Android Jetpack Compose Notes

# Compose Essentials

Jetpack Compose is Declarative UI Framework. You declare what you UI should contain and compose creates the elements UI using Kotlin.

You construct the UI by describing what and not how.

You don’t need to use XML Views, the UI will be described in code using Kotlin.

In Compose, UI elements are functions, known as composables, and not objects. That means you can’t find references to them and mutate them. Instead, UI elements are controlled by the state or arguments you pass.

We don’t tell Compose how it should render states. To do that, we use Event Handlers, which decides if the UI element’s state should be changed. If the UI state changes, the functions or UI elements that depend on that state is re-executed, this is called recomposition.

## Composable Functions

A composable function is a function that has a @Composable annotation. This annotation indicates the compiler that this function is intended to convert data into UI.

The composable functions allow you have reusable components.

Composables are immutable, that means you can’t hold a reference to them and later update its value.

Recomposition happens when a composable is re-invoked with different functions parameters but it can also happen when internal state in the function changes.

MutableState, remember and rememberSaveable can be used to store a component state and have Compose automatically track and recompose changes.

Composable functions can execute in any order.

Composable functions can also run in parallel.

Recomposition skips as much as possible.

Composable functions might run frequently.

With Compose, an Activity remains the entry point to an Android app. In our project, MainActivity is launched when the user opens the app (as it's specified in the AndroidManifest.xml file). You use setContent to define your layout, but instead of using an XML file as you'd do in the traditional View system, you call Composable functions within it.

To use the Android Studio preview, you just have to mark any parameterless Composable function or functions with default parameters with the @Preview annotation and build your project. Example:

@Preview(showBackground = true, name = "Text preview")  
@Composable  
fun GreetingPreview() {  
    BasicsCodelabTheme {  
        Greeting(name = "Android")  
    }  
}

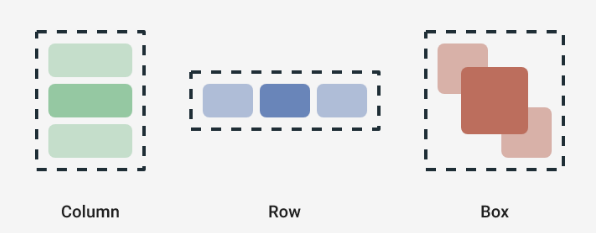
Surface and MaterialTheme are concepts related to [Material Design](https://m3.material.io/), which is a design system created by Google to help you create user interfaces and experiences.

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(color = MaterialTheme.colorScheme.primary) {  
        Text(  
            text = "Hello $name!",  
            modifier = modifier  
        )  
    }  
}

As a best practice, your function should include a Modifier parameter that is assigned an empty Modifier by default.

## Columns and Rows

The 3 basic standard layout elements in Compose are Column, Row and Box



Examples:

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Column(modifier = Modifier.fillMaxWidth().padding(24.dp)) {  
            Text(text = "Hello ")  
            Text(text = name)  
        }  
    }  
}

**Examples for a Row**

In this code, you are using a row and placing an ElevatedButton at the end of the row by using a weight to the first element (the Column). The weight modifier makes the element fill all available space, making it *flexible*, effectively pushing away the other elements that don't have a weight, which are called *inflexible*. It also makes the fillMaxWidth modifier redundant.

import androidx.compose.foundation.layout.Row  
import androidx.compose.material3.ElevatedButton  
// ...  
  
@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Row(modifier = Modifier.padding(24.dp)) {  
            Column(modifier = Modifier.weight(1f)) {  
                Text(text = "Hello ")  
                Text(text = name)  
            }  
            ElevatedButton(  
                onClick = { /\* TODO \*/ }  
            ) {  
                Text("Show more")  
            }  
        }  
    }  
}

### Recompose

Remember that the use of **remember** is to guard a state after recomposition.

You can think of internal state as a private variable in a class.

The composable function will automatically be "subscribed" to the state. If the state changes, composables that read these fields will be recomposed to display the updates

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    val expanded = remember { mutableStateOf(false) }  
    val extraPadding = if (expanded.value) 48.dp else 0.dp  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Row(modifier = Modifier.padding(24.dp)) {  
            Column(  
                modifier = Modifier  
                    .weight(1f)  
                    .padding(bottom = extraPadding)  
            ) {  
                Text(text = "Hello ")  
                Text(text = name)  
            }  
            ElevatedButton(  
                onClick = { expanded.value = !expanded.value }  
            ) {  
                Text(if (expanded.value) "Show less" else "Show more")  
            }  
        }  
    }  
}

### State Hoisting

In Composable functions, state that is read or modified by multiple functions should live in a common ancestor—this process is called **state hoisting**. To *hoist* means to *lift* or *elevate*.

Making state hoistable avoids duplicating state and introducing bugs, helps reuse composables, and makes composables substantially easier to test.

In Compose **you don't hide UI elements**. Instead, you simply don't add them to the composition, so they're not added to the UI tree that Compose generates. You do this with simple conditional Kotlin logic.

Example:

@Composable  
fun MyApp(modifier: Modifier = Modifier) {  
  
    var shouldShowOnboarding by remember { mutableStateOf(true) }  
  
    Surface(modifier) {  
        if (shouldShowOnboarding) {  
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })  
        } else {  
            Greetings()  
        }  
    }  
}  
  
@Composable  
fun OnboardingScreen(  
    onContinueClicked: () -> Unit,  
    modifier: Modifier = Modifier  
) {  
  
  
    Column(  
        modifier = modifier.fillMaxSize(),  
        verticalArrangement = Arrangement.Center,  
        horizontalAlignment = Alignment.CenterHorizontally  
    ) {  
        Text("Welcome to the Basics Codelab!")  
        Button(  
            modifier = Modifier  
                .padding(vertical = 24.dp),  
            onClick = onContinueClicked  
        ) {  
            Text("Continue")  
        }  
    }  
  
}

In the MyApp composable, we define the shouldShowOnboarding variable as a mutableStateOf and using the by remember so this state can be changed later. Depending of the value of this variable, we show the composable OnboardingScreen or the Greetings.

We share the shouldShowOnboarding with the OnboardingScreen composable, but instead of passing its value directly, we use a callback. Callbacks are functions that are passed as arguments to other functions and get executed when the event occur.

By passing a function and not a state to OnboardingScreen we are making this composable more reusable and protecting the state from being mutated by other composables. In general, it keeps things simple.

### Lazy Lists

When you define a list like this one:

names: List<String> = List(1000) { "$it" }

The it represents the index.

If for every item you create a Composable, like the Greetings to show the value, you can use a LazyColumn instead of a Column to redners only the visible items on screen, allowing performance gains when rendering a big list.

**Note**: LazyColumn and LazyRow are equivalent to RecyclerView in Android Views.

import androidx.compose.foundation.lazy.LazyColumn  
import androidx.compose.foundation.lazy.items  
// ...  
  
@Composable  
private fun Greetings(  
    modifier: Modifier = Modifier,  
    names: List<String> = List(1000) { "$it" }   
) {  
    LazyColumn(modifier = modifier.padding(vertical = 4.dp)) {  
        items(items = names) { name ->  
            Greeting(name = name)  
        }  
    }  
}

**Note:**LazyColumn doesn't recycle its children like RecyclerView. It emits new Composables as you scroll through it and is still performant, as emitting Composables is relatively cheap compared to instantiating Android Views.

### Persisting State

If your app allows screen rotation, you may notice your state is lost. The remember function works **only as long as the composable is kept in the Composition**. When you rotate, the whole activity is restarted so all state is lost. This also happens with any configuration change and on process death.

Instead of using remember you can use rememberSaveable. This will save each state surviving configuration changes (such as rotations) and process death.

    import androidx.compose.runtime.saveable.rememberSaveable  
    // ...  
  
    var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }

You can also use rememberSaveable to remember a state after configuration changes.

   var expanded by rememberSaveable { mutableStateOf(false) }

### Animations

In Compose, there are multiple ways to animate your UI: from high-level APIs for simple animations to low-level methods for full control and complex transitions. You can read about them in the [documentation](https://developer.android.com/jetpack/compose/animation?authuser=1).

One example of a low level animation is the animateDpAsState composable.

animateDpAsState takes an optional animationSpec parameter that lets you customize the animation.

import androidx.compose.animation.core.Spring  
import androidx.compose.animation.core.spring  
  
  
@Composable  
private fun Greeting(name: String, modifier: Modifier = Modifier) {  
  
    var expanded by rememberSaveable { mutableStateOf(false) }  
  
    val extraPadding by animateDpAsState(  
        if (expanded) 48.dp else 0.dp,  
        animationSpec = spring(  
            dampingRatio = Spring.DampingRatioMediumBouncy,  
            stiffness = Spring.StiffnessLow  
        )  
    )  
  
    Surface(  
    // ...  
            Column(modifier = Modifier  
                .weight(1f)  
                .padding(bottom = extraPadding.coerceAtLeast(0.dp))  
  
    // ...  
  
    )  
}

### Styling and Theming

In the ui/theme/Theme.kt file, you see the definition of the project’s Theme, which uses MaterialTheme, a composable function that reflects the styling principles from the [Material design specification](https://m3.material.io/). That styling information cascades down to the components that are inside its content, which may read the information to style themselves.

From any descendant composable you can retrieve three properties of MaterialTheme: colorScheme, typography and shapes.

Example:

            Column(modifier = Modifier  
                .weight(1f)  
                .padding(bottom = extraPadding.coerceAtLeast(0.dp))  
            ) {  
                Text(text = "Hello, ")  
                Text(text = name, style = MaterialTheme.typography.headlineMedium)  
            }

The Text composable in the example above sets a new TextStyle. You can create your own TextStyle, or you can retrieve a theme-defined style by using MaterialTheme.typography, which is preferred. This construct gives you access to the Material-defined text styles, such as displayLarge, headlineMedium, titleSmall, bodyLarge, labelMedium etc

Sometimes you need to deviate slightly from the selection of colors and font styles. In those situations it's better to base your color or style on an existing one.

For this, you can modify a predefined style by using the copy function. Make the number extra bold

import androidx.compose.ui.text.font.FontWeight  
// ...  
Text(  
    text = name,  
    style = MaterialTheme.typography.headlineMedium.copy(  
        fontWeight = FontWeight.ExtraBold  
    )  
)

This way if you need to change the font family or any other attribute of headlineMedium, you don't have to worry about the small deviations.

Example to show a preview with the Dark Mode active:

import android.content.res.Configuration.UI\_MODE\_NIGHT\_YES  
  
  
@Preview(  
    showBackground = true,  
    widthDp = 320,  
    uiMode = UI\_MODE\_NIGHT\_YES,  
    name = "GreetingPreviewDark"  
)  
@Preview(showBackground = true, widthDp = 320)  
@Composable  
fun GreetingPreview() {  
    BasicsCodelabTheme {  
        Greetings()  
    }  
}

You can find everything related to the current theme in the files inside the ui/theme folder. For example, the default colors that we have been using so far are defined in Color.kt.

In Theme.kt, define for example, the palette for dark colors:

private val DarkColorScheme = darkColorScheme(  
    surface = Blue,  
    onSurface = Navy,  
    primary = Navy,  
    onPrimary = Chartreuse  
)

These colors can be defined in the Color.kt file:

val Navy = Color(0xFF073042)  
val Blue = Color(0xFF4285F4)  
val LightBlue = Color(0xFFD7EFFE)  
val Chartreuse = Color(0xFFEFF7CF)

Example:

private val DarkColorScheme = darkColorScheme(  
    surface = Blue,  
    onSurface = Navy,  
    primary = Navy,  
    onPrimary = Chartreuse  
)  
  
private val LightColorScheme = lightColorScheme(  
    surface = Blue,  
    onSurface = Color.White,  
    primary = LightBlue,  
    onPrimary = Navy  
)  
  
@Composable  
fun BasicsCodelabTheme(  
    darkTheme: Boolean = isSystemInDarkTheme(),  
    // Dynamic color is available on Android 12+  
    dynamicColor: Boolean = true,  
    content: @Composable () -> Unit  
) {  
    val colorScheme = when {  
        dynamicColor && Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.S -> {  
            val context = LocalContext.current  
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)  
        }  
        darkTheme -> DarkColorScheme  
        else -> LightColorScheme  
    }  
    val view = LocalView.current  
    if (!view.isInEditMode) {  
        SideEffect {  
            (view.context as Activity).window.statusBarColor = colorScheme.primary.toArgb()  
            ViewCompat.getWindowInsetsController(view)?.isAppearanceLightStatusBars = darkTheme  
        }  
    }  
  
    MaterialTheme(  
        colorScheme = colorScheme,  
        typography = Typography,  
        content = content  
    )  
}

## The Compose UI toolkit

Jetpack Compose comes with a lot of UI elements out of the box.

Material Design is an adaptable system of guidelines, components and tools that supports the best practices of user interface design. Jetpack compose supports Material Design 2 and 3

With Material Design your app can be themed to match your brand by providing custom colors, typography and shapes.

You can define a Theme composable with all the customizations you want and this Theme function should be the outermost function we invoke. It is the first composable we invoke in the setContent

Scaffold is a fundamental material design component. It is a basic layout for arranging material components in common patterns, such as the screen with a small top app bar and a floating action button.

Another component is the Surface, which is where the content sits on.

There are 3 types of layouts as seen before: Rows (to order elements horizontally), Columns (to arrange elements vertically) and Boxes to show elements on top of anothers.

### Lazy Rows

A LazyRow allows you to implement a scrollable row composable. It only renders elements that are shown on the screen instead of all elements at the same time, which helps keep your app performance.

For the spacings between the elements we can make use of arrangements. There are several options to arrange the elements in a Row and in a column. In addition to the these arrangements, you can also use the Arrangement.spacedBy() to add a fixed space between each child composable.

import androidx.compose.foundation.layout.PaddingValues  
  
@Composable  
fun AlignYourBodyRow(  
   modifier: Modifier = Modifier  
) {  
   LazyRow(  
       horizontalArrangement = Arrangement.spacedBy(8.dp),  
       contentPadding = PaddingValues(horizontal = 16.dp),  
       modifier = modifier  
   ) {  
       items(alignYourBodyData) { item ->  
           AlignYourBodyElement(item.drawable, item.text)  
       }  
   }  
}

To maintain the same padding, but still scroll your content within the bounds of your parent list without clipping it, all lists provide a parameter to the LazyRow called contentPadding.

### Lazy Grids

The LazyHorizontalGrid provides a nice mapping from items to grid elements. In the following example you have a LazyHorizontalGrid with two fixed rows and some other measures like the height of the elements and the horizontal and vertical arrangement:

@Composable  
fun FavoriteCollectionsGrid(  
   modifier: Modifier = Modifier  
) {  
   LazyHorizontalGrid(  
       rows = GridCells.Fixed(2),  
       contentPadding = PaddingValues(horizontal = 16.dp),  
       horizontalArrangement = Arrangement.spacedBy(16.dp),  
       verticalArrangement = Arrangement.spacedBy(16.dp),  
       modifier = modifier.height(168.dp)  
   ) {  
       items(favoriteCollectionsData) { item ->  
           FavoriteCollectionCard(item.drawable, item.text, Modifier.height(80.dp))  
       }  
   }  
}

### Slot APIs

**Slot-based layouts** leave an empty space in the UI for the developer to fill as they wish. You can use them to create more flexible layouts.

In the following example:

@Composable  
fun HomeSection(  
   @StringRes title: Int,  
   modifier: Modifier = Modifier,  
   content: @Composable () -> Unit  
) {  
   Column(modifier) {  
       Text(stringResource(title))  
       content()  
   }  
}

You can use the content parameter for the composable's slot. This way, when you use the HomeSection composable, you can use a trailing lambda to fill the content slot. When a composable provides multiple slots to fill in, you can give them meaningful names that represent their function in the bigger composable container.

### Scrolling

You don’t always need a Lazy layout to automatically scroll a list of items. When a list has only a limited number of elements, you can instead choose to use a simple Column or Row and **add the scroll behavior manually**. To do so, you use the [verticalScroll](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "(androidx.compose.ui.Modifier).verticalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)" \t "_blank) or [horizontalScroll](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "(androidx.compose.ui.Modifier).horizontalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)" \t "_blank) modifiers. These require a [ScrollState](https://developer.android.com/reference/kotlin/androidx/compose/foundation/ScrollState?authuser=1" \t "_blank), which contains the current state of the scroll, used to modify the scroll state from outside. In this case, you're not looking to modify the scroll state, so you simply create a persistent ScrollState instance using [rememberScrollState](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "rememberScrollState(kotlin.Int)" \t "_blank).

Example:

import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.rememberScrollState  
import androidx.compose.foundation.verticalScroll  
  
@Composable  
fun HomeScreen(modifier: Modifier = Modifier) {  
   Column(  
       modifier  
           .verticalScroll(rememberScrollState())  
   ) {  
       Spacer(Modifier.height(16.dp))  
       SearchBar(Modifier.padding(horizontal = 16.dp))  
       HomeSection(title = R.string.align\_your\_body) {  
           AlignYourBodyRow()  
       }  
       HomeSection(title = R.string.favorite\_collections) {  
           FavoriteCollectionsGrid()  
       }  
       Spacer(Modifier.height(16.dp))  
   }  
}

### Scaffold

Material's [Scaffold](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1#Scaffold(androidx.compose.ui.Modifier,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Function0,androidx.compose.material3.FabPosition,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.foundation.layout.WindowInsets,kotlin.Function1)) composable. Scaffold gives you a **top-level configurable composable** for apps that implement Material design. It contains slots for various Material concepts, one of which is the bottom bar.

import androidx.compose.material3.Scaffold  
  
@Composable  
fun MySootheAppPortrait() {  
   MySootheTheme {  
       Scaffold(  
           bottomBar = { SootheBottomNavigation() }  
       ) { padding ->  
           HomeScreen(Modifier.padding(padding))  
       }  
   }  
}

### Navigation Rail

For the landscape mode, you can use a NavigationRail composable which is part of the Compose Material library and has a similar implementation to the [NavigationBar](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1" \l "NavigationBar(androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.Dp,androidx.compose.foundation.layout.WindowInsets,kotlin.Function1)" \t "_blank) that was used to create the bottom navigation bar.

import androidx.compose.foundation.layout.fillMaxHeight  
  
@Composable  
private fun SootheNavigationRail(modifier: Modifier = Modifier) {  
   NavigationRail(  
       modifier = modifier.padding(start = 8.dp, end = 8.dp),  
       containerColor = MaterialTheme.colorScheme.background,  
   ) {  
       Column(  
           modifier = modifier.fillMaxHeight(),  
           verticalArrangement = Arrangement.Center,  
           horizontalAlignment = Alignment.CenterHorizontally  
       ) {  
           NavigationRailItem(  
               icon = {  
                   Icon(  
                       imageVector = Icons.Default.Spa,  
                       contentDescription = null  
                   )  
               },  
               label = {  
                   Text(stringResource(R.string.bottom\_navigation\_home))  
               },  
               selected = true,  
               onClick = {}  
           )  
           Spacer(modifier = Modifier.height(8.dp))  
           NavigationRailItem(  
               icon = {  
                   Icon(  
                       imageVector = Icons.Default.AccountCircle,  
                       contentDescription = null  
                   )  
               },  
               label = {  
                   Text(stringResource(R.string.bottom\_navigation\_profile))  
               },  
               selected = false,  
               onClick = {}  
           )  
       }  
   }  
}

### Window Size

To show the landscape version when you turn the device, you need to use the calculateWindowSizeClass() function to see what configuration the phone is in.

There are three window size class widths: Compact, Medium and Expanded. When the app is in portrait mode it is Compact width, when it is in landscape mode it is Expanded width.

In the following example, we define which composable to use depending on the window size class:

import androidx.compose.material3.windowsizeclass.WindowSizeClass  
import androidx.compose.material3.windowsizeclass.WindowWidthSizeClass  
@Composable  
fun MySootheApp(windowSize: WindowSizeClass) {  
   when (windowSize.widthSizeClass) {  
       WindowWidthSizeClass.Compact -> {  
           MySootheAppPortrait()  
       }  
       WindowWidthSizeClass.Expanded -> {  
           MySootheAppLandscape()  
       }  
   }  
}

When we call this function, we use this in the setContent for the MainActivity:

import androidx.compose.material3.windowsizeclass.ExperimentalMaterial3WindowSizeClassApi  
import androidx.compose.material3.windowsizeclass.calculateWindowSizeClass  
  
class MainActivity : ComponentActivity() {  
   @OptIn(ExperimentalMaterial3WindowSizeClassApi::class)  
   override fun onCreate(savedInstanceState: Bundle?) {  
       super.onCreate(savedInstanceState)  
       setContent {  
           val windowSizeClass = calculateWindowSizeClass(this)  
           MySootheApp(windowSizeClass)  
       }  
   }  
}

## State in Jetpack Compose

State is any value that changes over time. In Android apps, state is updated in response to events.