Android Essential

Topics

- Building blocks
- Activity Lifecycle
- Application Architecture: Managing Code & Data
  - ViewModel
  - (Mutable)State or (Mutable)Flow for Jetpack Compose Apps

Essential Building Blocks

- **Activity / Fragment**: manages user interactions with the UI on the screen

- **Service**
- **Broadcast Receiver**
- **Content Provider**

$\Big\}$ *"middleware" components*

Service

- An app component that does not have a user interface
- Runs in the *background* *(detached from the UI)*
    - ***Foreground*** *Services: effect of background work is noticeable to user*
        - *Example: music player,* ***pedometer****, podcast player, ...*
    - ***Background*** *Services: effect of background work is NOT directly noticeable to user*
        - *Example: calendar sync, podcast files download, email sync,* ***pedometer (data logging)****....*

Broadcast Receiver

- An app component that handles broadcast **system messages** sent by other Android apps or by the Android system
- Examples
  - Switch from/to Airplane Mode
  - Incoming SMS/text messages
  - Incoming Phone Calls
  - 2FA push notifications: Duo
  - Time zone change
  - Headphone jack (un)plugged
  - Low battery
  - Photo Taken by device
  - *Many more (hundreds of them)*
- List of Broadcast Intents

Content Provider

- **Data Sharing**: allows data of an app to be made available to other apps
- Data can be stored in
  - a SQLite DB
  - Android File system
  - Other local storage
- Examples
  - Voice Recorder
  - Phone Contacts
  - Phone call log
  - Photo Gallery

AndroidManifest.xml

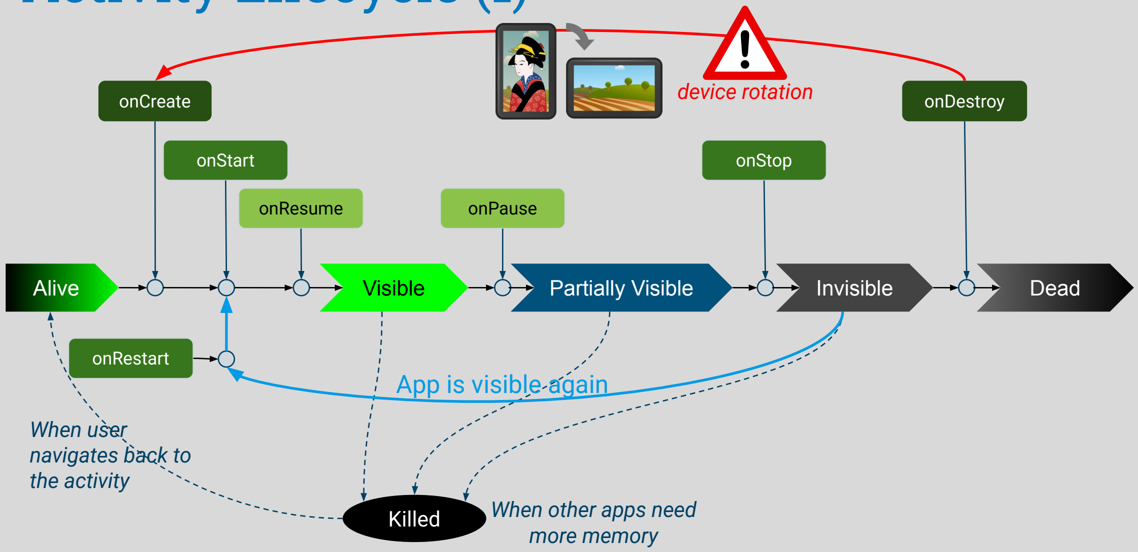
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools">
   <application
       <activity android:name=".MainActivity"
       </activity>
       <service android:name=”.MyAudioService”>
       </service>
       <receiver android:name=”.MyAppReceiver”>
       </receiver>
       <provider android:name=”.MyPhotoProvider”>
       </provider>
   </application>
</manifest>
```

Activity Lifecycle

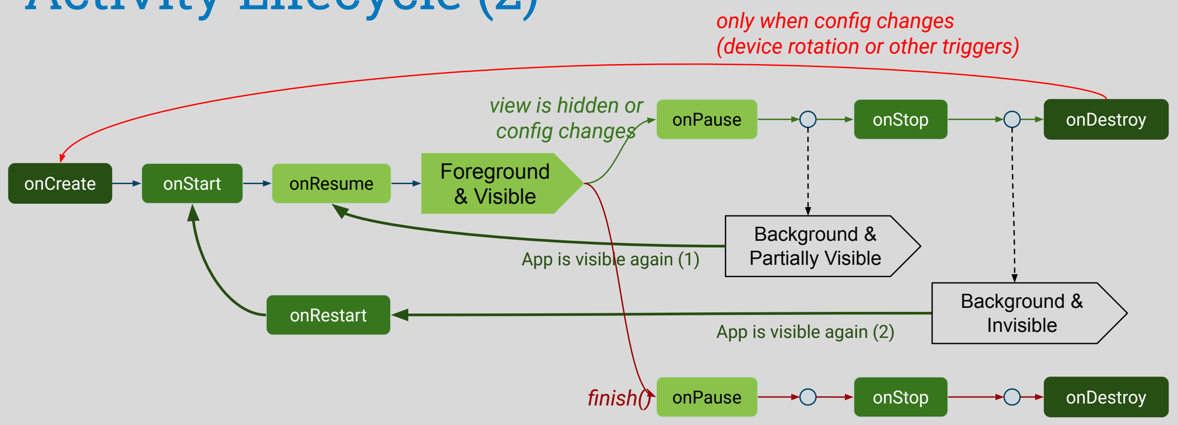
Activity Stages of Life

- Alive
- Foreground (interacting with user), and **entirely visible**
- Background/non-interactive and **partially visible**
- Background/non-interactive and **completely invisible**
- Dead

Activity Lifecycle (1)



Activity Lifecycle (2)



- Foreground ⇒ Background: Activity is pushed to the Activity Stack
- Background ⇒ Foreground: Activity is popped from the Activity Stack

Lifecycle Function Pairs

Becoming active	Becoming inactive	Usage Example(s)
`onCreate`	`onDestroy`	create/destroy timer(s)
`onStart / onRestart`	`onStop`	start/stop game time limit timer
`onResume`	`onPause`	start/stop “engagement” timer

Lifecycle (androidx.lifecycle:lifecycle-xxxx)

- On older Android runtime handling of lifecycle events is coupled with Activity/Fragment
- On newer Android runtime, lifecycle events can be handled outside of Activity/Fragment

```
class MyClass: DefaultLifeCycleObserver {
  override fun onCreate (owner: LifecycleOwner) {
    TODO("Write your code here")
  }

  override fun onStart (owner: LifecycleOwner) {
    TODO("Write your code here")
  }
  override fun onResume (owner: LifecycleOwner) {
    TODO("Write your code here")
  }

  override fun onPause(___) { }
  override fun onStop(___) { }
  override fun onDestroy(___) { }
}
```

⚠ Data in Android Activity will NOT survive after ANY configuration changes

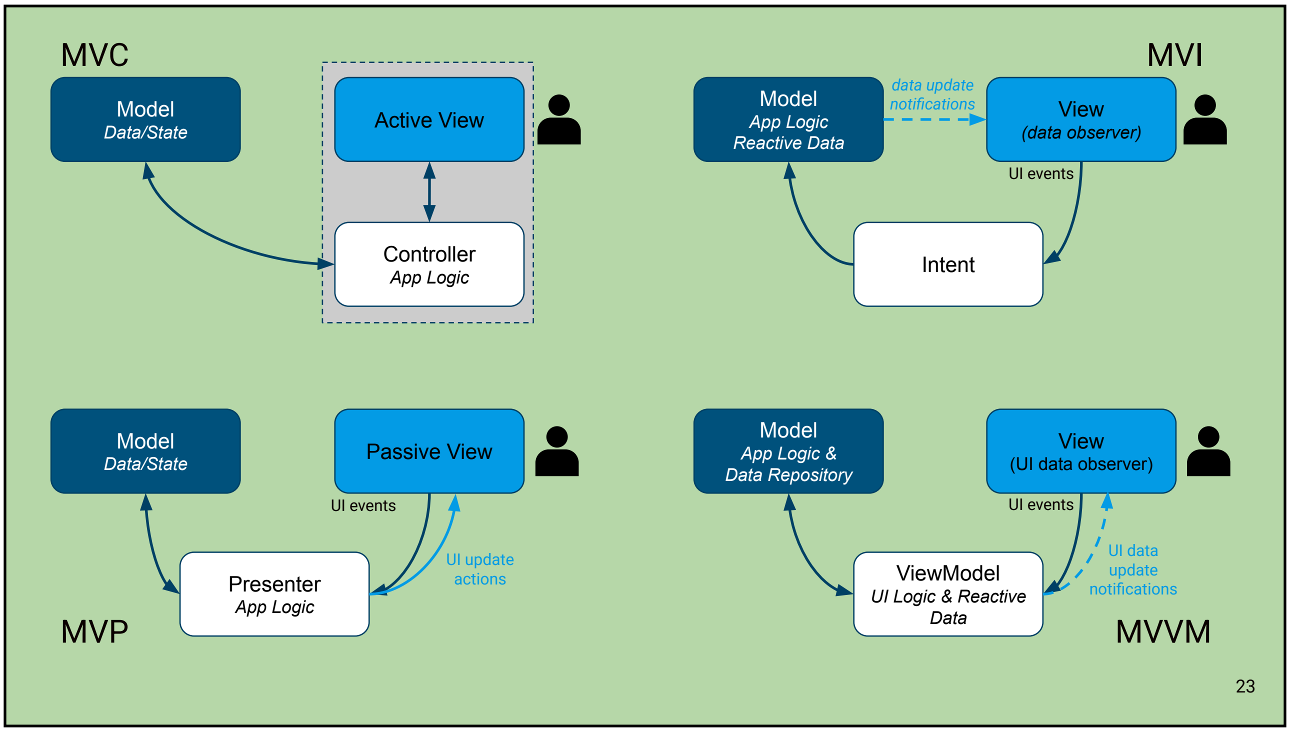
Configuration Changes

- Screen rotation (is just one of the easiest to understand)
- (Un)folding foldable device (Samsung Z Flip/Fold, Pixel 9 Fold)
- Switching to a different language
- Changing font scale factor
- Switching to a different font
- Switching the system them (dark vs. light mode)
- (Dis)connect keyboard

Application Architecture

Choices of Design Architecture

- Objectives
  - clean separation UI logic and application logic
  - single source of truth (app data)
  - testable application logic (separate from UI testing)
  - test the UI logic by mocking the application logic
- MVP: Model View **Presenter**
- MVC: Model View **Controller**
- MVI: Model View **Intent**
- MVVM: Model View **ViewModel**



Recommended Android Architecture

MVVM

- **Model**: data objects of the application (Ticket, Movie, Seat, Show, Theater)
- **View**: present data to user, take user input, handle UI actions
  - *Should not contain* **business logic** of the app
  - *May contain* **UI logic**
- **ViewModel**
  - Provide (**reactive**) data that can be “consumed/observed” by the View
  - May contain business logic (for small applications)
- (Optional) Data Repository (for larger applications)
  - Single source of truth for application data
  - Business logic in ViewModel should be moved here

MVVM in Android

Role	Implementation	Description	Typical Content
Model	`data class`	Holds application data	Data only, no methods/functions
View	`Activity` or `@Composable fun`	Handle *only* what the user sees or touches on the screen	UI logic and temporary local variables needed by the UI **No business logic** Relay user actions to ViewModel
ViewModel	`class (subclass of ViewModel)`	Provides/present data for the view, Handles UI logic	Observable data for the view Application Data & Business logic (for small apps)
Repository	`class`	Single source of truth for application data	Application Data Business logic (for larger apps)

UI Data vs. Business Data (Use case: Movie Tix Purchase)

	UI Data	Business Data
Number of tickets purchased	✔	
Total tickets available		✔
Movie Showtimes		✔
User-selected show time	✔	
Color of selected seats	✔	
Available seats		✔
Member bonus points		✔

- Keep UI data as state variable(s) in @Composable function
- Keep business data in ViewModel or Data Repository

ViewModel Design Guidelines

- Use Kotlin Flow for defining **reactive data**
  - Past recommendations: use LiveData
- Define business data as private MutableStateFlow
  `private val _userId = MutableStateFlow("Hello")`
- Provide a public **immutable** variant of the MutableStateFlow
  `val userId = _userId.asStateFlow()`
- Provide a public method to manipulate the private MutableStateFlow
  `fun manipulateUser(/* args */) { }`
- Use the class `init {}` block to initialize any data (if necessary)

MVVM Demo

(GitHub)

MVVM Example: TicketViewModel

```
class TicketViewModel: ViewModel() {
  private val _availableTickets = MutableStateFlow<Int>(10)

  // public immutable copy
  val availableTickets = _availableTickets.asStateFlow()

  fun purchaseTickets(tixCount: Int) {
    if (tixCount <= _availableTickets.value)
      _availableTickets.value -= tixCount
  }
}
```

Using State variables in @Composable UI

- Use Kotlin property delegation (`by`) to obtain a reference to flow variable
- Use `.collectAsState()` to make changes on state flow variable observed by UI recomposition logic

```
val userId by theViewModel.userId.collectAsState()
val tixAvail by tixViewModel.availableTickets.collectAsState()
```

- Use the variable (`userId` or `tixAvail`) somewhere in the UI

```
@Composable
fun TicketScreen(vm: TicketViewModel) {
   var numTix by remember {mutableStateOf("")}
   val availableTicket by vm.availableTickets.collectAsState()
   Column {
       Text("Available ticket ${availableTicket}")
       OutlinedTextField(
           onValueChange = {numTix = it},
           value = numTix
       )
       Button(
           onClick = {
               vm.purchaseTickets(numTix.toInt())
           },
       ) {
           Text("Buy")
       }
   }
}
```

ViewModel Instantiation

```
// Outside @Composable scope
// Dependency: implementation("androidx.activity:activity-ktx:x.y.z")
import androidx.activity.viewModels
class MainActivity: AppCompatActivity() {

  override fun onCreate() {
    super.onCreate(______);
    val myViewModel: TicketViewModel by viewModels()
    setContent {
      TicketScreen(myViewModel)
    }

  }
}
```

```
// Inside @Composable scope
// Dependency:
//   implementation("androidx.lifecycle:lifecycle-viewmodel-compose:x.y.z")
import androidx.lifecycle.viewmodel.compose.viewModel
class MainActivity: AppCompatActivity() {

  override fun onCreate() {
    super.onCreate(______);
    setContent {
      val myViewModel: TicketViewModel = viewModel()
      TicketScreen(myViewModel)
    }
  }
}
```

ViewModel Cheatsheet

Nullable Types in ViewModel StateFlow

```
class TicketViewModel: ViewModel() {
  private val _one = MutableStateFlow<Int>(10)    // <Int> can be omitted
  private val _two = MutableStateFlow(20)

  // Kotlin compiler can't infer the type from initial value null
  private val _lastBackupSize = MutableStateFlow<Long?>(null)
  private val _lastBackupDate = MutableStateFlow<String?>(null)
}
```

MutableStateFlow<Object>

```
// Provide a data class with default constructor
// Both name and age are immutable properties
data class Person (val name:String = "", val age: Int = 0)

val yourState = MutableStateFlow<Person>(Person())
val myAge = MutableStateFlow<Int>(20)
```

```
myAge.value = myAge.value + 1
myAge.update { curr ->
   curr + 1
}
myAge.update {
   it + 1
}

// This won't work (age is immutable)
yourState.value.age = 21
```

```
// Update only the age
yourState.update {
  it.copy(age = 21)
}
```

```
// Update to 4 years older
yourState.update {
  it.copy(age = it.age + 4)
}
```

```
// Update both name and age
yourState.update {
  it.copy(age = 21, name = "Ben")
}
```

Coalesce related variables into data class (1)

```
class TicketViewModel: ViewModel() {
  private val _inProgress = MutableStateFlow(false)
  private val _availableTickets = MutableStateFlow(10)
  // public immutable copy
  val availableTickets = _availableTickets.asStateFlow()
  val inProgress = _inProgress.asStateFlow()

  fun purchaseTickets(tixCount: Int) {
  }
}
```

```
data class TixPurchaseState(
   val inProgress: Boolean = false, val availableTickets: Int = 0)

class TicketViewModel: ViewModel() {
  private val _ticketState = MutableStateFlow(TixPurchaseState())
  val ticketState = _ticketState.asStateFlow()

  init {
     _ticketState.update {
       it.copy(availableTickets = 10)
     }
  }
  fun purchaseTickets(tixCount: Int) { /* not shown */ }
}
```

Coalesce related variables into data class (2)

```
class TicketViewModel: ViewModel() {
  private val _availableTickets = MutableStateFlow(10)
  private val _inProgress = MutableStateFlow(false)

  fun purchaseTickets(tixCount: Int) {
    _inProgress.value = true
    if (tixCount <= _availableTickets.value) {
      _availableTickets.value -= tixCount
      _inProgress.value = false
    }
  }
}
```

```
class TicketViewModel: ViewModel() {
  private val _ticketState = MutableStateFlow(TixPurchaseState())
  val ticketState = _ticketState.asStateFlow()

  fun purchaseTickets(tixCount: Int) {
    _ticketState.update { current ->
       current.copy(inProgress = true)
    }
    if (tixCount <= _ticketState.value.availableTickets) {
      _ticketState.update {
        it.copy(inProgress = false,
                availableTickets = it.availableTickets - tixCount)
      }
    }
  }
}
```

```
@Composable
fun TicketScreen(vm: TicketViewModel) {
   val availableTickets by vm.availableTickets.collectAsState()
   Column {
       Text("Available ticket ${availableTickets}")
       Button(
           onClick = { vm.purchaseTickets(3) },
       ) { Text("Buy") }
   }
}
```

```
@Composable
fun TicketScreen(vm: TicketViewModel) {
   val tixState by vm.ticketState.collectAsState()
   Column {
       Text("Available ticket ${tixState.availableTickets}")
       Button(
           onClick = { vm.purchaseTickets(3) },
       ) { Text("Buy") }
   }
}
```