There aren't many options when it comes to the letter Z so we're revisiting a classic: z-index.

We could have talked about the Sass `zip` function which combines lists into multidimensional lists but I've never really found much practical use for it.

Managing z-index across a project is a much more common concern and something we can definitely leverage Sass for. In this video we'll look at:

- Organising layers with Sass map variables
- and Creating a function to reference layers by name

Organising z-index

As discussed in the z-index epsiode in the first AtoZ CSS series, having a system for z-index can be useful to bring order and consistency to your code.

I mentioned that I like to use increments of 10 for creating different z-index values so it's more organised that picking random values like `z-index:57` or `z-index:9999`. Using increments of 10 means you can create different layers at `z-index` 0, 10, 20, 30 and so on and if you suddenly need to slot in a new layer between two existing ones you can pick multiples of 5: 5, 15, 25. This keeps the code organised but also flexible and maintainable.

To further improve code organisation we can use Sass map variables. Let's take a look.

I've got demo here with three different coloured boxes positioned on top of each other. At the moment the stacking order is determined by the HTML source order as no `z-index` values have been set.

In the Sass, let's first create a variable called `$layers` which will be a map variable - for more info on map variables check out the previous episode on Working With Sass Map Variables.

```
$layers: (
    1: 10,
```

```
    2: 20,
    3: 30,
    4: 40,
    5: 50
);
```

This variable allows us to reference layers by number and behind the scenes outputs a z-index value in a multiple of 10.

We could extend this variable to contain other named values instead of just numbered values which can be useful if you know there will be certain elements that you want to use in the project.

I often like to add a behind layer which uses negative z-index and layers for headers and modals if these will be used.

```
$layers: (
    behind: -1,
    1: 10,
    2: 20,
    3: 30,
    4: 40,
    5: 50,
    header: 100,
    overlay: 200,
    modal: 300
);
```

Having all the layers abstracted away into a variable like this means to change or adjust z-index values throughout the development process doesn't require changing values across partials; we can just modify the z-index values in our variable.

So, with this variable created, how would we use it?

In this example we have three boxes and if I wanted to change their stacking order we could do so as follows:

```
.box1 { z-index: map-get( $layers, 3 ); }
.box2 { z-index: map-get( $layers, 1 ); }
.box3 { z-index: map-get( $layers, 2 ); }
```

By using the Sass `map-get` method we can fetch the value from the `$layers` variable by its key. In the first example we go to the `$layers` variable and as for key 3 which will output `z-index:30`.

This works but repeatedly typing `map-get` and `$layers` might get a bit tedious and we can do better.

Creating a Layer Function

Instead of this repetition in our Sass, we can create a Sass function to output the z-index. We can then also ensure that we handle any errors that might occur from referencing invalid layer names and warn the user accordingly as we saw in the Warn Error Debug episode.

To clean up this code, first let's define the function

```
@function layer( $name ) {

}
```

The function takes one argument for the name of the layer we want to output. Within the function we simply return the value from `map-get`.

```
@function layer( $name ) {
    @return map-get( $layers, $name );
}
```

Having created this function we can update our Sass to use it. Instead of manually calling `map-get` we substitute it for the `layers` function and pass in the name of the layer we want.

```
.box1 { z-index: layer( 3 ); }
.box2 { z-index: layer( 1 ); }
.box3 { z-index: layer( 2 ); }
```

To make this function even more useful, we can add some conditional logic and some error handling.

If I call the layer function with a layer name that doesn't exist, Sass doesn't output the invalid value but fails silently - which isn't very useful. Let's modify the function to throw an error if we pass in an unknown layer name.

```
@function layer( $name ) {

    @if map-has-key( $layers, $name ) {
        @return map-get( $layers, $name );
    } @else {
        @error "layer #{ $name } not found in: #{ map-keys( $layers ) }";
    }

}
```

First we check that the layer name exists by using the Sass `map-has-key` method. This returns true or false depending on whether the key exists in the map.

If the map does have the layer name then return the corresponding value.

Else, use `@error` to alert the user. We can even use string interpolation to output the name of the layer we specified and the list of available keys in the map. This would be useful to let the user know if they made a simple typo or tried to use a non-existent layer name.

While this system of z-index management is certainly more complex than just plucking random values out of thin air, having an organised and programatic approach to authoring your code is a great way to keep your mind organised and your code maintainable.

Sass is the perfect partner for helping you achieve this with minimal effort. As it's a third-party tool and not native CSS it's possible that one day Sass will be replaced with another tool or that it's functionality eventually winds up as native to the CSS language. Whatever happens, learning to think like a programmer and solve problems in a systematic way will definitely stand you in good stead for your future career.