In a previous video we discussed the various data types of Sass variables.

One variable type we didn't have time to go into detail on is map variables but the time has come to fix that.

What Are Map Variables?

Map variables are a store of key and value pairs. They're a feature of Sass 3.3 and above and are probably one of the biggest game changers Sass has seen in recent years.

They are similar to list variables, but each item in the list also has a unique key or label.

A Sass map looks very similar to Javascript object or like the key:value pairs in a CSS style block:

{% highlight scss %}
$map: (
key1: value,
key2: value,
key3: value
);
{% endhighlight %}

The series of key:value pairs are separated by commas instead of semi-colons and are wrapped up in a pair of parentheses to group all items within the map together.

Map variables can be used to store any valid Sass datatype and you can even have a map of maps.

{% highlight scss %}
$settings: (
colors: (
brand: #cc3f85,
copy: #222,
heading: #333
),
font-families: (

```
heading: 'Avenir', arial, sans-serif;
copy: 'Avenir', arial, sans-serif;
),
font-sizes: (
h1: 50px,
h2: 40px,
h3: 25px,
p: 18px
)
);
```

Storing information in these more complex data structures means we need a new set of tools for accessing values from within the map. Let's take a look.

Using Map Values

Let's take a map of font-sizes that might be used to set up some sensible defaults for a project.

```
$font-sizes: (
h1: 50px,
h2: 40px,
h3: 30px,
h4: 20px,
p: 16px
);
```

To access a value in the map, we refer to it by its `key`. So to get the value of `50px` from within the `$font-sizes` map, we would use the following snippet:

```
h1 {
font-size: map-get( $font-sizes, h1 );
}
```

The `map-get` function is built into Sass and takes 2 parameters: first the map we want to get a value from and then the key.

If we have a whole series of values that we want to get out of the map, we could perhaps combine this with an `@each` loop like we saw in the previous video. In this case, Sass knows that we're iterating over a map and we don't need to use the `map-get()` function.

{% highlight scss %}
@each $selector, $font-size in $font-sizes {
#{ $selector } {
font-size: $font-size;
}
}
{% endhighlight %}

Querying and Merging

When building complex components or library code, it may be necessary to check that a map contains a particular key before trying to get the value or manipulate it in some way.

If you try to use a key that isn't found, Sass will throw an error.

For this purpose, Sass has a `map-has-key()` function which returns `true` if a key exists and `false` if it does not. This can be combined with a conditional statement as follows:

{% highlight scss %}
$settings: (
media-queries: true,
enlarge-text: 120%
);

html {
@if map-has-key($settings, enlarge-text) {
font-size: map-get($settings, enlarge-text);
} @else {
font-size:100%;
}
}
{% endhighlight %}

Another feature of maps is the ability to merge two maps into one or to add new keys into an existing map. I've not found the need to do this on many occasions but I'll include it here for completeness.

Imagine we have a button mixin which has some default settings for color, background, padding and amount of border-radius.

{% highlight scss %}
@mixin button {
$settings: (
color: #fff,
background: #000,
padding: 0.5em 1em,

```
radius: 0
);

display:inline-block;
text-transform:uppercase;

@each $property, $value in $settings {
    #{ $property }: #{ $value }
}

}
{% endhighlight %}
```

We could make this mixin more flexible by letting it accept an optional map of additional options to change how it looks and then merge our `$options` and `$settings` together before outputting the series of properties and values.

```
{% highlight scss %}
@mixin button( $options:() ) {
$settings: map-merge( $settings, $options );

@each $property, $value in $settings {
    // as before
}

}
{% endhighlight %}
```

Not only does this let us extend our button, it means we can override any of the default settings with our own custom options. This is because a map can only contain unique keys; if both settings and options contain the key `color` then the second color will override the first.

Practical Example

Dealing with complex structures like maps can seem a bit abstract outside the context of a real world example so let's fix that.

If you watched the previous episode about loops and control flow you may remember the example of creating a series of colour swatches for a style guide.

At the end of that video, this is the code we were left with. The crux of the exercise was to take two list variables - one of colour names and one of hex values - to output dynamic swatches of colours used within this hypothetical project.

We can improve on this by combining the two list variables into a single map variable which will greatly clean up the code.

I'll create a map variable called `$colors` where each key is the colour name and each value is the hex code.

{% highlight scss %}
$colors: (
white: #fff,
red: #cc3f85,
green: #9be22d,
blue: #66d9ef,
black: #000
);
{% endhighlight %}

Now we need to update the loop to pull out the name and colour for each item in the map as follows:

{% highlight scss %}
@each $name, $color in $colors {
.swatch–#{ $name } {
background: $color;

```
    @if lightness( $color ) < 50% {
        color:#fff;
    }

    &:before {
        content: "$color-#{$name}"
    }
    &:after {
        content: "#{$color}";
    }
}
```

}
{% endhighlight %}

We've combined two variables into one and removed the need to fetch values from a list based on their `nth()` index. All in all, a much neater solution and one that's easier to maintain.