Mixins are a Sass feature that enables blocks of code to be re-used with slight variations. They are really powerful and are a core tool in writing modular and maintainable code.

We'll go over a whole heap of handy mixins to help you out in all sorts of aspects of crafting styles for a project.

Anatomy of a Mixin

I explained the fundamentals of Sass mixins in my book Up and Running with Sass but by way of a refresher, there are a number of bits and pieces that make up a mixin.

First, we declare a mixin with the `@mixin` directive followed by a name for the mixin.

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

The mixin name is optionally followed by a set of parentheses where we define any input parameters. These are then made available as variables within the mixin to enable the output CSS to change based on the input values when the mixin is included.

```
@mixin uppercase-letter-spacing( $letter-spacing, $font-size ) {
    font-size: $font-size;
    letter-spacing: $letter-spacing;
    text-transform: uppercase;
}
h1 {
    @include uppercase-letter-spacing( 4px, 2rem );
}
```

When the mixin is used, the styles within the body of the mixin are output where the mixin is called with `@include`.

Some mixins may output other selectors instead of outputting properties and values. To do this, we use the `@content` directive within the body of the mixin.

```
@mixin hover-focus-active {
    &:hover,
    &:focus,
    &:active {
        @content;
    }
}
```

When this type of mixin in included, we use curly braces to define where the content of the mixin starts and ends.

```
a {
    color: #f00;

    @include hover-focus-active {
        color: #fff;
        background: #000;
    }
}
```

This would then compile into

```
a {
    color: #f00;
}
a:hover,
a:focus,
a:active {
    color: #fff;
    background: #000;
}
```

Mixins are very useful and can be a great time saver. But when you're first learning Sass, it can be hard to see what you might use them for or what type of problems they can help you solve. To help with that, here are a handful of helpful mixins and examples of how you might use them.

Selection Text

When styling selection text, we need to provide two separate blocks of CSS to handle vendor prefixes in the selector. Creating a mixin allows us to set the selection text colours once and have them replicated as necessary.

```
@mixin selection( $background, $foreground:#fff ) {
    ::-moz-selection {
        background: $background;
        color: $foreground;
```

```
        text-shadow: none;
    }
    ::selection {
        background: $background;
        color: $foreground;
        text-shadow: none;
    }
}
@include selection( #000, #ccc );
```

Form Input Placeholders

Form input placeholder selectors are similar to selection text in that the vendor prefixes in the selector mean each browser needs to be targeted separately.

Here the placeholder styles can be defined once and passed to `@content` in each of the four input placeholder selectors.

```
@mixin form-placeholder {
    ::-webkit-input-placeholder {
        @content;
    }
    ::-moz-placeholder {
        @content;
    }
    :-moz-placeholder {
        @content;
    }
    :-ms-input-placeholder {
        @content;
    }
}
@include form-placeholder {
    color:#ccc;
    font-style:italic;
}
```

Centring Things

Centring elements is a common practice and can be accomplished with fairly minimal CSS. But it's such a common practice that it's often simpler to write a single line instead of three or four.

```
// Center Block
//
```

```
@mixin center-block {
    display: block;
    margin-left: auto;
    margin-right: auto;
}

// Easy horizontal & vertical centering of anything in IE9+
//
@mixin absolute-center {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate( -50%, -50% );
}

// Easy vertical centering of anything in IE9+
//
@mixin vertical-center {
    position: absolute;
    top: 50%;
    transform: translateY( -50% );
}
```

Creating mixins for common styles like this could be considered an abstraction too far and perhaps makes the code less meaningful. Use your own judgement but just be consistent in which method you choose.

Hiding Things

There are a number of approaches for hiding elements whilst still allowing them to be visible to users of screen readers. This mixin taken from the HTML5 Boilerplate is a solid approach.

When hiding something, we want to really make sure it's hidden so I've included a `!important` after each declaration here to ensure the element is definitely not visible to sighted users. This is really the only time such liberal use of `!important` makes sense - try to avoid it for getting out of specificity trouble.

```
@mixin visuallyhidden {
    border: 0 !important;
    clip: rect(0 0 0 0) !important;
    height: 1px !important;
    margin: -1px !important;
    overflow: hidden !important;
    padding: 0 !important;
```

```
    position: absolute !important;
    width: 1px !important;
}
```

I keep all my mixins in their own `_mixins.scss` partial in each of my projects to keep them all together and provide a logical place for any new ones to be added.

If you like the sound of having a library of mixins but aren't too keen on writing them yourself, there are a few Sass tools out there that provide just that.

We'll be covering the library Bourbon & Neat in a future video but you could also check out Compass or Scut which both provide a whole heap of mixins for all sorts of day to day tasks.