Jetpack Compose Fundamentals

Functional Approach to Building UI

Topics

- UI Design Strategies
- Function Composition (in programming)
  - Tree of computational nodes
- UI Recomposition ⇒ *fancy term for UI refresh cycles*
  - A UI refresh implies invoking the functions again
- Maintaining UI States
  - Using local variables (with cache & why cache)
  - In ViewModel
- Prerequisites:
  - Proficient in reading & writing lambda syntax
  - [Coroutines basics]

Android Jetpack Compose

- Two main strategies for building user interface
  - *Imperative*: build the UI programmatically
  - *Declarative*: express the hierarchical relationship of UI widgets using a structured notation (typically XML or JSON)
- Jetpack Compose takes a *hybrid approach* by combining the two strategies
  - Based on the idea of mathematical function composition
  - Imperative: each (@Composable) function call is an action to build a “widget” (emit UI)
  - Declarative: the hierarchical relationships of the UI widgets is implied by the nesting structure of function call compositions

(Math) Function Composition

$$\boxed{f(g(x), h(\_))} \longrightarrow$$

```
fun main() {
  f {
     g {
     }
     h {
     }
  }
}
```

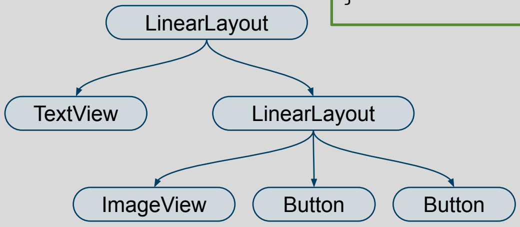
Kotlin Online Playground

Function Composition for Building UI?

Declarative XML vs. Function Composition

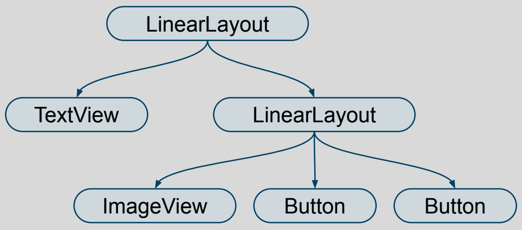
```
<LinearLayout orientation=”vertical”>
    <TextView .../>
    <LinearLayout orientation=”horizontal”>
        <ImageView .../>
        <Button .../>
        <Button .../>
    </LinearLayout>
</LinearLayout>
```

```
// Hypothetical example, not an actual
// Jetpack Compose code
fun MyUI() {
  LinearLayout(orientation=”vertical”) {
    Text (text = “Allendale”  ....)
    LinearLayout(orientation=”horizontal”) {
        ImageView {   }
        Button {...}
        Button {...}
    }
}
```



Tree Construction Cost?
(DFS traversal)

UI State, Composition & Recomposition



- **State**: any data needed to render the UI
- **Composition**: when the UI is built *<u>for the first time</u>*, the build function associated with each node must be evaluated.
  - These initial "output" can then be cached
- **Recomposition**: When the data for a particular node X changes, only the nodes *<u>on the path from the root to X</u>* must be reevaluated
- Any data needed by a node to render itself should be stored as its **(internal) state**

Performance considerations:
- *Each build function should be fast (it should not slow down the Main UI thread)*
- *Expensive operations in the build function should be dispatched as a coroutine running outside of the Main UI thread*

Jetpack Compose Key Concepts

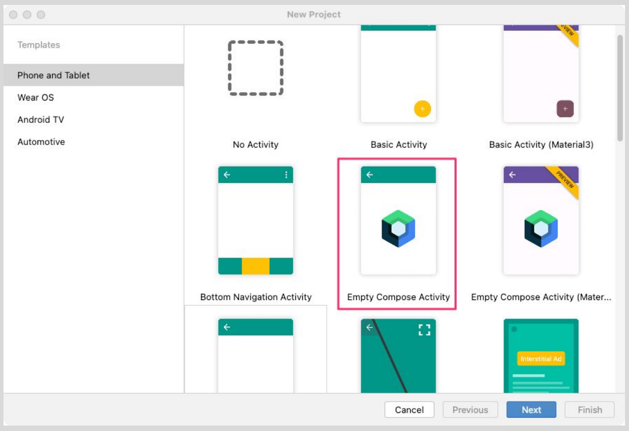
- `@Composable` Functions
  - A Kotlin functions that renders UI
  - Defined like any other functions: **may take arguments and has return value**
- UI States
  - Variables that contribute to the rendition of the UI contents
  - These variables can be either
    - parameter(s) of a @Composable
    - local state variables
- (Re)composition
  - Any updates to the UI states will initiate a UI refresh

UI State Updates ⇒ UI Recomposition

For those who know React:

This should remind you of `useState<T>(_)`?

Getting Started



Library Dependencies (Kotlin Script Syntax)

```
dependencies {
   implementation(“androidx.core:core-ktx:1.7.0”)
   implementation(“androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle_version”)
   implementation(“androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version”)
   implementation(“androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version”)
   implementation(“androidx.activity:activity-compose:1.3.1”)
   implementation("androidx.compose.ui:ui:$compose_ui_version")
   implementation("androidx.compose.ui:ui-tooling-preview:$compose_ui_version")
   implementation(“androidx.compose.material:material:1.2.0”)
   debugImplementation("androidx.compose.ui:ui-tooling:$compose_ui_version")
   debugImplementation "androidx.compose.ui:ui-test-manifest:$compose_ui_version")
}
```

Warning

- *It is challenging to know exactly which library to include, so many of them to choose from. Google should provide a better documentation!*
- *Mixing newer versions of libraries with older ones may fail your build with a “duplicate classes” error. Be wise in choosing the version numbers.*

Library Dependencies (Groovy Syntax)

```
dependencies {
   implementation 'androidx.core:core-ktx:1.7.0'
   implementation “androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle_version”
   implementation “androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version”
   implementation “androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version”
   implementation 'androidx.activity:activity-compose:1.3.1'
   implementation "androidx.compose.ui:ui:$compose_ui_version"
   implementation "androidx.compose.ui:ui-tooling-preview:$compose_ui_version"
   implementation 'androidx.compose.material:material:1.2.0'
   debugImplementation "androidx.compose.ui:ui-tooling:$compose_ui_version"
   debugImplementation "androidx.compose.ui:ui-test-manifest:$compose_ui_version"
}
```

Warning

- *It is challenging to know exactly which library to include, so many of them to choose from. Google should provide a better documentation!*
- *Mixing newer versions of libraries with older ones may fail your build with a “duplicate classes” error. Be wise in choosing the version numbers.*

Jetpack Compose Hello World

```
class MainActivity : ComponentActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContent {
     ComposedemoTheme {
       Surface(
               modifier = Modifier.fillMaxSize(),
               color = MaterialTheme.colors.background
       ) {
         Greeting(“World”)
       }
     }
   }
 }
}

@Composable
fun Greeting(name: String) {
   Text(text = "Hello $name!")
}
```

- *Parent class is ComponentActivity() as opposed to AppCompatActivity()*
- *setContentView(my_xml_layout) becomes setContent*
- *Surface is a “drawing canvas” to render the UI*
- *UI builder functions are annotated with @Composable*
- *Similar to ordinary functions, composable functions may take arguments of any type*
- *Composable functions can only be called inside another composable*
- *The @Composable functions can be declared in the same file XXXActivity.kt or in a separate file*

Android Studio Demo

Android Studio Demo

- Project Structure
- Layout Editor
- @Composable
- @Preview
- Online (interactive) documentations of Compose Components

Code Organization

Compose Best Practices

- App Logic and Data shall be placed in its own ViewModel
  - The subset of app data that affect the UI should be defined as `LiveData<T>`
- LiveData in the VM can be observed
  - as a UI state (using the `observeAsState()` extension)
  - manually using a typical `observe()` + lambda block
- Use local UI states with `remember {}` or `rememberSaveable {}` block and `mutableStateOf<T>`
  - Use local UI states sparingly

Failed Attempt for Managing UI States

```
@Composable
fun Counting() {
   var counter = 0
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $counter", fontSize = 24.sp)
       Button(onClick = { counter++ }) {
           Text("Click me")
       }
   }
}
```

❌

- *The text “Hello $counter” will not show updated counter value*
- *During recomposition the function Counting() is called again, and `counter` is reset to zero*

Failed Attempt for Managing UI States

```
@Composable
fun Counting() {
   var counter = MutableLiveData<Int>(41)
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $counter", fontSize = 24.sp)
       Button(onClick = { counter.value = counter.value + 1 }) {
           Text("Click me")
       }
   }
}
```

- *The text “Hello $counter” will not show updated counter value*
- *During recomposition the function Counting() is called again, and `counter` is reset to 41*

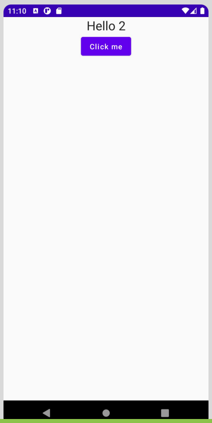
Cached State Variables

`remember { ... }`

`rememberSaveable { ... }`

Option 1: With Kotlin Delegation

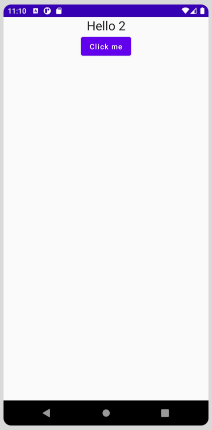
```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.saveable.remember

@Composable
fun Counting() {
   var counter by remember { // Will not survive screen rotation
       mutableStateOf<Int>(2)
   }
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $counter", fontSize = 24.sp)
       Button(onClick = { counter++ }) {
           Text("Click me")
       }
   }
}
```



- *`remember()` is a **caching** property delegate; it retains current value across multiple calls of recomposition*
- *remember() **does not** survive screen rotations*

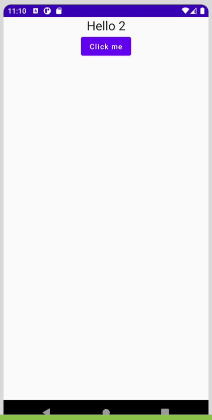
Option 2: Without Kotlin Delegation

```
import androidx.compose.runtime.saveable.remember

@Composable
fun Counting() {
   var counter = remember { // Will not survive screen rotation
       mutableStateOf<Int>(2)
   }
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello ${counter.value}", fontSize = 24.sp)
       Button(onClick = { counter.postValue (counter.value + 1) }) {
           Text("Click me")
       }
   }
}
```



by rememberSaveable { }

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.saveable.remember

@Composable
fun Counting() {
   var counter by rememberSaveable { // Survive screen rotation
       mutableStateOf<Int>(2)
   }
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $counter", fontSize = 24.sp)
       Button(onClick = { counter++ }) {
           Text("Click me")
       }
   }
}
```



Replace `remember()` with `rememberSaveable()` to survive screen rotations

Wiring Up With ViewModel

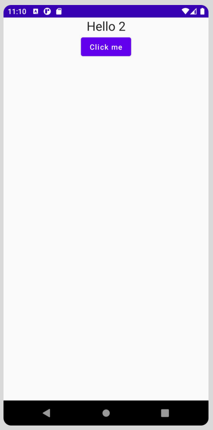
Define The ViewModel

```
class CountingViewModel: ViewModel() {
  private val _counter: MutableLiveData<Int> = MutableLiveData(0)

  val counter: LiveData<Int> get() = _counter;

  fun countUp () {
    _counter.postValue(_counter.value!! + 1)
  }
}
```

Similar ViewModel designed used earlier!!!

Use The ViewModel

```
@Composable
fun Counting(val vm = CountingViewModel()) {

   val countState by vm.counter.observeAsState();

   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $countState", fontSize = 24.sp)
       Button(onClick = { vm.countUp() }) {
           Text("Click me")
       }
   }
}
```

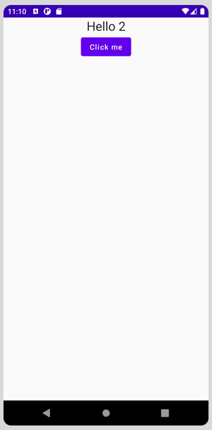


Lib dependency: `androidx.compose.runtime:runtime-livedata:1.6.7`

Observing LiveData

Manual Observer Block

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.saveable.remember

@Composable
fun Counting(val vm = CountingViewModel()) {
   val countState by vm.counter.observeAsState();
   vm.counter.observe(LocalLifecycleOwner.current) {
       // Your code here when vm.counter changes
       // (other than showing its value in the UI)
   }
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
       Text(text = "Hello $countState", fontSize = 24.sp)
       Button(onClick = { vm.countUp() }) {
           Text("Click me")
       }
   }
}
```

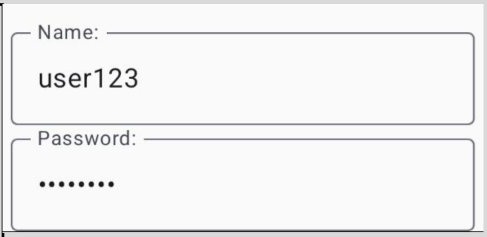


GitHub
android-jetpack-compose

Using Common Widgets

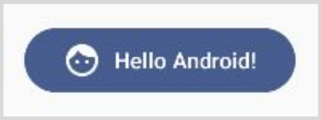
TextField

```
@Composable
fun SampleCode() {
    var loginName by remember { mutableStateOf(“”) }
    var password by remember {mutableStateOf(“EasyToGuess”)}

    OutlinedTextField(value = loginName,
        onValueChange = {
           loginName = it
        },
        label = {
          Text("Name:")
        })
    OutlinedTextField(value = password,
         onValueChange = { password = it },
         label = { Text("Password:") },
         visualTransformation = PasswordVisualTransformation()
   )
}
```


Name:
user123
Password:
••••••••

Button

```
@Composable
fun SampleCode(modifer: Modifier = Modifier) {
    Button(onClick={
              println (“....”)
           },
           modifier = modifier.padding(16.dp))
      {
        Icon(imageVector = Icons.Default.Face,
             contentDescription = "")
        Text(text = "Hello Android!",
             modifier = modifier.padding(start = 8.dp))
      }
}
```



Reading Assignment?

Jetpack Compose is not covered in Engelsma/Dulimarta textbook 😞

SECOND EDITION
Covers Xcode 12 / iOS 14 / Swift and Android Studio 4 / Android 11 / Kotlin
MASTERING
MOBILE APPLICATION
DEVELOPMENT
Learning iOS and Android Side-By-Side
14
11
Jonathan Engelsma and Hans Dulimarta