We've been using Sass variables throughout this series. We've discussed the various different datatypes and you should now be comfortable managing all sorts of different values across your styles.

One aspect of using variables in any programming language is variable scope. Sass is no different and in this video we'll cover all aspects of variable scope in Sass.

- The default scope of variables
- Scope in nested styles
- Scope variables in functions and mixins
- Scope in @if and @else conditions
- and finally: variable flags

Default Variable Scope

When defining variables in Sass, they can come in two types of variable scope: global scope and local scope.

Code example

```
$color: rgba( #f00, 0.4 );

body, .box {
  display:flex;
}
.box {
  width:100px;
  height:100px;
  background:$color;
  border:5px solid $color;

  h1 {
    margin:auto;
    color:$color;
  }
}
```

If we declare a variable outside of any selectors, functions or mixins it's in the global scope and can be used in multiple places throughout

our code.

Here my `$color` variable is being used as the background and border colour of my box and as the colour of the `h1` text.

This is probably the most common use of Sass variables and it enables you to use the same value across multiple partials, in multiple selectors and have a single line of code to change if you want to adjust the value. I could change the colour from red to blue and that change is reflected in every place the variable is used.

Nested Variable Scope

When usign variables in nested selectors, the nesting creates a local scope for any variables declared inside the parent selector.

If I move the `$color` variable declaration into the `.box` nested selector, everything still works as it did before but the `$color` variable is now only available within the `.box` class.

If I try to use this variable outside of the nesting structure, Sass throws an error because that variable is undefined for that part of the file.

Similarly, when working with nesting and variables, child selectors have access to variables in parent selectors - as we saw before. But parent selectors don't have access to variables in child selectors. This shows us that each selector in a nested chain can create a local scope for variables declared within each selector.

Even moving the order of the styles around show that it's not source order that determines which variables are applied, it's the scope that has the biggest affect on what is eventually rendered in the browser.

Variable Scope in Functions & Mixins

Just like local scope in nested selectors, functions and mixins create their own local scope too.

If we create a mixin that outputs a color property from a variable local to the mixin, we'll see that this local variable will always take precedence to external variables - even if they have the same name

```
@mixin solid-border {
    $color: red;
    border: 5px solid $color;
}
.box {
    @include solid-border;
```

```
}
```

To make this mixin more flexible we could define it so it takes a `$color` parameter. Again, the local scope of the mixin variable will always apply even if the variable name is used elsewhere.

```
@mixin solid-border( $color ) {
    border: 5px solid $color;
}
.box {
    @include solid-border;
}
```

The same can be said for functions. We could rewrite this mixin as a function which returns the border color to set.

```
@function solid-border( $color ) {
    @return 5px solid $color;
}
```

This approach would be a little excessive for setting a border value in a real-world project but I'm just using it to briefly demonstrate how variable scope works in functions and mixins.

Scope in Conditionals

Like in many programming languages, logical flow control - if, else and else if - also creates local scope. But there are a couple of things to watch out for.

```
h1 {
    $color: red;
    @if true {
        $color: blue;
    } @else {
        $color: green;
    }
    color: $color;
}
```

The following snippet of Sass conditionally sets the color of an `h1` tag. We have a local `$color` variable defined inside the selector and then a conditional that changes the color.

If the condition is `true` the color is blue. If the condition is false, the color is green.

This is all good and logical. But something strange happens if we move the variable declaration out into the global scope.

```
$color: red;
```

```
h1 {
    @if true {
        $color: blue;
    } @else {
        $color: green;
    }
    color: $color;
}
```

Now the text color is always red. This is because the global variable is not overriden in the local scope of the if or else block. To make this setup work, we have to move the setting of the `color` property inside the conditional statement.

```
$color: red;
h1 {
    @if true {
        $color: blue;
        color: $color;
    } @else {
        $color: green;
        color: $color;
    }
}
```

Because the braces of the conditional creates a local scope, we need to declare the property and value within that local scope if we expect to see the right colors. It's a bit tricky to get your head around at first and is probably the least intuitive aspect of variable scope in Sass.

Variable Flags

In a previous video we looked at the Sass `!optional` and `!default` flags. There is one more that's useful to look at when discussing variable scope: the `!global` flag.

This can be appended to the declaration of any variable in local scope to make it a global variable.

If we go back to our example from earlier when Sass was throwing errors about undefined variables, we can illustrate how this works.

In this example, we have a locally scoped `$color` variable within the nesting of the `.box` selector. Currently this variable is not available outside of this nesting so the use of the variable in the `body` tag selector below is throwing an error.

If we add the `!global` flag to the end of the declaration, this is then made available to the body tag and the background of the page is given

a faint tint.

The rules of variable scope still apply and if we redefine the `$color` variable elsewhere in a local scope, it will override the global value. If a second `!global` flag is added then the last one is used and the global scope is overridden.