In this episode, you'll learn all about **extends** and how they're different to **mixins**, how to create and extend placeholder selectors, and about some of the pros and cons of using extend in real projects. To understand extending selectors, let's first have a quick recap of the mixin directive.

I have an example here with a mixin that sets up some default font styling. To use the mixin we declare `@include` followed by the mixin name within another CSS selector. And the styles from within the mixin are output at that point in the code when the CSS compiles.

```
@mixin uppercase-letter-spacing {
    text-transform:uppercase;
    letter-spacing: 2px;
}
.page-title {
    @include uppercase-letter-spacing;
}
```

In the HTML I've got three different elements which I want to apply the mixin to.

```
<h1 class="page-title">
    lorem ipsum
</h1>

<p class="intro">
    lorem ipsum dolor <span>sit amet</span>.
</p>

<p>
    lorem <b class="highlight">ipsum</b> dolor sit amet.
</p>
```

This example is quite compact but imagine these selectors being used in multiple Sass files or separated by a number of lines of code.

When using mixins the code is duplicated within each selector and in this case, the generated CSS has the same number of `text-transform` and `letter-spacing` declarations for each time we include the mixin. This is all well and good but can lead to a lot of duplication in the compiled CSS.

Now let's look at using `@extend` instead.

Extends are used to have one selector inherit all the styles of another selector and the compiled CSS is a comma separated list of all the selectors that share the same properties.

So instead of creating a mixin for the uppercase and letter spaced text, we could create a class instead.

```
.uppercase-letter-spacing {
    text-transform:uppercase;
    letter-spacing: 2px;
}
```

And then have all our other selectors inherit these properties via `@extend`.

```
.page-title {
    @extend .uppercase-letter-spacing;
}
.intro span {
    @extend .uppercase-letter-spacing;
}
.highlight {
    @extend .uppercase-letter-spacing;
    font-weight:bold;
}
```

Again, do imagine that these selectors are separated by many lines of code or in different files - I'm keeping everything compact here so you can see what's going on.

When the CSS compiles, one long selector chain is produced and comma separated.

Therefore the major difference between `@extend` and `@mixin` is what is output as CSS by the compiler.

```
.uppercase-letter-spacing, .page-title, .intro span, .highlight {
    text-transform: uppercase;
    letter-spacing: 2px;
}

.highlight {
    font-weight: bold;
}
```

Placeholder Selectors

In the previous example we looked at creating a class which had its styles inherited by a number of other selectors throughout a codebase.

But what if the `uppercase-letter-spacing` class was never designed to be used on its own and never intended to be added as a class to elements in the HTML.

In this case, we can create something called a placeholder selector which is never compiled into CSS until used in conjunction with `@extend`.

Placeholder selectors are defined with the `%` percent character to differentiate them from other selectors like class (`.`) or id (#). To rewrite our previous example with a placeholder selector, we'd use the following snippet:

```
%uppercase-letter-spacing {
    text-transform:uppercase;
    letter-spacing: 2px;
}
.page-title {
    @extend %uppercase-letter-spacing;
}
```

When the CSS compiles, there is no selector generated with the name "uppercase-letter-spacing" but this silent selector can be used within the Sass.

Adding the extends back in, we now get the following result in the compiled CSS:

```
.page-title, .intro span, .highlight {
    text-transform: uppercase;
    letter-spacing: 2px;
}

.highlight {
    font-weight: bold;
}
```

So, when extending normal selectors like classes, the selector being extended will be compiled into CSS but when using placeholders, only the result of the extend will be generated.

A great use-case for extend is when applying clearfix to prevent container collapse when building floated layouts. So instead of creating a clearfix class, create a clearfix placeholder selector:

```
%clearfix:before,
%clearfix:after {
    content: " ";
    display: table;
}
%clearfix:after {
```

```
    clear:both;
}
```

And then instead of adding classes to your HTML, just extend the placeholder where necessary:

```
.image-grid {
    @extend %clearfix;
}
.portfolio-items {
    @extend %clearfix;
}
.share-this {
    @extend %clearfix;
}
```

I use this technique myself to avoid cluttering up the HTML with utility classes and I've found it to be a great use-case for extend.

Pros and Cons of `@extend`

So this extend thing sounds great, right? By using extend everywhere we can reduce the amount of code generated and have a super-lean, super efficient comma-separated list of all the selectors that share the same properties. Right?

Well, you may be able to tell from the tone of my voice (or not, sometimes people complain that they don't get my sarcasm) that using extend is not always a great idea.

Firstly, what are the pros?

Well, we've already seen the main one which is that the compiled output is fewer lines of code - all the shared properties are applied to one, automatically generated comma separated series of selectors.

Another positive reason for using extend is that you can reduce the number of classes applied to an element in the HTML because you can have a single class inherit the behaviour of multiple classes. This can help bring more meaning to your class names too. So, instead of:

```
<div class="col medium-4 large-6 last"></div>
```

We could have a more meaningful class name that exhibits the same behaviour:

```
<div class="latest-posts"></div>
```

Which is an extension of:

```
.latest-posts {
    @extend .col;
```

```
    @extend .medium-4;
    @extend .large-6;
    @extend .last;
    /* other styles below */
}
```

But. All of this does come with some downsides.

There is a balance to be struck between meaningful class names and how easy the code is to understand when being read by your team mates.

Reading the class name `latest-posts` gives a sense of the purpose of the element but you'd have to refer to the CSS file to find out what that's actually comprised of. Having to look up all this info takes time whereas the first, admittedly less attractive markup with many classes, at least shows all the various bits and pieces that make it up at surface level.

The second major drawback of extend comes with complex projects with many extends or even nested extends. It would take a very long and complex example to illustrate this point but the general gist is that `@extend` can easily be misused and if extending complex selectors can actually lead to very bloated code and very long compile times.

Thirdly, when your CSS is minified and compressed with gzip any duplicated properties generated by `@mixin` will be compressed to the point that the benefits of `@extend` are actually negligible in many cases.

A final downside is seen when debugging and inspecting elements in the developer tools. Seeing a huge list of comma-separated selectors can be tricky to navigate and takes up a lot of space in the inspector, making things harder to debug efficiently.

This is not to say that you should never use `@extend`, just be aware that it's not a magic bullet and there can be side effects. I thought it was amazing when I first discovered it and that ended up biting me in a large-scale project and I've learned my lesson.

So, if you do see the benefit of extend and want to use it most effectively, here are some tips from Sass Gudielines.

- Use extend on %placeholders primarily, not on actual selectors.
- When extending classes, only extend a class with another class, never a complex selector.
- Directly extend a %placeholder as few times as possible.
- Avoid extending general ancestor selectors (e.g. .foo .bar) or general sibling selectors (e.g. .foo ~ .bar). This is what causes selector explosion.

This is some good advice and there's lots more of the same in the rest of the Sass Guidelines. You may find them a little opinionated, but there's definitely a lot of wisdom in them too.