from **View** to **Composable**

Learn Jetpack Compose from an Android View mindset

Alex Styl

Contents

Preface

Thank you for reading my book! I am Alex Styl and I have been developing for Android for 10+ years.

You do not have to relearn everything you know from Android Views to use Jetpack Compose!

This is a no fluff, concise book that answers any specific question you might have while developing with Compose. Say for example that you are working with text and you need to style it. Jump straight to the Text chapter and find the answer to your styling questions there. Or you might not know how to structure your Jetpack Compose application or how Navigation works. Everything is here!

There are parts that might not give much information about a specific aspect of Compose and that is intentional. In the Text chapter, there is little emphasis to how you would use theming to style your entire app's Text in a single way, as the focus there in on how to use the Text composable. Instead you can read all about it in the Theming chapter.

If you are new in Compose or you struggle with understanding reading Compose code, I strongly encourage you to read the first chapter.

Special thanks to our proof-readers for their incredible work reviewing chapters of this book. More specifically I would like to thank: Colton Idle, Daniele Bonaldo, Davide Agostini, Ian Lake, Maia Grotepass, Manuel Vicente Vivo, Michael Sim, Rafael R. Tonholo and Jose Alcérreca.

Enjoy and happy coding,

– Alex Styl

PS: This book is also accessible via your browser. Go to viewtocomposable.com to read.

Understanding Compose code

In Jetpack Compose, each component that needs to be rendered on the screen can be defined as a Kotlin Unit function marked with the `@Composable` annotation like this:

```
@Composable
fun Article(title: String, description: String) {
    Card {
        Column {
            Text(title)
            Spacer(Modifier.height(10.dp))
            Text(description)
        }
    }
}
```

We call those functions *composables*. The above composable will render a Card with a title and a subtitle, with a spacing of 10 dp in between.

Every time the title and description change, the UI will be updated to reflect the updated values. This is what we call *recomposition*.

You may only call composable functions from other composable functions. Because of this, activities that use Composable to render their layouts will look like this:

```
class MyActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            // use your composables here
        }
    }
}
```

`setContent {}` is an extension function of the `ComponentActivity`[1]. Using composables in a Fragment needs a `ComposeView` like so:

[1] ComponentActivity is part of the androidx.activity:activity-compose dependency.

```
class MyFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater?,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return ComposeView(requireContext()).apply {
            setViewCompositionStrategy(DisposeOnViewTreeLifecycleDestroyed)
            setContent {
                // use your composables here
            }
        }
    }
}
```

In the View world, there are some common attributes and features found in most Views out there. Things like setting click and touch listeners, applying elevation, alpha, to name a few. For anyone creating their own custom views, there was a lot of boilerplate code to implement for your view to support such operations.

Compose introduces the concept of `Modifiers`. Modifiers provide functionality and features to composables without being tied to specific composables. Some Modifiers can be used for styling the composable (see `background()`, `border()` , `clip()` , `shadow()`, `alpha()`, `animateContentSize()`), others help with the placement and sizing of the composable (see `fillMaxWidth()` , `size()`, `heightIn()`, `padding()`) and others can bring functionality to the composable such as enabling click behavior or dragging (see `clickable()` ,`draggable()`, `toggleable()`, `swipeable()`).

It is considered a good practice to always provide a `Modifier` when creating your own composable. This will allow callers of your composable to provide custom styling and specify any layout requirements they might have independently of the composable's code.

That was the basic knowledge you need, in order to understand Compose code you see in the wild. There are a few concepts that are new to Compose. Those new concepts will be explained in later parts of this book.

TextView to Text

`Text` is the direct replacement of `TextView`. It includes all standard `TextView` customizations and many more. Customizations can be passed directly as parameters to the composable. You can extract all customizations into a `TextStyle` that you can use through your app. You can provide a `TextStyle` directly into a `Text` as a parameter or via theming.

Example of usage:

```
Text("hello from Compose")
```

How to customize your text (text color, size, max lines, etc)

The `Text` composable supports functionality you would expect such as changing its `textSize`, setting its `maxLines` or its `color`.

In addition to standard `TextView` functionality, you get the option to modify the text's `letterSpacing`, `lineHeight` and more. All customization options can be found by inspecting the parameters of the `Text` composable.

```
Text(
    "Hello from Compose",
    fontSize = 18.sp,
    maxLines = 1,
    color = Color.Blue,
)
```

How to reuse your text styles

One of the most important parameters of `Text` composable is the `style` parameter. It allows you to pass a `TextStyle`. This works in a very similar way to Views's `TextAppearance`.

A `TextStyle` contains all customizations you can apply as parameters to a `Text` composable. The benefit is that you can reuse the created style in multiple Text composables in your app. This makes styling consistent across the app and makes your composables easier to read and maintain.

A typical `TextStyle` looks like this:

```
val h1 = TextStyle(
    fontFamily = FontFamily.Default,
    fontWeight = FontWeight.SemiBold,
    fontSize = 42.sp
)
```

which you can use in your `Text` like this:

```
Text("My big header", style = h1)
```

Alternatively, you can define the style in your application's theme like this:

```
val Typography = Typography(
    h1 = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.SemiBold,
        fontSize = 42.sp
    )
)

@Composable
fun MyTheme(content: @Composable () -> Unit) {
    MaterialTheme(
        typography = Typography,
        content = content
    )
}
```

and use it in your composable:

```
MyTheme {
    Text("My big header", style = MaterialTheme.Typography.h1)
}
```

How to use custom Fonts

One of the parameters of `Text` is `fontFamily`. Compose provides some built-in fonts you can use that are device dependant, using the `FontFamily` object.

If the passed font family is not available on the device, a default font will be used instead. Alternatively, you can also define your own custom fonts in order to personalize your app further.

Example:

```
Text("This text is written in cursive", fontFamily = FontFamily.Cursive)
```

The font family can be used directly as a parameter to your `Text` composable or via a `TextStyle`. The `TextStyle` can be used in your application's theme.

There is a big chance that you will need to change the font of the entire app. For this purpose, you can use the `defaultFontFamily` parameter of the `Typography` class, which is part of the `MaterialTheme`.

How to create custom Fonts

First, store your *.tff* font files into your project under your `res/font` folder.

> You can find a plethora of free to use fonts at Google Fonts.

Each font family might come with multiple files. Each file represents a different weight of the font, such as bold, semi-bold or thin. You will have to copy all those files in your resource folder.

Then, create a new `FontFamily` object and link each file to a respective styling:

```
val Urbanist = FontFamily(
    Font(R.font.urbanist_thin, weight = FontWeight.Thin),
    Font(R.font.urbanist_light, weight = FontWeight.Light),
    Font(R.font.urbanist_regular, weight = FontWeight.Normal),
    Font(R.font.urbanist_medium, weight = FontWeight.Medium),
    Font(R.font.urbanist_bold, weight = FontWeight.Bold),
    Font(R.font.urbanist_italic, style = FontStyle.Italic),
)
```

How to use strings from xml resources

You can continue using all your translated XML string resources in your composables. Instead of having to rely to a `Resource` or `Context` object, you can use the `stringResource()` function:

```
Text(stringResource(R.string.my_translated_string))

// stringResource accepts args too
Text(stringResource(R.string.hello_to), "Alex")
```

How to use string plurals from xml resources

There is currently no built-in way to use plurals. You would have to create your own function for the job.

You can use a mechanism called *Composition Locals* to get the current `Context` from a current composable function. This is what `stringResource()` and other similar resource functions use internally to get a hold of `Context` when needed.

A function for string plurals might look like this:

```
    @Composable
    fun quantityStringResource(
        @PluralsRes pluralResId: Int,
        quantity: Int,
        vararg formatArgs: Any? = emptyArray()
    ): String {
        return LocalContext.current.resources
            .getQuantityString(pluralResId, quantity, *formatArgs)
    }
```

which you can use in your app like this:

```
// "Contact" or "Contacts" depending on the numberOfContacts
Text(quantityStringResource(R.plurals.contact), numberOfContacts)

// "No contact", "1 contact", or "$number contacts"
Text(
    quantityStringResource(R.plurals.x_number_of_contacts),
    numberOfContacts,
    numberOfContacts
)
```

How to make Text clickable

Instead of using a `Modifier.clickable {}` to make your text clickable, it is recommended to wrap your `Text` into an `TextButton` composable instead. The `TextButton` will style your Text according to your theme, and will include the right minimum touch target size required.

```
TextButton(onClick = { /*TODO start search*/ }) {
    Text("Search")
}
```

How to make part of Text clickable (UrlSpans)

Create an `AnnotatedString` with the styling and URL information you need:

```
val tag = "info"
val annotatedString = buildAnnotatedString {
    val text = "For more info click here"
    append(text)

    val start = text.indexOf("here")
    val end = start + 4

    addStyle(
        style = SpanStyle(
            color = MaterialTheme.colors.primary,
            textDecoration = TextDecoration.Underline
        ),
        start = start,
        end = end
    )

    addStringAnnotation(
        tag = tag,
        annotation = "https://viewtocomposable.com",
        start = start,
        end = end
    )
}
```

then pass the annotated String into a `ClickableText` and use the `LocalUriHandler` to launch the respective URL:

```
val uriHandler = LocalUriHandler.current
ClickableText(
    text = annotatedString,
    onClick = { offset ->
        annotatedString
            .getStringAnnotations(tag, offset, offset)
            .firstOrNull()
            ?.let { string ->
                uriHandler.openUri(string.item)
            }
    }
)
```

How to align my Text according to its baseline

Use `Modifier.paddingFromBaseline()` to place paddings to your composable according to your Text's first or last line's baseline.

```
Text(
    "Choose an account",
    modifier = Modifier.paddingFromBaseline(top = 40.dp),
    style = MaterialTheme.typography.h6
)
```

About the Author

I am Alex Styl (short for Alexandros Stylianidis) and I have been developing for Android since version 2.3 (Gingerbread). I have worked on projects for international companies such as Apple's Shazam and Channel 4's All-4. I have also used Android to conduct research during my Human-Computer Interaction studies.

Other than building for Android, I was one of the founding members of the Greek Android developers Slack community, which later turned into the GDG Android Athens and monthly meetups.

Since 2021, I quit my full time job as a Software Engineer to focus on my own ventures. This gives me enough time and flexibility to focus on creating Jetpack Compose content such as this book.