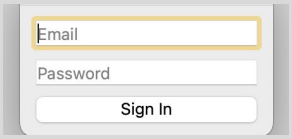
Jetpack Compose Fundamentals

Functional Approach to Building UI

Topics

- Approaches to Constructing UI
- Function Composition
  - Tree of (computed) UI nodes
- Jetpack Compose UI (Re)Composition
- Maintaining UI States
  - Using “special” local variables (with backing memory)
  - In ViewModel

UI Design Techniques




```
<div>  <!-- HTML: DECLARATIVE  -->
    <input type="text" placeholder="Email" >
    <input type="text" placeholder="Password">
    <button>Sign In</button>
</div>
```

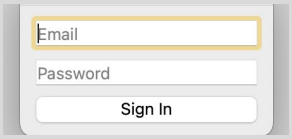
```
# Python - PyQt: IMPERATIVE
layout = QtWidgets.QVBoxLayout()
emailinp = QtWidgets.QLineEdit()
emailinp.setPlaceholderText("Email")
layout.addWidget(emailinp)

pwinp = QtWidgets.QLineEdit()
pwinp.setPlaceholderText("Password")
layout.addWidget(pwinp)

button = QtWidgets.QPushButton("Sign In")
layout.addWidget(button)
```

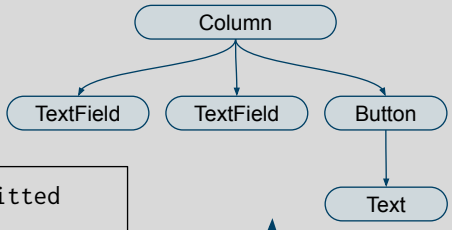
- **Declarative**: parent-child relationships among UI elements are expressed in a hierarchical structure
- **Imperative**: the UI is build programmatically

Android Jetpack Compose

- Jetpack Compose takes a *hybrid approach* by combining both the imperative and declarative
  - 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
    - All @Composable functions define a trailing lambda parameter of type @Composable
- Important Key Concepts
  - @Composable annotation (on function names)
  - (UI) State variables
  - (Initial) Composition & Recomposition
  - Side Effects (advanced)

UI Design in Jetpack Compose






```
// For brevity some required arguments are omitted
@Composable
fun LoginScreen() {
    Column {
        TextField(placeholder = “Email”)
        TextField(placeholder = “Password”)
        Button(onClick = { /* more code here */}) {
            Text(“Sign In”)
        }
    }
}
```

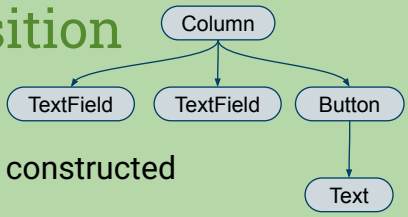
Function calls ⇒ Tree of UI Nodes

Nested Lambdas

```
@Composable
fun LoginScreen() {
    Column {
        TextField(placeholder = “Email”)
        TextField(placeholder = “Password”)
        Button(onClick = { /* more code here */}) {
            Text(“Sign In”)
        }
    }
}
```

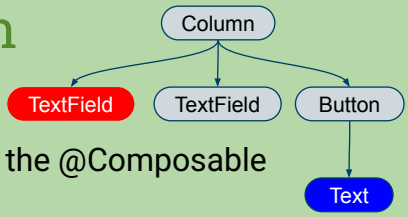
```
@Composable
fun LoginScreen() {
    Column(content = {
        TextField(placeholder = “Email”)
        TextField(placeholder = “Password”)
        Button(onClick = { /* more code here */},
               content = { Text(“Sign In”) })
    })
}
```

Tree of UI Nodes: Initial Composition



- Each node in the UI tree (TextField, Button, Text) is a constructed by invoking the associated @Composable function
  - When each function returns, all its local variables are destroyed
- The “cost” of **initial composition** ≈ full DFS traversal for invoking these functions
- *The function calls end but the UI tree persists!*
- In a real application, the tree can be significantly huge

Tree of UI Nodes: Recomposition



- UI update will trigger UI **recomposition,** i.e. invoking the @Composable function which creates the affected UI node(s)
- When a child node is recomposed, its parent must also be recomposed
  - When the red TextField changes, the Column must be recomposed
  - When the blue Text changes, the Button and Column must be recomposed
- To optimize performance, the composer logic **skip some nodes** which do not required updates. Hence **avoiding a full DFS (re)scan**

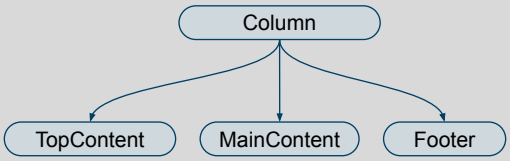
Key Concept #1: `@Composable` annotation

@Composable
UI state variable
(Re)composition

- A @Composable function is an ordinary function, so any rules for a Kotlin function apply to @Composable.
  - But, **no heavy computation work** in a @Composable. Otherwise, UI render takes too long
- Important facts about execution of @Composable functions
  - It can invoke another @Composable function
  - They can be invoked in any order
  - They can be invoked in parallel
  - They can be may be **invoked multiple times as frequently as animation rate (30 times/sec)**
  - The Compose Compiler transforms a @Composable function to an ordinary function

@Composable Order of Execution

```
@Composable
fun MyMainScreen() {
    Column {
        TopContent()
        MainContent()
        Footer()
    }
}
```



- DO NOT assume the three functions `TopContent()`, `MainContent()`, and `Footer()` execute sequentially in the order you write
- DO NOT impose content dependency among the the composable.
  - If MainContent() depends on outcome of executing TopContent(), such as updating a “global” variable, the UI may not show up as expected

Key Concepts #2: UI State Variables

@Composable
UI state variable
(Re)composition

- (UI) State: any variable(s) whose value affect the appearance of the UI
  - Function arguments
  - Local variables (require special declaration, to make the data **observable by the UI**)

```
@Composable
fun Greeting(name: String) {
   val inputAge by remember { mutableStateOf(“10”) } // UI State
   val minAge = 21                                   // NOT a UI state
   Text(“Hello $name. You must be at least $minAge years”)
   TextField (value = inputAge,
              onChanged = { inputAge = it })
}
```


Incorrect use of local variables

@Composable
UI state variable
(Re)composition

During recomposition, @Composable functions are re-invoked/re-evaluated

- All local variables are reallocated

```
@Composable
fun Greeting(name: String) {
    var inputAge by remember { mutableStateOf(“10”) }
    TextField (value = inputAge,
               onChanged = { inputAge = it })
}
```

```
@Composable
fun Greeting(name: String) {
    var inputAge = “10”
    TextField (value = inputAge,
               onChanged = { inputAge = it })
}
```

- `mutableStateOf()` makes a variable reactive
- `remember()` retains the value of the variable across recomposition

Kotlin “by”: property delegation

```
val foo by FooClass()
var bar by BarClass()
```

- The getter() logic of `foo` is handled by the `getValue()` method in `FooClass`
- The getter() and setter() logic of `bar` are handled by the `getValue()` and `setValue()` methods in `BarClass`
- Required import

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


Declaring UI State Variables

```
// Declaration
var inputAge by remember { mutableStateOf(“10”) }
var dogAge = remember { mutableStateOf(3) }
val (myAge, setMyAge) = remember { mutableStateOf(10) }
// The last syntax is similar to React useState()
```

```
// Usage
inputAge = “14”
dogAge.value = 5
setMyAge(15)
println(“$myAge”)  // 15
```

```
// Declaration of collections
val cities by remember { mutableStateOf(listOf(“Ada”, “Bath”, “Erie”)) }
val counties by remember { mutableStateOf(mutableListOf(“Kent”, “Ottawa”)) }
val purchase = remember { mutableStateOf(mutableListOf<String>()) }
```

```
// Usage
~~cities.add(“Rome”)~~   // Can’t alter immutable list
counties.add(“Ingham”)
purchase.value.add(“Candy”)
```

Key Concepts #3: (Re)Composition

@Composable
UI state variable
(Re)composition

- (UI) State: any variable(s) whose value affect the appearance of the UI
  - Function arguments
  - Local variables (require special declaration)
- Updates to these variables will trigger (UI) recomposition

```
@Composable
fun Greeting(name: String) {
   val inputAge by remember { mutableStateOf("10") }
   Text("Hello $name")
   TextField (value = inputAge,
              onChanged = { inputAge = it })
}
```

- *If **name** changes, Text is recomposed (so is Greeting)*
- *If **inputAge** changes, TextField is recomposed (so is Greeting)*

remember with "key"

```
@Composable
fun SomeWork() {
    val codeLen by remember { mutableStateOf(3) }
    val apple by remember {
        List(codeLen) { randomInt() }   // evaluated only ONCE
    }
    val banana by remember(codeLen) {
        List(codeLen) { randomInt() }   // re-evaluated when codeLen changes
    }
}
```

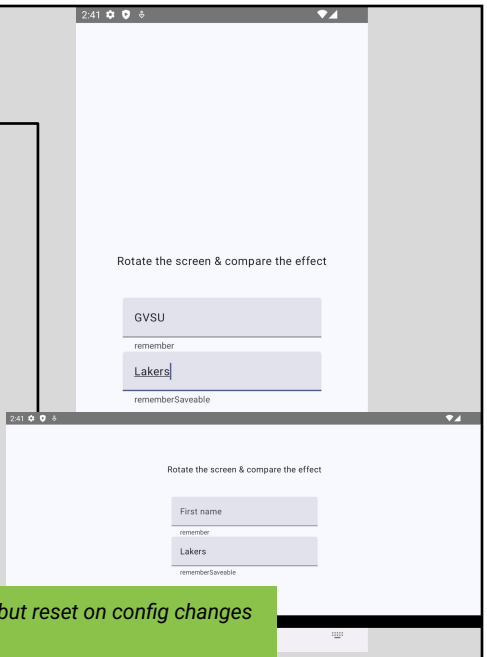
- Upon initial composition: `apple` and `banana` get 3 random integers
- When `codeLen` changes to 5, `SomeWork()` is recomposed
  - `apple` will NOT get updated
  - `banana` will get reinitialized to 5 new random numbers

Configuration changes?

```
remember { ... }
rememberSaveable { ... }
```


remember(Saveable) { . . . . . }

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

@Composable
fun Counting() {
   var firstName by remember { mutableStateOf("") }
   var lastName by rememberSaveable { mutableStateOf("") }
   Column(
       horizontalAlignment = Alignment.CenterHorizontally,
       modifier = Modifier.fillMaxWidth()
   ) {
      TextField(value = firstName, onValueChange = {firstName = it})
      TextField(value = lastName, onValueChange = {lastName = it})
   }
}
```



- *`remember()` retains current value across multiple calls of recomposition, but reset on config changes*
- *`rememberSaveable()` also retains the content on config changes*

Sample Code
app-arch-compose

Library Dependencies (Kotlin Script Syntax)

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

Warning

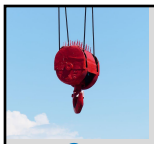
- *It is challenging to know exactly which library to include, so many of them to choose from. Google should provide a better documentation!*
- *Mixing newer versions of libraries with older ones may fail your build with a “duplicate classes” error. Be wise in choosing the version numbers.*

Compose Modifier

Best Practice: *Every @Composable function should define a Modifier parameter (with default initializer) and pass it to the top-level @Composable*

```
@Composable
fun Greeting(name: String) {
  val inputAge by remember { mutableStateOf("10") }
  Column {
    Text("Hello $name")
    TextField (value = inputAge,
               onChanged = { inputAge = it })
  }
}
```

```
@Composable
fun Greeting(modifier: Modifier = Modifier, name: String) {
  val inputAge by remember { mutableStateOf("10") }
  Column(modifier = modifier) {
    Text("Hello $name")
    TextField (value = inputAge,
               onChanged = { inputAge = it })
  }
}
```

State Hoisting

Sometimes, state must be **supplied by** and **"returned" to** parent @Composable

```
@Composable
fun MyScreen(name: String) {
  var inputAge by remember { mutableStateOf("10") }
  Column {
    Text("Hello $name")
    TextField (value = inputAge,
               onChanged = { inputAge = it })
  }
}

@Composable
fun ParentContent() {
  MyScreen("World")
}
```

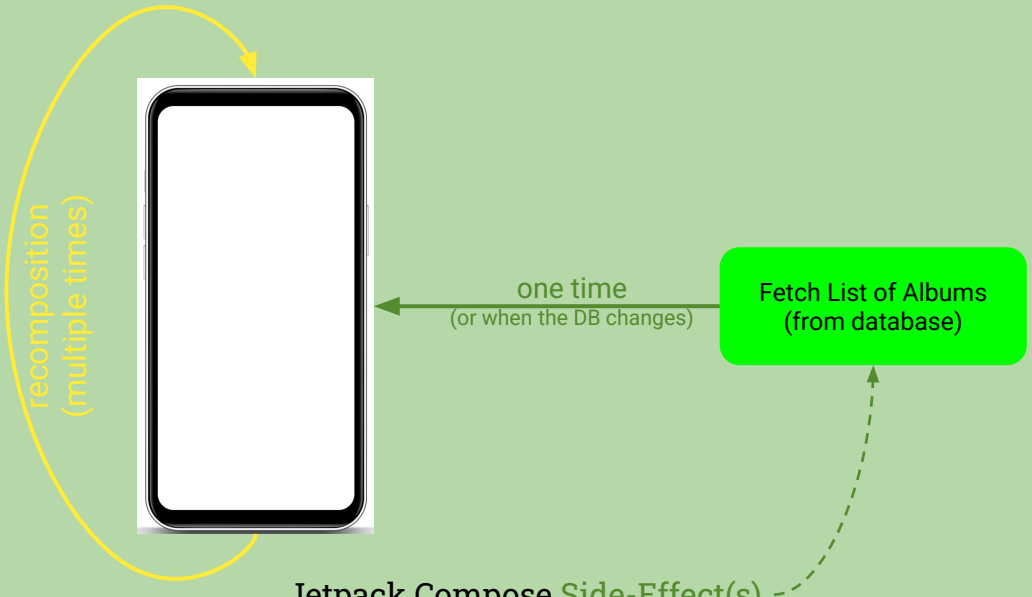
```
@Composable
fun MyScreen(name: String, inputAge: String,
             onAgeUpdated: (String) -> Unit) {
  Column {
    Text("Hello $name")
    TextField (value = inputAge,
               onChanged = { onAgeUpdated(it) })
  }
}

@Composable
fun ParentContent() {
  var age by remember { mutableStateOf("10") }
  MyScreen("World", age) {
     age = it
  }
}
```

More Caching Functions

Function	Purpose
`remember {}, remember(key) {}`	Preserve the value of a state variable across recomposition
`rememberSaveable {}`	Similar to remember {} but also preserve the value across device configuration changes
`rememberUpdatedState {}`	Remember the most recent value of a function **parameter** used in recomposition, so subsequent update to the parameter will NOT trigger recomposition. Practical use: memoize lambda expressions used in a side effect
`rememberCoroutineScope {}`	Provide a coroutine scope to run suspendable function inside a @Composable function
`rememberNavController{}`	Provide a navigation controller to NavHost
`rememberScaffoldState {}`	(Material 2 only) use for managing drawer and snackbar
`rememberLazyListState {}`	control the scroll position and behavior of LazyColumn and LazyRow

side Effects side

Main Dish vs. Side Dish(es)



Jetpack Compose Side-Effect(s)

“Main” Effect vs. Side Effects (of a @Composable)

- “Main” effect: the work of (re)composing the UI
- Side Effects:
  - work defined **inside** `@Composable fun xxxScreen()`
  - work for updating UI state variables
  - to be executed outside/separate from (re)composition logic
- Why needed?
  - Occasionally we need code/work done inside a @Composable, but **should not be repeated** on every recomposition
  - Invoking a work which must be performed inside a coroutine
  - (Long) running cancellable tasks`

Side-Effects

Type of effect	When run?
`LauncedEffect(Unit) {` `}`	Only once
`LauncedEffect(gameLevel) {` `}`	After gameLevel changes, also cancel previous co-routine

`DisposableEffect(gameLevel) {`
`  onDisposed {`
`  }`
`}`
	After gameLevel changes, at the end of composition. The onDisposed lambda executes before gameLeve changes to the new value.

`SideEffect {` `}`	After every (re)composition