Over the first half of this series, we've looked at the some of
the major features of Sass and a number of practical examples of putting
Sass into action.

Today we're going to look at two smaller features of the language that
have a very specific use case.

Optional Extends

We covered the `@extend` directive back in Episode 5 of AtoZ
Sass and covered extending classes and
silent placeholder selectors.

Here's a quick recap of how it works:

I've got an example here with a base class of `.message` which sets up
some visual characteristics for a notification message.

```
{% highlight scss %}
.message {
padding:1em;
background: #eee;
border:1px solid #ccc;
}
{% endhighlight %}
```

If our project needs additional types of message - perhaps for
a success message or an error message - we could create additional
classes and then have them inherit the base notification styles via `@extend`.

```
{% highlight %}
.success-message {
@extend .message;

background:lightgreen;

}
.error-message {
@extend .message;

color:#fff;
background: red;
}
{% endhighlight %}
```

This produces a comma separated list of selectors that share the same properties and additional selectors that contain any unique properties or values.

```
{% highlight %}
.message, .error-message, .success-message {
padding:1em;
background: #eee;
border: 1px solid #ccc;
}
.success-message {
background:lightgreen;
}
.error-message {
color:#fff;
background:red;
}
{% endhighlight %}
```

But imagine the situation where this `.message` module is being used from a 3rd-party library or framework. Perhaps this is unintentionally removed or perhaps it's loaded in the wrong order. When our Sass compiles, it would not be able to `@extend` the `.message` class and it will throw an error.

To mimick that behaviour, I'm going to comment out the message class and save. The compiled CSS now contains the error that the `.success-message` class failed to `@extend` the `.message` class.

To silence this error and have our Sass code fail gracefully, we can add the `!optional` flag to the end of the `@extend` statement. This prevents the compile from failing and we can continue on our merry way.

```
{% highlight scss %}
.success-message {
@extend .message !optional;

background:lightgreen;

}
{% endhighlight %}
```

However, I'm not too sure this is a good idea.

One of the great things about Sass is that any errors in our CSS are caught by the compiler. If we make a mistake, we can fix it and ensure that the styles we write will be valid and eventually painted to the screen.

Having errors go undetected like this doesn't sound like a smart idea to me but perhaps I'm missing the point - if I am, please get in touch and let me know!

Default Variables

Another "flag" available to Sass authors is the `!default` flag.

This is used in conjunction with variables and allows a variable to be given a value if it's not already been defined.

If it has already been defined, it will use that value instead.

Let's take a look at an example of using variables in this way.

Imagine we are creating a simple starter template for new projects and a number of the key settings for the site are controlled via global variables for things like the primary and secondary colours, base font size and heading sizes.

```
{% highlight scss %}
// Site settings

$primary-color: red;
$secondary-color: blue;
$base-font-size: 16px;
$h1-font-size: 50px;
{% endhighlight %}
```

We would normally create separate partials for the variables and the styles but I've kept them in the same file here so you can see everything on a single screen.

Here are a series of variables with their initial values. If we want to override any of these values, we can just redeclare the variable below and change the value.

```
{% highlight scss %}
// Site settings

$primary-color: red;
$secondary-color: blue;
$base-font-size: 16px;
$h1-font-size: 50px;

// Overrides

$primary-color: pink;
$secondary-color: lightblue;
$h1-font-size: 80px;
{% endhighlight %}
```

The variables further down the file (or lower in the source order) override the ones above.

However, if we make the inital settings all `!default` variables, the source order doesn't have any effect at all. The variables without the default flag will always win.

```
{% highlight scss %}
// Overrides

$primary-color: pink;
$secondary-color: lightblue;
$h1-font-size: 80px;

// Site settings

$primary-color: red !default;
$secondary-color: blue !default;
$base-font-size: 16px !default;
$h1-font-size: 50px !default;
{% endhighlight %}
```

Any default variables that are not redeclared just use their default value.

The `!optional` and `!default` flags aren't the most powerful or even the most commonly used Sass features but they are often used in CSS frameworks and boilerplate code. You may have seen them around but not been entirely sure what they do. Well, now you know.