Start applying your kotlin knowledge

In our starter code from Android Studio, we have these lines:

```
setContent {
    MyLabsTheme {
        // A surface container using the 'background' color from the theme
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            Greeting( name: "Android")
        }
    }
}
```

The line setContent{ } means that the **setContent** function takes lambda function as parameter, so you could rewrite it as:

```
setContent( {  code...  })
```

Inside the setContent function, there is something called MyLabsTheme{ }, which is also a function which takes a lambda function.

Notice inside the MyLabsTheme, there is a line of code:

```
Surface(modifier=Modifier.fillMaxSize(),
        color=MaterialTheme.color.background)
{
  Greeting("Android")
}
```

This means that the Surface function has some default parameters, but we are only overriding the **modifier** and **color** parameters. Also, the last parameter is a lambda function, which is being supplied outside of the ( ) parentheses. Have a look at the declaration of the Surface function.

```
@Composable
fun Surface(
    modifier: Modifier = Modifier,
    shape: Shape = RectangleShape,
    color: Color = MaterialTheme.colorScheme.surface,
    contentColor: Color = contentColorFor(color),
    tonalElevation: Dp = 0.dp,
    shadowElevation: Dp = 0.dp,
    border: BorderStroke? = null,
    content: @Composable () -> Unit
) {
```

Since modifier is the first parameter in the list, you don't need to write the **modifier=** parameter specification. You could also bring the lambda function inside the parentheses, and declare that as the value for the content parameter:

```
Surface( Modifier.fillMaxSize(),
         color = MaterialTheme.colorScheme.background,
         content = { Greeting("Android")})
```

However, it's best to use named parameters to make clear what parameter you are passing, and to leave lambda functions outside the parentheses:

```
Surface(
      modifier = Modifier.fillMaxSize(),
      color = MaterialTheme.colorScheme.background
) {
      Greeting("Android")
}
```

The Greeting function is declared in your code a little lower down:

```
@Composable
fun Greeting(name: String) {
```

```
        Text(text = "Hello $name!")
    }
```

It calls a constructor for a Text object, and passes the text="Hello $name" value. Have a look at the Text constructor:

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
) {
```

Since the text parameter is the first in the list, you can rewrite the code as:

```
    Text("Hello $name!")
```

However, it is good form to leave the parameter name there as a reminder. The

@Composable annotation means that this function will return something that you can show on screen. This is the new technology used in Jetpack compose, is that @Composable replaces the View object in the old way of programming Android. We'll use that next week when we practice putting objects on screen.

Lastly, notice that there is another function with the **@Preview** annotation. The @Preview means that this function will return a Composable object that represents a preview of what this will look like on a phone.