When crafting complex functions, mixins or perhaps even your own framework or Sass plugin it can be useful to display messages on the console to ensure your code is working correctly and being used correctly.

Sass has three different methods of interacting with the console which we'll cover in this episode. They are:

- @warn
- @error
- @debug

@warn

The `@warn` directive prints a warning message to the console.

```
@warn 'this is a test warning message!';
```

If you add this code to any Sass file, the warning will be printed to the console on every save of the file.

```
WARNING: this is a test warning message!
     on line 3 of assets/scss/_functions.scss
     from line 1 of assets/scss/style.scss
```

While this is a simple way to demonstrate what the `@warn` directive is, it's not a particularly useful message.

Instead of having the warning display always, we can use it within a Sass function or mixin to alert the developer under certain circumstances. For example, we may want to warn about improper use of arguments or add deprecation warnings about use of an old function.

In this example, I've got a function we created in a previous screencast that converts pixels to rems. Let's imagine that this function is part of a Sass library or framework that's used on multiple projects by multiple developers - like the Bootstrap or Foundation framework for example.

As the developers of the framework we want to rename a function but don't want to create a breaking change in the code. The `px-to-rem` function is a bit long and instead could be renamed to just `rem`.

```
@function strip-units( $length ) {
    @return $length / ( ( $length * 0 ) + 1 );
}
@function px-to-rem( $value ) {
    $value: strip-units( $value );
    $base-font-size: strip-units( $base-font-size );

    @return ( $value / $base-font-size ) * 1rem;
}
h1 {
    font-size: px-to-rem( 64px );
}
```

If we just rename the function then any instances in the code where the function is used will now throw errors.

```
@function rem( $value ) {
    $value: strip-units( $value );
    $base-font-size: strip-units( $base-font-size );

    @return ( $value / $base-font-size ) * 1rem;
}
h1 {
    font-size: px-to-rem( 64px ); // throws undefined function error
}
```

To ensure our new shortened function name doesn't create a breaking change we can create a function with the same name as the old one that calls the new one.

```
@function px-to-rem( $value ) {
    @return rem( $value );
}
```

And to let the developer know what's going on behind the scenes - so they can be aware of what's changed - we can then add a warning message.

```
@function px-to-rem( $value ) {
    @warn 'The `px-to-rem()` function has been deprecated and will be
    removed in a future version. Use `rem()` instead.'

    @return rem( $value );
}
```

Now when we use the old function there are no errors thrown but the developer is alerted to the fact that there's a new, shorter named function they can use instead.

@error

In a similar fashion to the `@warn` directive, the `@error` directive can be used to display a custom error message in the console.

```
@error 'This is a test error message!'
```

One difference between a warning message and an error message is how they display on the console.

```
error assets/scss/_functions.scss (Line 3: this is a test error message!)
```

But the key difference is that the `@error` directive will stop the compiler at the point of error whereas when a `@warn` directive is reached, the compiler keeps going.

As such, the `@error` directive should be used to alert the developer that something fatal has happened which needs to be corrected before proceeding. A classic example of this is when using incompatible values in a function or mixin. Let's take a look at an example.

Using our `rem` function from the previous example, we might want to throw an error if units other than pixels are passed to our function for conversion.

To handle this, we can add a conditional statement before the main body of the function to catch values that are not unitless or in pixel units.

```
@function rem( $value ) {

    @if unitless( $value ) or unit( $value ) == 'px' {
        $value: strip-units( $value );
        $base-font-size: strip-units( $base-font-size );

        @return ( $value / $base-font-size ) * 1rem;
    } @else {
        @error 'Only unitless or px values should be passed to `rem()`';
    }
}
```

If we try to call the `rem()` function with an `em` value, our error is thrown.

```
h1 {
    font-size: rem( 3em ); // throws an error
}
```

To make our error message even more helpful, we can output the value within the message using Sass string interpolation:

```
@else {
    @error 'Invalid value of #{ $value } passed to `rem()`. Only unitless or px values should be u
}
```

Much better.

@debug

Finally, the `@debug` directive can be used when you're in the process of developing your own functions and mixins. This is a bit like the `console.log` function used when developing and debugging code in JavaScript.

Like the `@warn` directive, `@debug` doesn't stop the compiler running and just prints a messgage to the console. As we saw above, you can use string interpolation to print the value of variables to the console which is incredible useful.

Here we are printing out the value passed to the function before we strip its units and then again after to ensure we see the expected behaviour.

```
@function rem( $value ) {

    @if unitless( $value ) or unit( $value ) == 'px' {

        @debug 'Initial value: #{ $value }';
        $value: strip-units( $value );
        @debug 'Stripped units value: #{ $value }';

        $base-font-size: strip-units( $base-font-size );
        @return ( $value / $base-font-size ) * 1rem;

    } @else {
        @error 'Invalid value of #{ $value } passed to `rem()`. Only unitless or px values should
    }
}
```

Instead of writing out a debug message it's also possible to debug just a variable or the result of an expression:

```
@debug $value;
@debug ( $value / $base-font-size ) * 1rem;
```

Each of these directives are useful in their own right but when combined together like this, allow you to create very robust code that clearly communicated back if it's being used incorrectly or sub-optimally.

`@warn` and `@error` are particularly useful if you're developing a framework or your own boilerplate code that might be used by a number of people but `@debug` will be useful wether you're a solo developer or work as part of a large team.