 3

Autoprefixer creates cross-browser compatible CSS by:

- Removing non-compliant code.
- Placing non-compliant code in a separate file.
- Automatically adding the necessary prefixes to our CSS.
- Alerts the user to upgrade their browser to the current version.

Question 4

scss

```scss
$map: (
    pink: #f442f1,
    mint: #15dea5,
    navy: #2a2760,
);

@each $key, $value in $map {
    .btn--#{$key} {
        background-color: $value;
    }
}
```

Which of the following selectors would be generated from the @each loop above?

Careful, there are several correct answers.

css

```css
.btn--pink {
    background-color: #f442f1;
}
```

css

```css
.btn--mint {
```

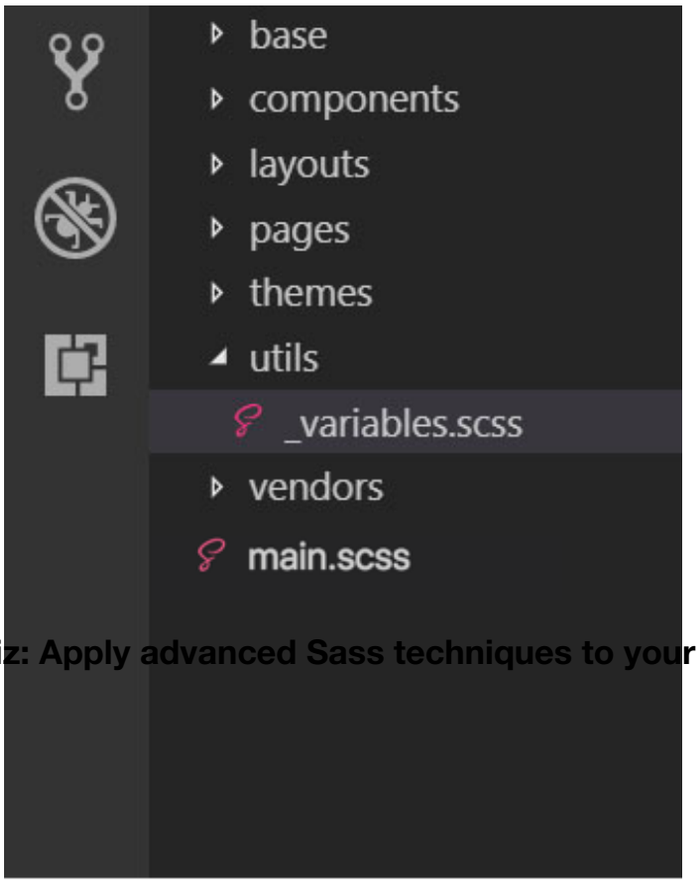
```
2     background-color: #15dea5;
3 }
```

```css
1 .btn .navy {
2     background-color: #1a2760;
3 }
```

```css
1 .mint {
2     background-color: #15dea5;
3 }
```

Question 5

Which of the following .sccs snippets is properly formatted to import the _variables.scss partial into main.scss given the following file-structure:



Quiz: Apply advanced Sass techniques to your code

```scss
1 @import "./Utils/variables";
```

```scss
1 @import "variables";
```

```scss
@import "./Utils/variables";
```

```scss
@import "./Utils/variables";
```

Question 6

Which of the following are validly formatted Sass lists?

Careful, there are several correct answers.

```scss
$list: $value1, $value2, $value3;
```

```scss
$list: ($value1, $value2, $value3);
```

```scss
$list: $value1 $value2 $value3;
```

```scss
$list: $value1$value2$value3;
```

Question 7

What will the margin of `.block` be if the browser width is 480px?

```scss
@mixin mobile-only {
    @media screen and (max-width: 599px){
        @content;
    }
}

.block {
    margin: 1.5rem;
    @include mobile-only {
        margin: .5rem;
    }
}
```

- [ ] `1.5rem`
- [ ] `2rem`
- [x] `.5rem`
- [ ] `0`

Question 8

Which of the following snippets will return a value of 3rem from the following Sass list?

```scss
$list: 1rem, 2rem, 3rem, 4rem;
```

- [ )

```scss
nth($list, 2);
```

- [x]

```scss
nth($list, 3);
```

- [ ]

```scss
$map-get($list, 2);
```

- [ ]

```scss
nth(3, $list);
```

Question 9

```scss
$map: (
    small: 1rem,
    medium: 5rem,
    large: 10rem
);
```

Given the map above, which of the Sass loops would produce the following selectors:

```css
.btn--small {
    font-size: 1rem;
}
.btn--medium {
    font-size: 5rem;
}
.btn--large {
    font-size: 10rem;
```

```
9 }
```

○

```scss
@each $size, $measurement in $map {
    .btn--#{$measurement} {
        font-size: $size;
    }
}
```

✓

```scss
@each $size, $measurement in $map {
    .btn-#{$size} {
        font-size: $measurement;
    }
}
```

✓

```scss
@each $size, $measurement in $map {
    .btn--#{$size} {
        font-size: $measurement;
    }
}
```

○

```scss
@each $size, $measurement in $map {
    .btn--${size} {
        font-size: $measurement;
    }
}
```

Question 10

```scss
$font-sizes: (
    heading: 3rem,
    byline: 1.75rem,
    caption: 1rem
);
```

Given the Sass map above, which code snippet would return the value for the `byline` key?

✓

```scss
.article__byline {
    font-size: map-get($font-sizes, byline);
}
```

scss

```scss
.article__byline {
    font-size: map-get(byline, $font-sizes);
}
```

scss

```scss
.article__byline {
    font-size: nth($font-sizes, byline);
}
```

it's a map, not a list → nth is for lists
→ map-get is for maps

scss

```scss
.article__byline {
    font-size: map-get($font-sizes, 2);
}
```

maps aka, the Sass equiv. of Python dictionaries use keys, not indexes → and lists, aka Sass lists aren't zero indexed

Question 11

scss

```scss
.form{width:100%;padding-bottom:1.5rem}.form__heading{width:100%;color:#fff;text-shadow:.55rem .55rem #11af82;background:#15DEA5;line-height:5rem;padding:1.5rem}.form__field label{color:#D6FFF5;display:block;font-size:2rem;line-height:2rem;padding-top:1.5rem}.form__field input{width:100%;background:#001534;border:0.1rem solid #15DEA5;padding:1.5rem;color:#D6FFF5;font-weight:900;font-style:italic;font-size:2.75rem}.form__field textarea{width:100%;color:#15DEA5;background:#001534;border:0.1rem solid #15DEA5;outline:none;padding:1.5rem;margin-bottom:.75rem}
```

To prepare for deployment, which of the following snippets will compile Sass code to minified CSS, as seen in the code above?

- ✗ `--style nested`
- ✗ `--style expanded` regular CSS
- ✗ `--style compact` spaces

Sass → CSS auto compiles → compiler modes to e.g. save space

→ those lines setting the compiler mode - those would go in the json file which is

```
--style compressed
```

Question 12

What must you add to your package.json file to ensure Autoprefixer adds prefixes to cover the previous three versions of the major browsers?

- ○
```json
1 "browser-history": "3 versions"
```
- ○
```json
1 "browserslist": "last 3 versions"
```
- ○
```json
1 "browsers": "last 3 versions"
```
- ○
```json
1 "browserslist": "last 3"
```

You need to answer 12 more questions.