%%
Contents 7i Produce Maintainable CSS With Sass **Panteli**
%%

%%
7i Part #1 - Structure your code efficiently
%%

1. Get the most out of this course

- css
  - -> for styling webpages
  - -> we end up with too much of it, which slows down the webpages
  - -> bem, for naming css classes
    - there isn't a lot of structure in css syntax
      - -> clean, well structured css
      - -> maintainable and modular
      - -> to code faster and more efficiently
  - -> saas which is a css pre-complier
- contents of the course
  - -> css structure
    - -> specificity
    - -> BEM structure

        - -> css precompliers
    - -> sass
        - -> .scss files
        - -> variables, functions, conditions, mixins, extensions
        - -> loops, maps, different browser types
- learning outcomes
    - -> BEM selectors
    - -> css preprocessors
    - -> sass variables
        - mixins, extensions, functions
        - responsive webpages
    - -> css and html is a pre-requisite
- how to produce maintainable css <- this is with sass
    - -> this course has a portfolio project
    - -> there are four parts to it
    - -> key concepts, quizzes, reviews, practice
    - -> it's a four part sass course
    - -> and saas is for the css
- -> it's narrated by Patrick Gerke
    - web developer

2. Structure your CSS effectively

- **video notes**
    - -> this is about cleaning up the css
        - it can become messy
        - css syntax <- minimal structure
    - -> because the syntax for css is just flat, with little indentation
- **notes on the text below the video**
    - Why write well structured code?
        - -> code is created bit by bit -> and then you realise it's a mess
        - -> when the project is done
        - -> so if you go back to it ---
        - -> html has structure but the css is the complete opposite
            - -> when changes are made to the webpage, the css just becomes even more messy
            - -> changing things manually
            - -> when the client wants things changed on the webpage
            - -> you need to understand what the different parts of the css do - when you come back to them
            - -> or making changes the client wants will be inefficient
        - -> the html has a hierarchy and the css just turns into a mess
        - -> specificity - if multiple pieces of css targeting the same element, then the one which is higher on the hierarchy is the one which takes precedence for that element
        - -> file structures, css precomplier called sass to make the css cleaner

HTML

```html
<body>
    <main class="container">
        <form class="form">
            <h1 class="form__heading">Contact me</h1>
            <div class="form__field">
                <label for="contact-name">name</label>
                <input type="text" name="contact-name" id="contact-name">
            </div>
            <div class="form__field">
                <label for="contact-email">email</label>
                <input type="email" name="contact-email" id="contact-email">
            </div>
            <div class="form__field">
                <label for="mssg">message</label>
                <textarea name="mssg" id="mssg" cols="30" rows="10"></textarea>
            </div>
            <button class="btn btn--full-width">Submit</button>
        </form>
    </main>
</body>
```

CSS

```css
.container {
    margin: 6rem;
    padding: 1.5rem 1.5rem 0 1.5rem;
    border: 0.1rem solid #D6FFF5; }

.form {
    width: 100%;
    padding-bottom: 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;
}
```

Html has more structure than css, this means css becomes a mess a lot faster than the css does

- -> sass is used to target the css
- -> **saas is software as a service and sass is a css precompiler - something which makes the css less messy, because css has way less structure than the html does**

- Good fences make good neighbours
  - the thought process to make clean / maintainable css
    - -> find categories, repeated lines of code
      - repeated parts of css in the code <- DRY, don't repeat yourself
      - when you change the values, the more of them which are repeated, the more likely you are to miss a value
    - -> refactor them
      - -> cleaning up the code
      - -> find the repeated styles
      - -> in other words, some of them are irrelevant
      - -> when there is an element of html and it's having multiple classes -> these could be condensed into one
    - -> separation of concerns - aka, break a css class into smaller ones
      - -> this is when you break down e.g a button class into multiple smaller classes
      - -> you're not taking the repeated code and making it into one class, you're taking a class and breaking it down into smaller classes / dx's
        - -> the aim is not to form one larger class out of the repeated code, it's to break down the classes which already exist into lots of smaller classes - so that they aren't being repeated in the css
        - -> so before: < button class="btn" >...</button>
          - after: < button class="btn btn-rounded btn-wide" >...</button>
      - -> markup is another word for html
- Have a plan and stick to it
  - make a plan for the css naming conventions and stick to it
    - -> there are no rules for the css, but this course presents methods which generally work
      - the aim is to develop clean css <- for large / small projects etc
    - -> provisos - a condition or qualification attached to an agreement or statement, in this case it's a condition / context under which the css rule etc is true / applies
    - -> make a plan for the front end code
      - you're not looking at the code which you have once it's written, and then doing the separation of concerns on that at the end
      - -> you're planning the parts which the larger classes would be split into before you even start writing the css
        - -> everything is in the planning, and then in the implementation
    - -> this course includes how to create a naming convention for the css classes, and how browsers decide which css rules take precedence
  - exercise
    - -> codepen exercise https://codepen.io/nicolaspatschkowski/pen/abjxoxj
    - -> the language is
      - you have a css class, with a lot of properties -> e.g a heading class
      - -> you are "separating its concerns" <- this is 'code' for, break the larger css class down into smaller constituents
      - **-> the thought process is, take a class and break it down into smaller classes - e.g, break apart the heading class from the uppercase one - and then use both of the smaller classes**
        - -> so say for example, we have horizontal and vertically flexed cards for the Booki project
        - -> you define a card class, then you define two more classes, one which

flexes the contents in a column and one which does it in a row <- don't redefine an entirely new section in a sandbox
                - -> the more classes which are used on something, the merrier
                    - because the aim isn't to reduce the amount of classes, it's to separate the css in those classes out, so that it's more reusable - the aim is to reduce the amount of repeated css, which means separating it out into more classes - overall the amount of css is reduced
    - Let's recap!
        - -> the aim isn't just to create the webpage, it's to do it in a way which makes the css flexible and reduces the amount of extra tripe
        - -> there is no rigid structure to css
            - so it can get messy quickly
            - -> these are the best practices to reduce the amount of extra css
            - -> DRY, don't repeat yourself
        - -> so make your own structure -> separation of concerns, break down a class into smaller classes, to reduce the amount of css which is repeated
            - -> this also makes it more readable and maintainable

3. Understand how specificity affects your code structure

- **notes on the video**
    - -> making a plan to structure the css
    - -> this is looking at how browsers apply css
    - -> keeping score of which styles are being used
- **notes on the text below the video**
    - What is specificity?
        - when one html element has multiple pieces of css targeting it, a hierarchy is used to see which is used
            - -> if one html element is being targeted by a class and an id
            - -> then the css which is associated with the id takes precedence
            - -> because ids are higher in the hierarchy than classes are
            - -> this is what happens when the rules for each of them are different <- in other words how css handles conflicts
            - -> when there are multiple pieces of css targeting one html element -> the one with the highest status in the hierarchy takes precedence
                - precedence not presidence or presence <- it's with c's and no i's
            - -> the css hierarchy
                - -> inline <- e.g Click Here!
                - -> ids
                - -> classes, pseudo-classes, attributes
                - -> elements and pseudo-elements
                    - there is an html tag which is <style></style>
        - -> <style>
            - this is to embed css in html
            - -> you use one of those tags in html, and then inside it - that's where all the css goes - what you use for putting css in an html document
            - -> you can also link them to external styles in the html document
        - -> how browsers do this when there are multiple conflicting css rules for an element
            - -> you can take an element and count the frequency of times it's being targeted by different pieces of css
            - -> then you can categorise that into a frequency table - of inline, ID, class and element pieces
                - -> i.e with the highest status in the css hierarchy first

- -> "specificity", because some elements (e.g inline elements are more specific than entire generic classes)
- -> if e.g there are two ids then it moves on to see which has more classes / elements - and then depending on the one with the larger overall precedence - this is the style which is used
- -> it's a system to quantify the styles when there are multiple conflicting css parts
- -> by hand, he's done it by creating a tallying system -> with a table

`#submit-button {...}` :

Inline	ID	Class	Element
0	**1**	0	0

`.button {...}` :

Inline	ID	Class	Element
0	0	**1**	0

`div {...}` :

Inline	ID	Class	Element
0	0	0	**1**

- -> **specificity is an actual number ->**
  - **you can calculate the specificity of the elements from a table, by tallying it**
  - **-> this is what browsers do when there are multiple pieces of css targeting the same element**
- -> ids are still more specific than two classes used together
  - -> one can only be used to target that specific element, and the other can be used to target multiple
  - -> the browser works from LHS to RHS
  - -> syntax
    - .button.styles {
    - }
    - vs
    - .button .styles {
    - }
    - **-> the top one looks for an element which has both of those classes, and the bottom one looks for elements which have the styles class coming off of the button class (dependents)**
      - -> this would be called a "descendant combinator"
        - -> this one, then this one
        - -> vs - this and this -> this and this is no spaces, this one embedded in this one - this one then this one has a space in it
        - -> it's like in probability theory, when you times things together "and" is when you have . and no spaces between the elements
        - -> one is this and this and one is this then this - and the difference between them is a space
        - -> you can use this for - e.g if there is a style you want to override - you can define another style and give it a higher specificity
    - -> if you have two css classes - you can override the styling by just putting an id there <- because no matter the amount of css classes targeting one element, the ids always take precedence
- -> minimise the numbers of ids you use to make the code reusable
  - they can only be used once - but they have the highest specificity

- Battle royale: specificity with a tie
  - if you have two classes which are targeting the same element and they have the same specificity, change their order to change the styling of the element on the webpage which they're targeting
    - -> e.g if there are, in this case two classes targeting the same element
    - -> those two classes have the same specificity and are targeting an html element with two different styles
    - **-> you can calculate the css specificity from doing a tally, a literal table like the three which were above**

- **-> in this case, the style of the one which takes precedence is the one which was defined last in the css document** (or in the <style> html tag - which is for defining css in an html file)
    - so to undo this, you can just swap the order which the two classes are defined in in the css document etc
- -> ensuring that the classes have low but consistent specificity
    - the entire idea of avoiding using things like ids with very high specificity, because it makes the styles they provide harder to override - and thus the code less reusable
    - -> the next chapter is modular css with low specificities
        - low specificities - more reusable code
        - so conflicting rules are easier to manage / predict
            - -> when you use ids, they can't be overridden / are harder to
- -> exercise
    - -> in this example, there's a button
    - -> we're adding another one
    - -> but we want the background colour of the second one to be pink, and the first one to be green
    - -> so they're defining css which overrides the green of the overarching button class
        - -> nested classes?
    - -> how they do this is
        - calculate the specificities of the overarching button class
        - -> the entire idea behind specificity is that this style is targeting this specific element - it's how specifically this element is targeted (e.g an id is just for that one element, you can have classes which target the child elements of a class o.e)
        - -> then they've refactored those styles -> separated out the green from the button class -> and made it into two separate classes, and then defined a new pink button class - rather than id'ing the new button as pink
            - -> because ids have a higher specificity and we want to do it in a way which is as reusable as possible - the better method is to separate out the button class
            - -> the aim is to do it in a way which minimises the specificity as much as possible
            - -> codepen is another one
- Let's recap!
    - -> levels of specificity are defined in order of
        - inline styles
        - ids
        - classes, pseudo-classes, attributes
        - elements and pseudo-elements
    - -> styles are applied by the browser according to which style has the highest specificity attached to it (e.g if it's coming from an id)

4. Write selectors with BEM

- **notes on the video**
    - -> more time is spent reading code than writing it
    - -> you need to write code with reading it in mind
        - BEM <- block, element and modifier
        - these are categories of selector
- **notes on the text below the video**
    - Write with intent
        - -> scrolling through code / trying to remember how it works

- -> you need to write it with reading it being in mind
    - selector / class names
- Building blocks: blocks, elements, and modifiers
    - BEM
        - -> this is a css naming scheme
        - -> funcitonalities / hierarchies of different css hierarchies
    - a block
        - -> an element of the page which is independent from the rest
            - -> its css doesn't depend on the css for the rest of the website
            - -> parts of the project which make sense by themselves
            - -> blocks can have their own css
    - elements
        - -> these are the different parts in a block
        - -> doubble underscores__ are called dunders
        - -> **.proj-prev__heading <- this is an example of how an element is named with the BEM naming convention**
            - **-> the element is the heading, and it falls under the block (in this case) it's a button for the section which says 'previous project'**
        - -> modifiers
            - ones which changes the appearance of a block or element
            - -> these make, e.g different editions of buttons -> to override the larger button classes
            - -> how you name these using the BEM naming convention is .proj-prev--mint <- so it's the button and the -- is the modifyer
        - -> BEM
            - blocks are the entire elements
                - -> e.g sections of the webpage
            - -> E are the elements in those
            - -> M are the lines of code e.g the classes which would change the colours of those buttons
- Putting it all together: creating our navigation using BEM
    - -> in this example, he's used the different classes from the previous section all in one nav bar
    - -> the aim is to reduce the overall specificity, by breaking down the classes into sections which are as small as possible (within the use of the webpage)
    - -> in BEM, the IDE interprets the naming conventions with specificity
        - -> in line then IDs, classes
        - -> you want to reduce the amount of IDs you use as much as possible, because then the BEM selectors can be overwritten
    - **-> BEM syntax is .nav__link--active, another example is .nav__link a:hover**
        - -> you can use pseudoclasses <- these increase the specificity, but can reduce the amount of classes required
    - -> combinators
        - -> rather than having to make an entirely new class
        - -> ul.list or ul.list > li
            - the first one is targeting elements with the class called list
            - -> the other one is targeting the list elements under that
        - -> BEM is a naming convention for css elements
    - -> the next section is tools for more css structure
    - **-> refactoring the css -> converting the larger css classes into multiple BEM classes**
        - exercise -> https://codepen.io/nicolaspatschkowski/pen/dyjLyXB

- Let's recap!
  - -> BEM <- block, element, modifier
    - -> block <- units of code
      - named by their purpose
    - -> elements <- parts which make them up
      - __ are dunders -> form__label
    - -> modifier <- parts which modify them
      - -> button--green
  - -> sometimes classes aren't necessary <- parent child selectors

5. Use CSS Preprocessors for advanced code functionality

- **notes on the video**
  - -> BEM is a naming convention
  - -> it doesn't organise the code
  - -> this is what css precompilers are for
- **notes on the text below the video**
  - Creating a hierarchy in your css
    - -> indenting elements in the css
    - -> this makes it easier to see the hierarchy
    - -> nesting elements in css
    - -> relative blocks

```scss
.nav {
    padding-right: 6rem;
    flex: 2 1 auto;
    text-align: right;
        .nav__link {
            display: inline;
            font-size: 3rem;
            padding-left: 1.5rem;
                .nav__link--active {
                    color: #001534;
                }
        }
}
```

Indenting css using a visual hierarchy (sass)

- -> this is what scss does -> creates indentations in the css so that it's easier to read the hierarchy of information

scss

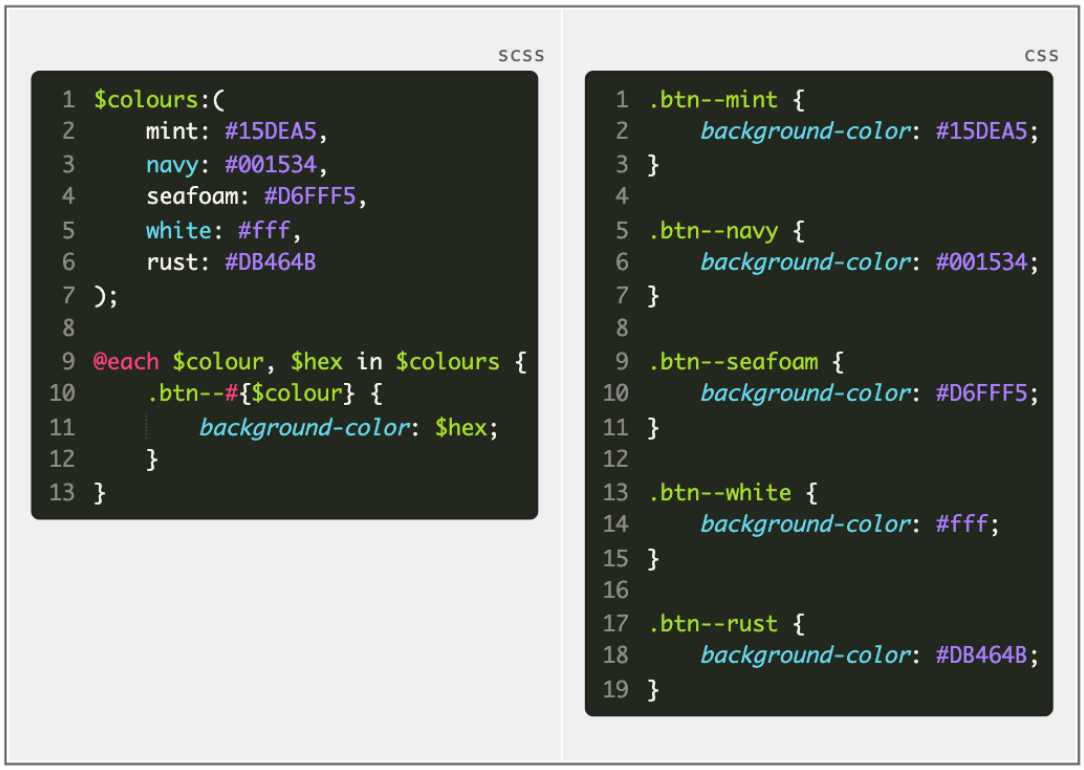
```scss
.nav {
    padding-right: 6rem;
    flex: 2 1 auto;
    text-align: right;
    nav__link {
        display: inline;
        font-size: 3rem;
        padding-left: 1.5rem;
        nav__link--active {
            color: #001534;;
        }
    }
}
```

css

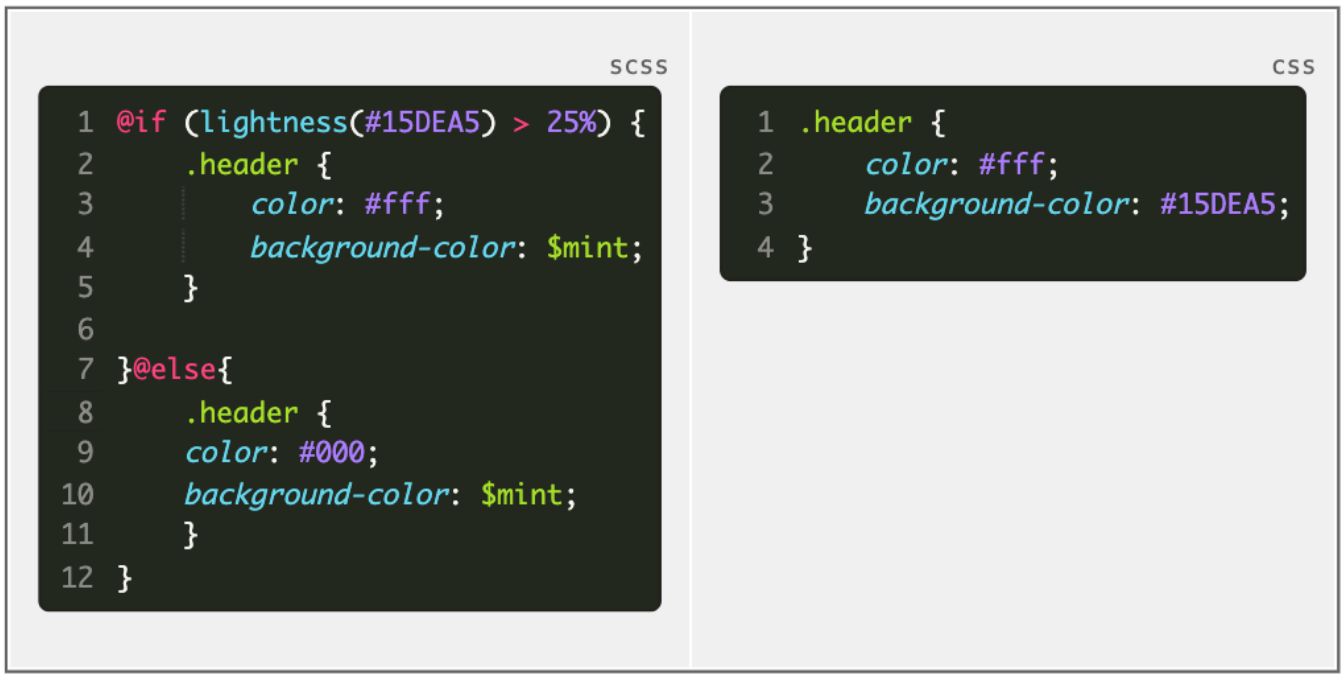
```css
.nav {
    padding-right: 6rem;
    flex: 2 1 auto;
    text-align: right;
}

.nav nav__link {
    display: inline;
    font-size: 3rem;
    padding-left: 1.5rem;
}

.nav nav__link nav__link--active {
    color: #001534;
}
```

  - -> pre-processors -> these compile the css into this syntax
  - -> storing colours and measurements into a single elements
  - -> variables
    - -> you only need to change the code once

- -> loops
  - -> these automate repetitive tasks
- **-> variables are in $ -> their valid is set once and it can be reused throughout the document**
  - for reusing the value
  - -> you only need to change the value of the variable once



Loops in saas, for automating repetitive tasks



Logical operations

- -> loops automate repetitive values
- -> logical operations
  - you can do something based off of the condition -> e.g if the background is a certain colour, you can change how the elements behave
- -> using pre-proccessors to automate the creation of selectors
  - i.e there are toold which we can use to make the css more concise -> variables, logical operations etc

- Meet the players
  - -> sass, less, and stylus
  - -> these three are similar
  - -> this course uses sass
  - -> the concepts are the same -> for css pre-processors

        - -> sass is the most popular professionally
        - -> next is the syntax for sass
    - Let's recap!
        - -> css pre-processors
            - nesting
            - units of code
            - -> these are for css
        - **-> Syntactically Awesome Style Sheets <- sass (the most popular css style sheet)**
        - -> these make css more programmatic -> e.g via the use of logic statements / variables which they introduce

6. Write SASS Syntax

- **notes on the video**
    - -> writing css with sass
- **notes on the text below the video**
    - Getting Started
        - -> sass syntax, how to write it
        - -> you can use sass in an IDE
        - -> this one is how to write sass
    - Two ways of doing the same thing
        - -> **either you can use .sass or scss file extensions**
        - -> you can copy / paste css into sass
        - -> opening a .sass file extension in an IDE, then pasting css into it
        - -> .scss <- this is in the standard css syntax
        - -> you can write css in a .scss file
        - -> declare the selector
        - when you write in the IDE in the .scss file, you can use wither .css or .sass syntax
            - -> below - this is the difference between them

```
.class-selector {
    color: white;
    background-color: black;
    }
```

But you also have the option of writi

```
.class-selector
    color: white
    background-color: black
```

```
.nav {
    padding-right: 6rem;
    flex: 2 1 auto;
    text-align: right;
    nav__link {
        display: inline;
        font-size: 3rem;
        padding-left: 1.5rem;
        nav__link--active {
            color: #001534;;
        }
    }
}
```

Sass just looks like indented slightly easier to read css <- code for styling the webpage which gets messy, more messy than the html when changes are made

- -> i.e, css can't take .sass syntax, but the other way round - this works / is true

        - -> .saas is more concise <- no {}'s, ;'s
        - -> test exercise
            - https://codepen.io/nicolaspatschkowski/pen/OJwGJjb
            - -> codepen -> create .test
            - -> then add two properties -.> background colour / padding
            - -> then add the test to the html div

- One syntax to rule them all
  - -> sass -> this is referring to a .scss file
  - **-> so sass -> refers to .scss not .sass here <- this file extension**
    - -> and not saas, which is software as a service
- Let's recap!
  - -> css code runs in .scss, aka 'sass' syntax
  - -> .sass files are more concise but are less commonly used than scss

7. Use nesting with SASS

- **notes on the video**
  - -> css lacks structure <- this is what sass is for
    - nesting
      - -> this creates a visual hierarchy
      - -> we can also group blocks of code together
- **notes on the text below the video**
  - Using selectors inside of selectors
    - -> in sass you can add indentations to the css

```
ul {
    list-style: none;
    text-align: right;
    li {
        display: inline;
        font-size: 3rem;
        color: #D6FFF5;
    }
}
```

    - <- example sass indentation in code
      - -> the li's are nested int he ul's
      - -> the child elements are under the parent elements in sass, as with the html
        - -> compared to a regular styles.css file
        - -> nesting like this in comparison to css means everything is in one place
  - -> you could, in css do something like ul li <- this is called a descendant combinator
  - -> he's mentioned the structure which sass has, and then the descendant combinators which css has -> ul li and then ul
  - exercise
    - -> https://codepen.io/nicolaspatschkowski/pen/JjBVeqg?editors=1100
    - -> there is a parent and there is a child selector
    - -> he's just using the parent and child combinators in css
    - -> but in sass
    - -> he continually uses codepen throughout the course <- this website
  - -> selectors separated by spaces <- these are descendant combinators (no space means this and this, and a space between classes in css means style this element nested inside this larger one)
    - -> MDN has documentation on this
  - -> other types of combinators
    - -> descendant combinators
      - .parent
        - -> anything related to the parent element
      - .parent .descendant
        - -> this targets the second element which is nested inside the parent element
      - .parent > .child
        - -> when the second element is a child of the first element
      - .parent + .adjacent
        - -> when the second element comes after the first

- -> i.e we're targeting the element which is adjacent to the first one
- -> you can technically use all of those lines of code in the same css file

- -> you can use these in sass -> by embedding them so that they take the same form they do in html
  - .parent {
  - background-color: #15DEA5;
  - .descendant {
  - color: #fff;
  - }
  - >.child {
  - color: #D6FFF5;
  - }
  - +.adjacent {
  - color: #001534;
  - }
  - }
  - -> you are nesting everything that would normally come off of one of these elements in css -> in sass, you are nesting that underneath the element in css
  - -> this is adding a css selector to the parent selector
  - so there is a root selector <- selector is another term for class in css (you are changing the styles of certain elements with it)
  - -> the combinators are the +, > et al in the example code above

- -> descendant vs child element
  - **-> it's the same as in family trees - the child of the parents is the descendant of the grandparents, but not the child of the grandparents**
  - **-> the child element comes directly after the parent element**
- -> targeting the root selector in css
  - -> the root selector is the default one
    - this is in the "code base"
  - -> nesting and combinators to target it
  - -> the task is to reuse the previous sass but for the root selector
  - -> to embed the child selectors under that
  - -> ~ is the general sibling combinator
  - -> .adjacent <- for the child element which comes after it
  - -> nesting child selectors under it in sass
    - -> > is the child combinator in sass
    - -> .descendant is for the descendant
    - **-> it's Sass, not sass <- this is for indented css**
      - -> **not SAAS, that's software as a service**
    - -> directly nesting the selector -> this is all you need for the sass to be taken as the descendent of the element it's nested under

- Enter the ampersand
  - -> this is in Sass, if you don't want separation between the parent and child
  - -> in this example
    - he's added an li:hover pseudo-class to the list elements
      - -> pseudo-class because it's being hovered over

- -> this example is in Sass <- the hover state of the list elements has been nested under the list element
- -> the language of "which compiles to" means -> this is the more complicated css which would achieve this same thing in Sass (the condensed syntax css version)

```
ul {

    list-style: none;
    text-align: right;
        li {
            display: inline;
            font-size: 3rem;
            color: #D6FFF5;
            :hover {
                color: #001534;
            }
        }
}
```

- this is it, the condensed syntax css version ->
  - Sass nesting created combinators -> it's targeting the children under that parent element which the css is nested under
    - -> nesting css in a Sass class implies that the thing being nested is a child element
    - -> you can nest something without it being considered a child element if you use &'s in the nested code
- -> & <- this is for concatenating the parent and child selectors

```
ul {
    list-style: none;
    text-align: right;
}
ul li {
    display: inline;
    font-size: 3rem;
    color: #D6FFF5;
}
ul li :hover {
    color: #001534;
}
```

  - so this is the Sass ->
    - in other words, it stops the hover being taken as a child element of li, and instead treats it as a separate element in css -> this is in Sass, so if there were no & over :hover, then nothing would happen because nested elements in Sass are treated as child elements (and not otherwise)

```
ul {
    list-style: none;
    text-align: right;
        li {
            display: inline;
            font-size: 3rem;
            color: #D6FFF5;
            &:hover {
                color: #001534;
            }
        }
}
```

- task
  - -> not wanting to nest pseudoclasses in Sass
  - -> & can be used to join the parent to the child being nested
  - -> another example is with the .input class selector
    - in this case, it's nesting a :focus state under it

- Nesting: a dab'll do ya
  - -> you can carry on nesting in Sass -> there is no limit
  - -> don't completely replicate the nesting structure in Sass of the html
    - -> the html will change
    - -> and nesting increases the specificity
    - -> you want the css which gets the job done and has the lowest possible specificity
      - -> the aim is to create code which can be modified / overridden
      - -> the problem with very specific selectors -> e.g .parent-div .child-div .grandchild-div
        - they require even more specific selectors to override
        - -> the code is less reusable / maintainable
        - -> tripe css is css which is too specific and can't be reused for other elements
        - -> you'd need to refactor the code
        - -> only writing nested css relative to the root
        - -> **try not to nest more than two css elements deep -> because then it will increase the specificity, which makes the code harder to reuse**
          - **you want generic and al least specific css as possible - so it can be reused / overridden more in case the client wants something changed**
  - exercise
    - -> moving nested elements up one css selector
    - -> particle divs showing up
    - -> nesting in Sass
      - it increases specificity
      - -> the entire idea of BEM is to reduce the specificity
      - -> you can use both BEM and Sass
      - -> combining them using flat selectors
      - **-> Sass used nesting which increases specificity and BEM is a css naming convention which reduces it**
- Let's recap!
  - -> Sass uses nesting
  - -> css combinators
  - -> don't create selectors which are too specific (notes above)

8. Use BEM selectors with SASS

- **notes on the video**
  - -> to organise css
    - BEM <- naming conventions -> use / relation to other selectors
    - sass <- not saas, software as a service, sass is sassy for css
    - hierarchies
    - -> nesting BEM selectors in sass
- **notes on the text below the video**
  - Keeping things flat
    - from before
      - -> nesting Sass
      - -> so BEM is how you name selectors and .scss files, aka Sass is a version of css where the css can be nested, to improve the syntax
        - -> but the nesting increases the specificity of the elements -> compared to the BEM naming conventions which reduce the specificity back down
        - -> you want the specificity to be small - because this makes the code reusable
        - -> this is about how you combine those two methods, BEM and Sass

- he's playing with different ways to combine BEM and Sass
  - -> BEM <- the naming conventions (splitting one class in css into many smaller classes so that css isn't reused -> naming each of these classes according to a convention)
    - -> reduces the specificity (the code is less specific to one element, because it can be reused)
  - -> Sass <- the .scss file extension compared to the .css file system
    - -> the one which indents the css so that it is more modular, like other coding languages - more readable, but which increases the specificity
  - -> one way to combine them:

```
.block{
    background-color: #15DEA5;
    .block__element {
        color: #fff;
    }
}
```

    - -> (above) this is the BEM naming convention for css, nested under another css class as in Sass
    - -> this wouldn't work in a .css class
    - -> in this example
      - he's calculated the specificity of the elements <- in another table like approach
      - -> the common thought process is to calculate the specificity of the elements in a table like this
        - -> we are trying to reach the code with the lowest specificity / repeats, but which is repeatable and gets the job done
        - -> it's written like this 0/0/1/0
        - -> CID <- classes, IDs, elements
      - -> when you're combining something which increases the specificity of elements (Saas) with something else which decreases the specificity (BEM), you need to make sure that the overall effect is one which reduces the specificity -> or just so that it's below that of the other styles targeting that element
        - -> so if you're breaking down the css class into a lot of smaller classes (e.g a style would be classed as a BEM modifier), it might not work because the indentations in the Saas which you're combining it with increase the specificity - in comparison to regular css
        - -> inconsistencies between selectors -> unpredictable results
          - when considering which styles can and can't be overridden in css
      - -> how you nest css classes in Saas, without messing up BEM
        - -> i.e in BEM, splitting classes into different functions works less well is the code is nested, because nesting the code messes with the css specificity_
        - -> the solution to this is to use &'s
          - -> the parent and child are joined together in the compiled css
          - -> the child can be the element which you are targeting
          - -> e.g in this nested css (aka the same as Saas) -> they've used &__name-of-element
          - -> block only needs to be written once for this

        - -> you get the benefits of BEM and Saas -> using &'s when combining indented BEM modifiers (aka things which target the style of certain elements) gets rid of the specificity increases which happen when you nest elements - so certain styles can be overridden without such unpredictable behaviour
        - -> 0/0/1/0 <- this is an example of flat specificity - this is the aim (something more along these lines, rather than using &'s)
    - exercise
        - -> Sass nesting
        - -> focusing it with the & to control the specificity
        - -> then refactoring -> aka taking a class and splitting it into smaller parts / features / functions using BEM
            - -> doing this when combining Saas and BEM
            - -> but using &'s
                - we're taking classes
                - -> splitting them into smaller constituents using BEM
                - -> putting &'s in them when nesting them using Saas
                    - to avoid reducing their specificity
                    - -> so changes in the css / Saas can be overridden without such unpredictable behaviour
        - -> Sass nesting, with flat specificities
        - -> break down the class into BEM
        - -> then nest it
        - -> then put in the &'s to reduce the specificity
            - the aim is to "flatten" the specificity
                - -> in other words 0/0/1/0 in this example, to write the Saas / css in a way which makes the css as reusable and overridable as possible
- Save specificity for when you need it
    - -> the aim isn't to completely flatten the specificity, just to minimise it
        - -> and then to use it when you need it - to override certain properties
    - -> in this example, there's a wider button class and there are two buttons
        - he's defined a new class which changes the colour of one of the buttons
        - -> and it's been listed in the css -> last, so that it takes precedence
        - -> in this one, in Sass -> they've used a similar approach, but with nesting instead
            - &-- {
            - }
            - -> for example
        - -> so he's defined the button class in css, and then under it nested &-- the names of the two different types of buttons
        - -> we are using the & because -> nesting would increase the specificity of the code, so & is used to override this - the indented elements in the Sass are seen as the child elements of the class which they're nested under
        - -> the styles which are listed under the nested & elements -> these take precedence for those elements (the colours of the buttons in this case)

```
.btn {
    display: inline-block;
    margin: 0 auto;
    background: #15DEA5;
    padding: 1rem;
    &--disabled {
        background: grey;
    }
    &--outline {
        background: transparent;
        border: 2px solid #15DEA5;
        &.btn--disabled{
            border: 2px solid grey;
        }
    }
}
```

        - -> in this example, he's broken down one class into four smaller classes using the

BEM approach / naming conventions
    - -> exercise
        - -> a standard button class and a modified button element
        - -> the naming conventions used are in BEM
        - -> it's listing out the requirements for the new button
  - Put your specific selectors into HTML
    - -> in this example
        - solid and outlined buttons
        - -> so we have html for a button with a green background, and in the other case there is only a green outline
        - -> he's just defined multiple classes of css styles -> these change the styling of the buttons - he's applying multiple of the css classes to the same button
    - -> height and specificity
        - you need to use them carefully, or the 'apply multiple css classes' might backfire because of the hierarchies in css
  - Let's recap!
    - -> inconsistencies in selectors / specificities
    - -> &'s to nest in Saas to avoid messing up the specificity which comes with nesting elements in css
    - -> nesting and specificity to create a maintainable selector system

Quiz: Structure your code efficiently

- -> there are 12 questions
  - .block--modifier {...}
  - -> the BEM syntax is .block__element--modifier
    - you can just look this one up
    - -> and then when naming the elements, its-done-like-this
  - -> ids have a higher specificity than classes do
  - -> when you take sass and 'compile' it 'as css' -> then it's asking what css the Sass breaks down into (quite a few of the questions are asking this, the ability to translate from css to Saas with its hierarchies)
    - Sass is css which is indented
    - -> so this is asking what css is equivalent to that Saas
    - -> CSS syntax is also valid .scss syntax (you can paste css into a .scss file and it will still be considered valid/readable)
    - -> the &'s in Saas are to stop the specificity from being a problem
    - -> __ is called a dunder under the BEM naming convention
  - -> Sass is all about the hierarchical nesting of css and BEM is a naming convention which it uses to break larger classes into smaller ones, to avoid the reuse of code
  - -> the Sass classes with the most indentations generally have higher specificity, rather than the ones which don't use &'s
    - -> &'s are to stop the css from unnecessarily increasing
    - -> they combine to the selector's parent in the hierarchy <- when they compile to css
    - -> the greatest danger of nesting in Sass is that it can create overly specific selectors