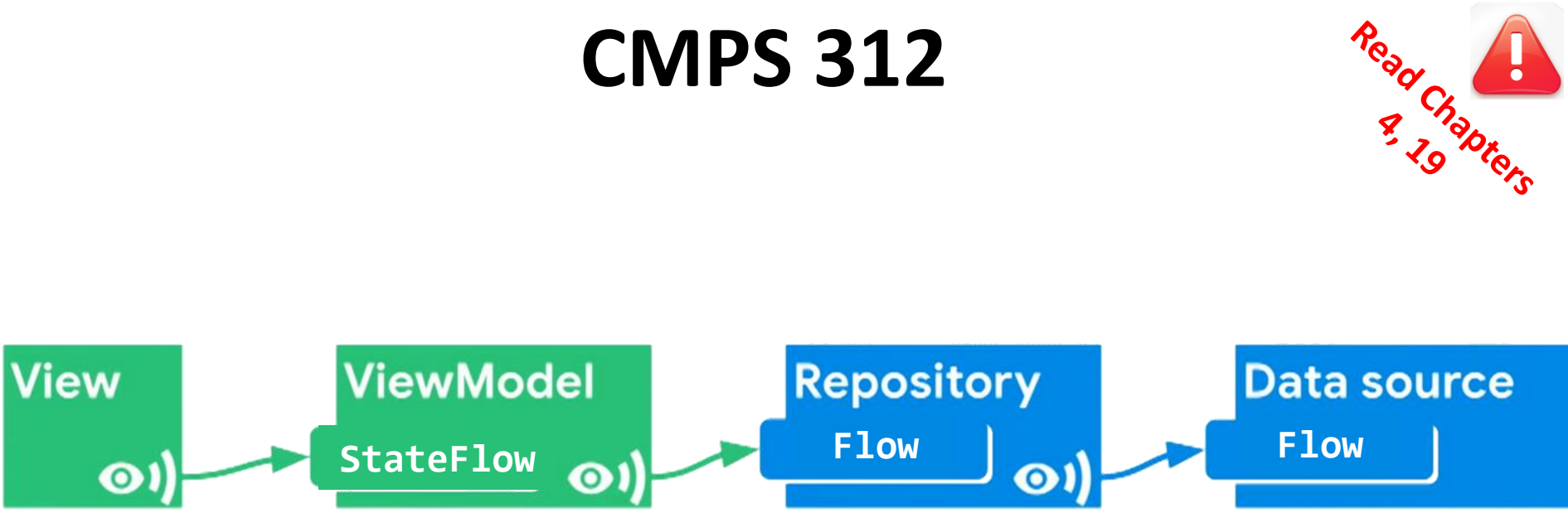
CMPS 312



Model-View-ViewModel (MVVM) Architecture

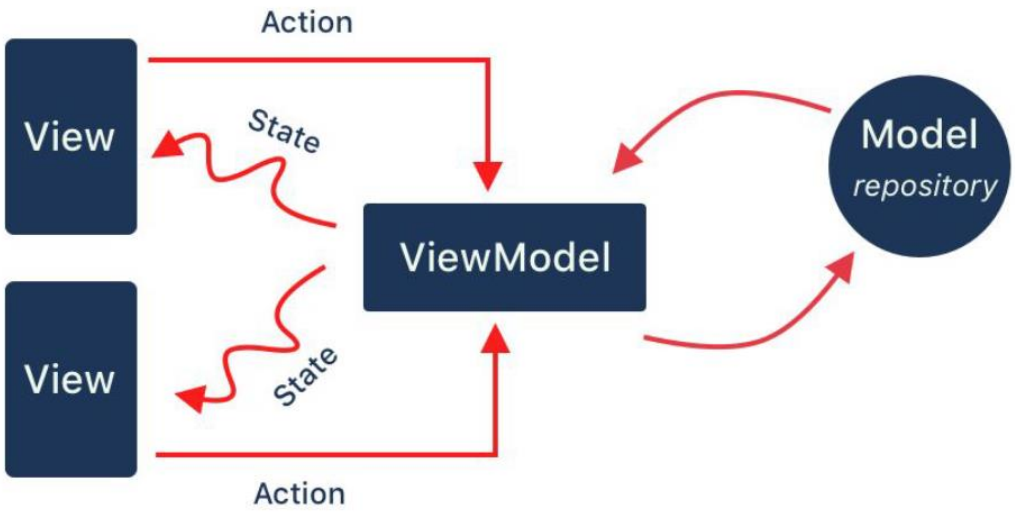
Dr. Abdelkarim Erradi

CSE@QU

Outline

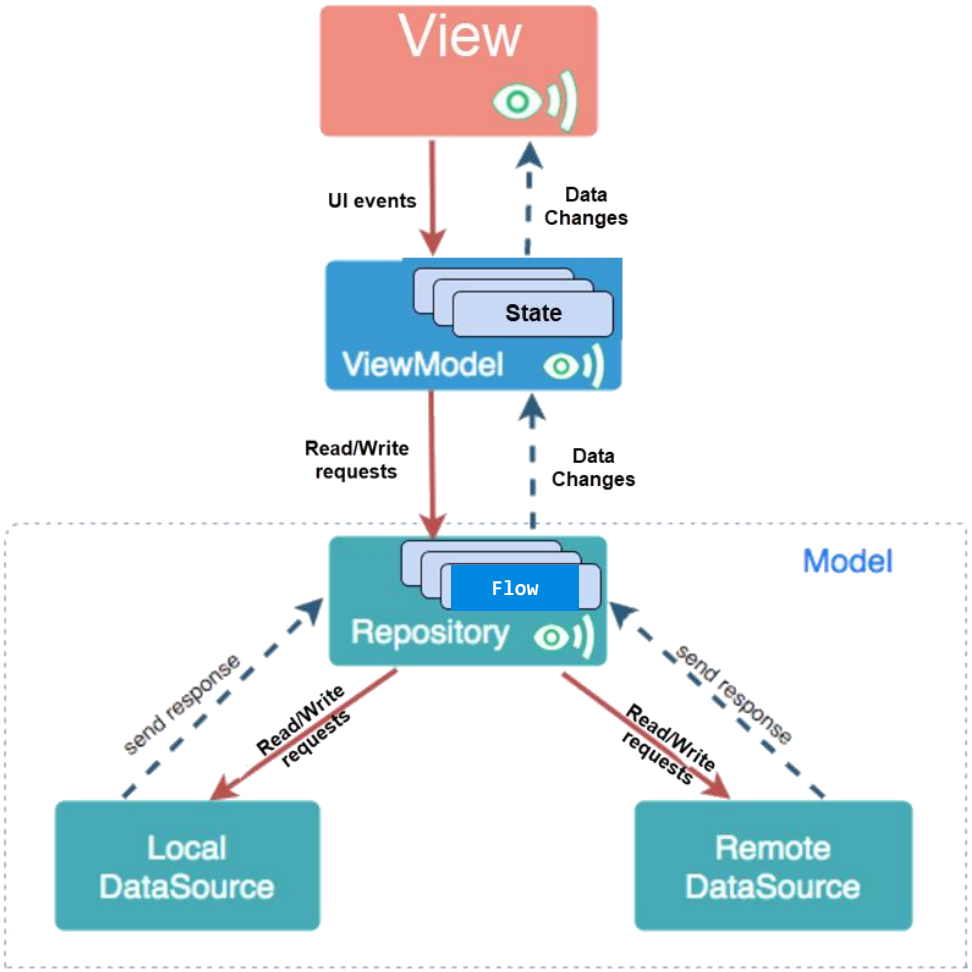
1. Model-View-ViewModel (MVVM)
2. ViewModel
3. State variables
4. Flow

MVVM Architecture



Back

Model-View-ViewModel (MVVM) Architecture



View = UI to display state & collect user input

- It **observes** state changes from the ViewModel to update the UI accordingly

- Calls the ViewModel to handle events such as button clicks, form input, etc.

ViewModel

- Manages **state** (i.e, data needed by the UI)
  - Interacts with the Model to read/write data based on user input
  - Expose the state as **Observables** that the UI can subscribe-to to get data changes
- Implements UI logic / computation (e.g., data validation)

Model - handles data operations

- Model has **entities** that represent app data
- **Repositories** read/write data from either a Local Database (using **Room** library) or a Remote Web API (using **Ktor** library)
- Implements data-related logic / computation

MVVM Key Principles

- Separation of concerns:
  - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
  - ViewModel **has no direct reference to the View**
  - View never accesses the model directly
  - Model unaware of the view
- Observer pattern:
  - View observes the ViewModel (to get data changes)
  - ViewModel observes the Model (to get data changes)

- Inversion of Control:
  - Uses Dependency Injection instead of direct instantiation of objects

e.g., `val scoreViewModel = viewModel<ScoreViewModel>()`

Advantages of MVVM

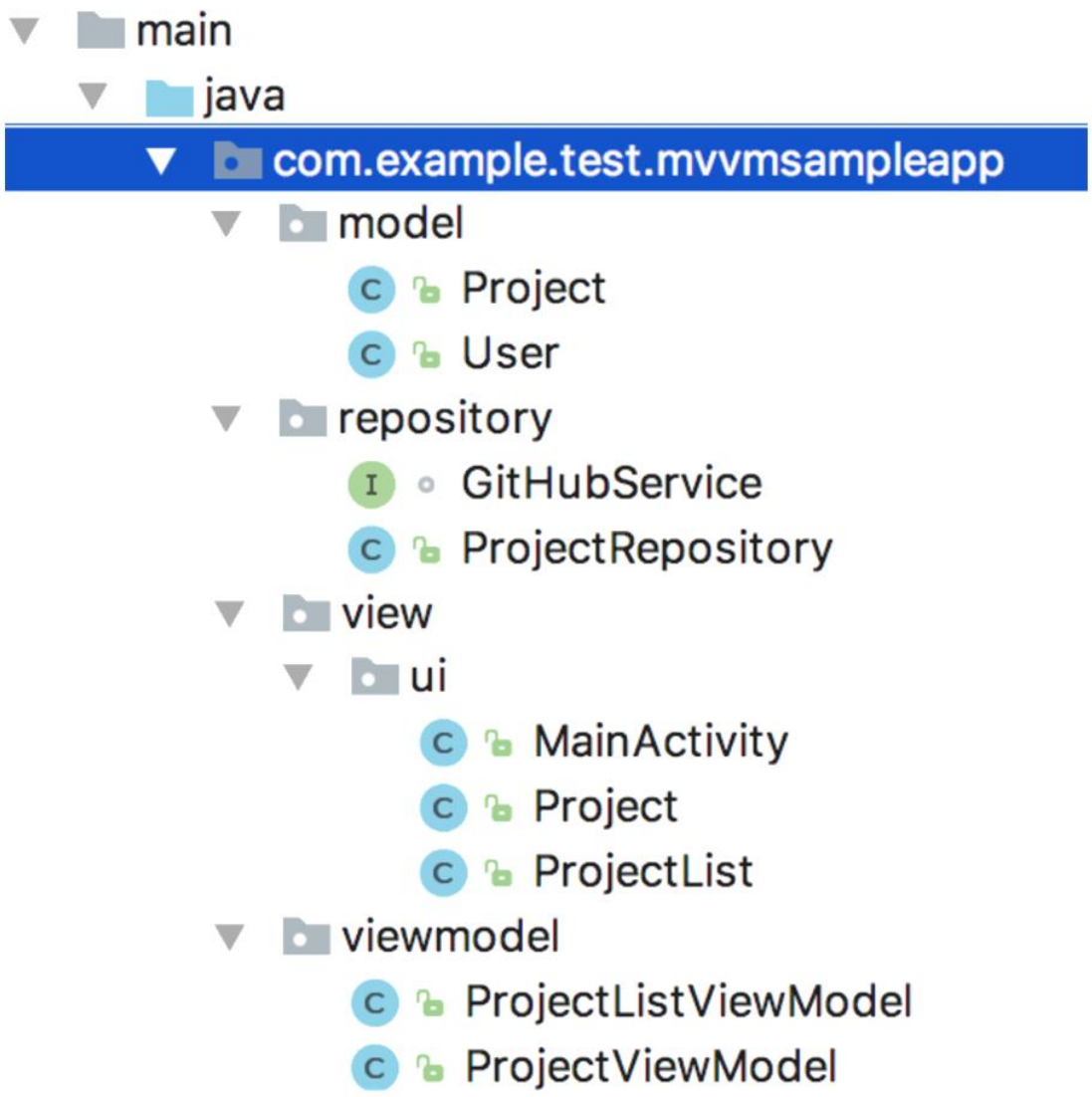
- ***Separation of concerns* = separate UI from app logic**
  - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
  - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
  - Easier **testing** of the App components

MVVM => Easily maintainable and testable app

Android Architecture Components

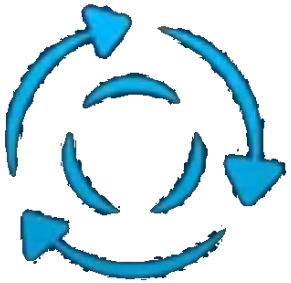
- Android architecture components are a collection of libraries to ease developing MVVM-based Apps
- Part of Android Jetpack  They include:
  - ViewModel stores UI-related data that isn't destroyed on screen rotation
  - StateFlow data holder that notifies the View when the model data changes
  - Room to read / write data to local SQLite database

Recommended Project Structure



You may organize the view by feature

ViewModel



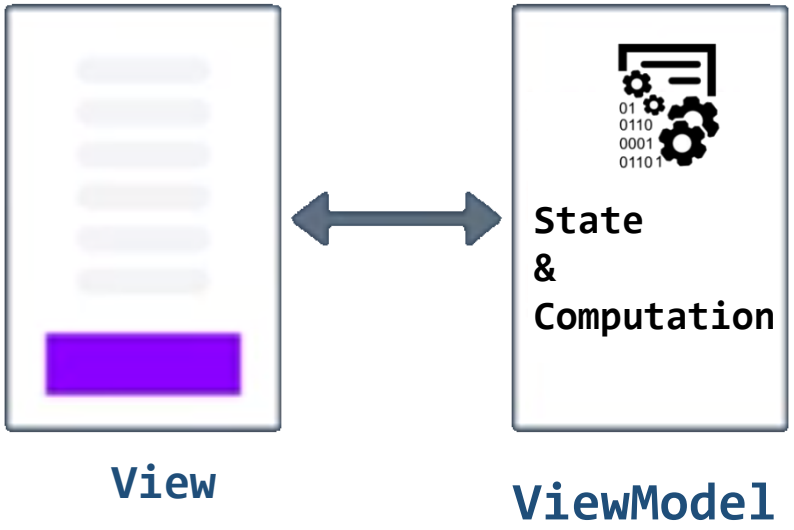
Lifecycle Aware

Survives Config Changes

Back

ViewModel

- ViewModel is used to **store and manage state** (i.e., data needed by the UI)
  - in a lifecycle conscious way
  - allows **state** to survive device configuration changes such as *screen rotations* or *changing the device's language*

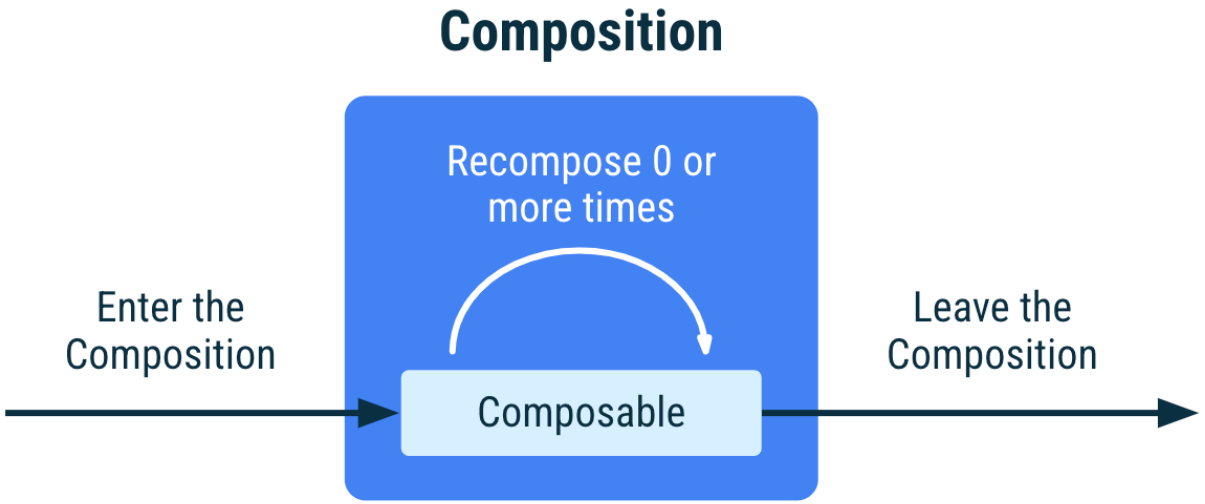


- If the system destroys or re-creates a UI component (e.g., when the screen rotates), any state stored in the View is lost
  - State is NOT retained across configuration changes (landscape/portrait)

Use **ViewModel**:

- Manages state
- Read/write data from a Repository

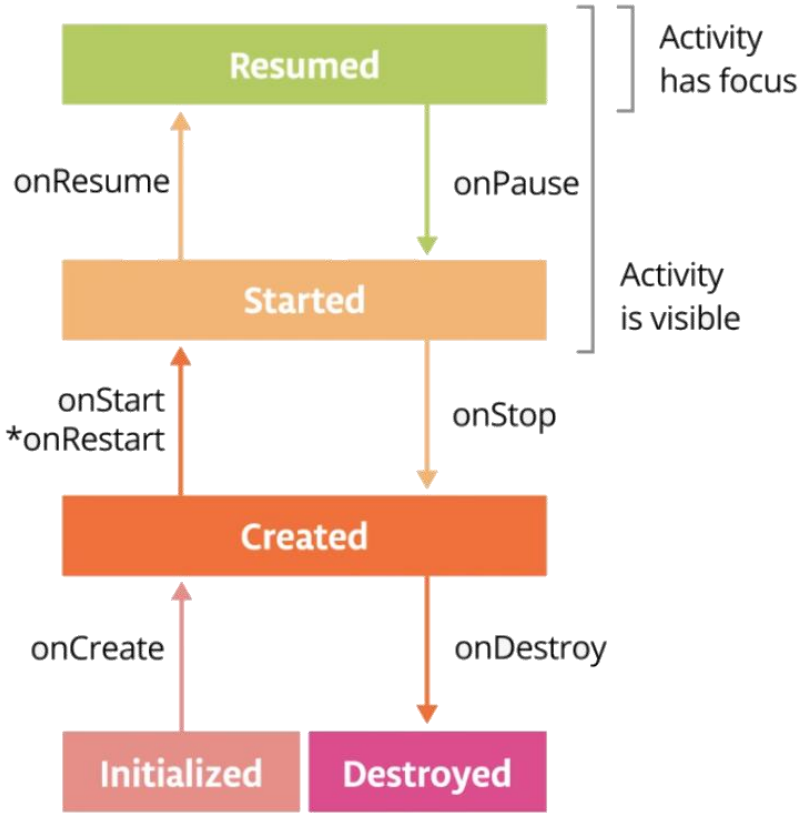
Composable Lifecycle



- When the screen is displayed for the first time, the Composable **enters** the Composition, gets **recomposed** 0 or more times, and **leaves** the Composition
- Recomposition is triggered by a change to a State<T> object. Compose tracks these and runs all composables in the Composition that read that particular State<T>

https://developer.android.com/jetpack/compose/lifecycle

Activity Lifecycle

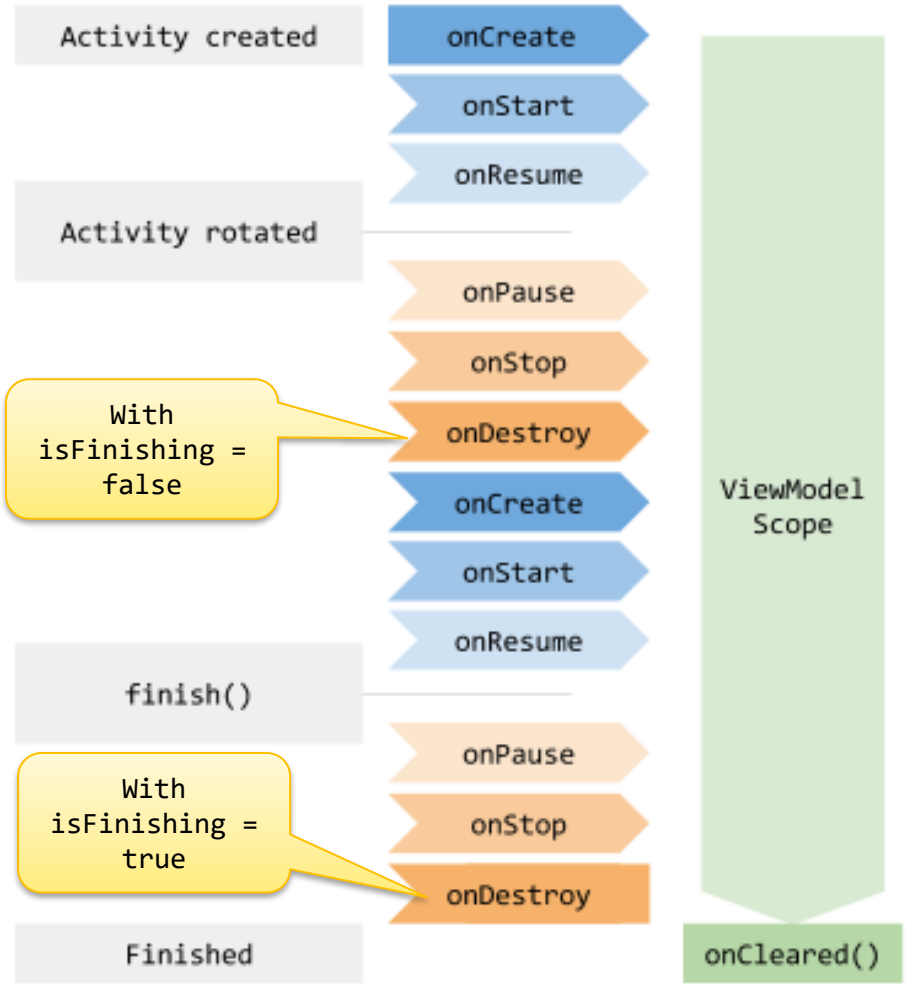


An activity has essentially **four states**:

- ***Resumed*** if the activity in the foreground of the screen (has focus)
- ***Started*** if the activity has lost focus but is still visible (e.g., beneath a dialog box).
  - When the user returns to the activity, it is **resumed**
- **Created** if the activity is completely obscured by another activity.
  - When the user navigates to the activity, it must be **restarted** and restored to its previous state.
- **Destroyed** when the user closes the app or if the activity is killed (when memory is needed or due to `finish()` being called on the activity)

ViewModel Lifecycle

- ViewModel object can be scoped to the main activity
- However, it has a **longer lifespan** compared to the associated Activity which may undergo a rotation and get recreated
- It remains in memory until the activity is `completely destroyed`
  - When the activity is recreated (after a screen rotation) the associated ViewModel remains alive



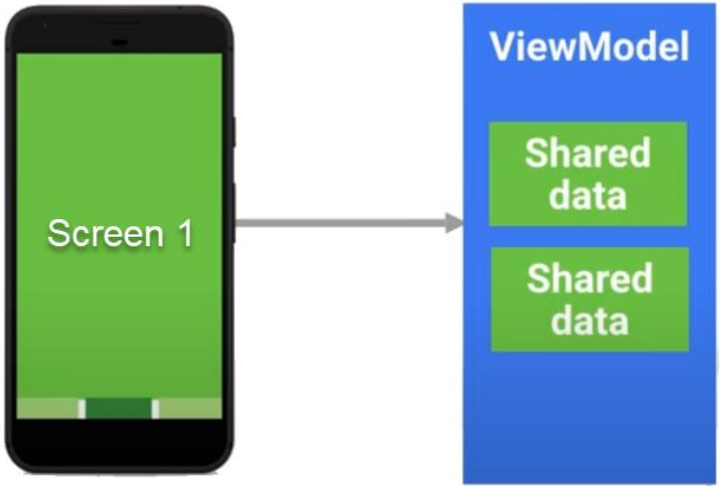
ViewModel Example

```
class ScoreViewModel : ViewModel() {
    // Private mutable state variables
    private var _team1Score = mutableStateOf(0)
    // Public State variables
    val team1Score : State<Int> = _team1Score
    fun onIncrementTeam1Score() { _team1Score.value++  }
```

```
@Composable
fun ScoreScreen() {
    // Get an instance of the ScoreViewModel
     val scoreViewModel = viewModel<ScoreViewModel>()
    Text(text = scoreViewModel.team1Score.value)
    Button(onClick = { scoreViewModel.onIncrementTeam1Score() }) {
       Text(text = "+1")
    }
    ...
}
```

Shared data between Screens using ViewModel

- Screens can **share** data using a shared **View Model** class that extends `ViewModel()`




```
@Composable
fun ProfileScreen(userId: Int) {
 /* Get an instance of the shared viewModel
    Make the activity the store owner of the viewModel
    to ensure that the same viewModel instance is used for all screens */
 val userViewModel = viewModel<UserViewModel>()
 val user = userViewModel.getUser(userId)
... }
```

“no contexts in ViewModels” rule

- ViewModel should **not be aware of the View** who is interacting with
  => It should be decoupled from the View

  - ViewModel should not hold a reference to Activities or Views (i.e. Composables)
  - Should not have any Android framework related code
  - As this defeats the purpose of separating the UI from the data
  - Can lead to memory leaks and crashes (due to null pointer exceptions) as the ViewModel outlives the View
    - if you rotate an Activity 3 times, 3 three different Activity instances will be created, but you only have one ViewModel instance

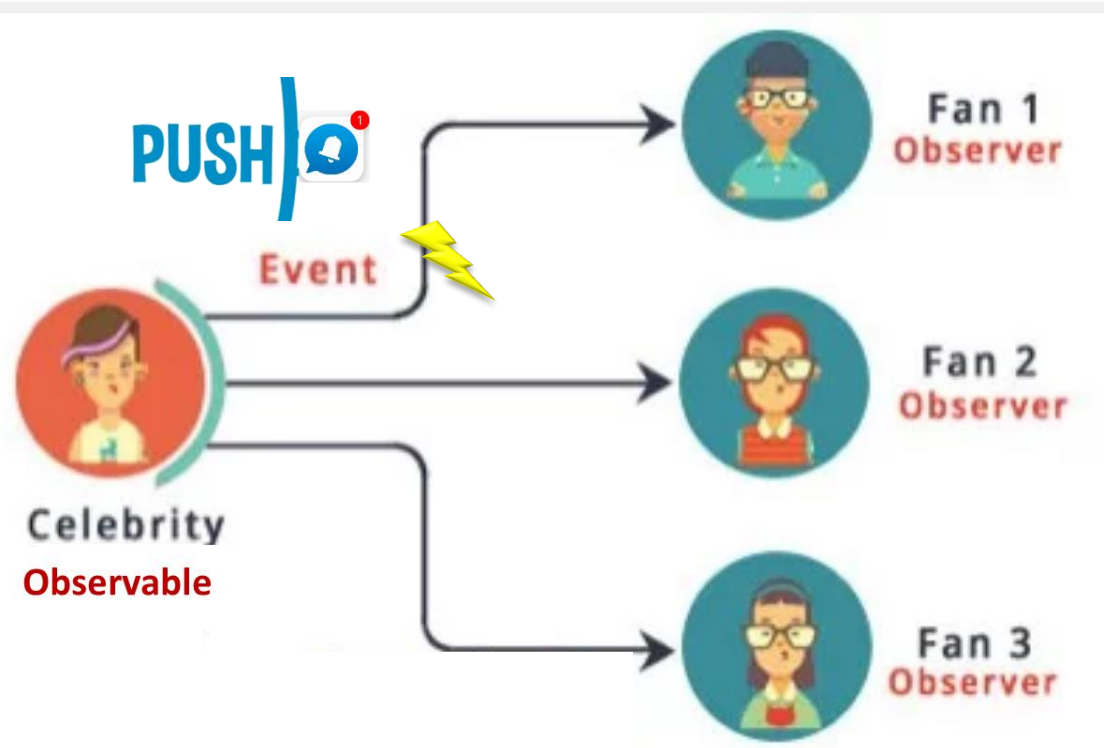
State variables

State variables

- State in an app is any value that can change over time
- A **State** variable is an **observable data holder** whose reads and writes are observed by Compose to trigger UI recomposition
  - State variable warps around an object and allows the view to **observe** it
- The ViewModel exposes **State** variables that the View observes and update the UI accordingly
  - This decouples the ViewModel from the View: the ViewModel does NOT have any direct reference to the View
  - The View can observe the ViewModel State variables for changes then update the UI (aka recomposition)

Observable - Real-Life Example

- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable**



Example - State variable

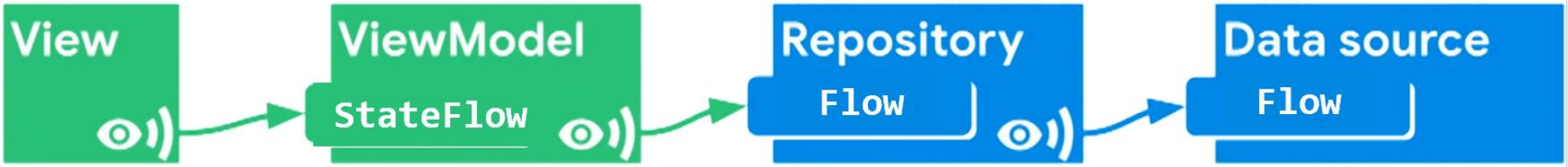
```
class ScoreViewModel : ViewModel() {
    // Private mutable state variables
    private var _team1Score = mutableStateOf(0)
    // Public State variables
    val team1Score : State<Int> = _team1Score
    fun onIncrementTeam1Score() { _team1Score.value++  }
```

```
@Composable
fun ScoreScreen() {
    // Get an instance of the ScoreViewModel
     val scoreViewModel = viewModel<ScoreViewModel>()
    Text(text = scoreViewModel.team1Score.value)
    Button(onClick = { scoreViewModel.onIncrementTeam1Score() }) {
       Text(text = "+1")
    }
    ...
}
```

Other Observable Types

- `mutableStateListOf()` creates an instance of MutableList that is observable
- `mutableStateMapOf()` creates an instance of MutableMap<K, V> that is observable
- **Flow**: a flow is a type that can emit a stream of values overtime
  - e.g., you can use a flow to receive live updates from a database
- **StateFlow** is lifecycle-aware observable data holder class that notifies the View when the model data changes
  - Meaning that StateFlow only sends updates to app component observers that are in an active lifecycle state

Flow

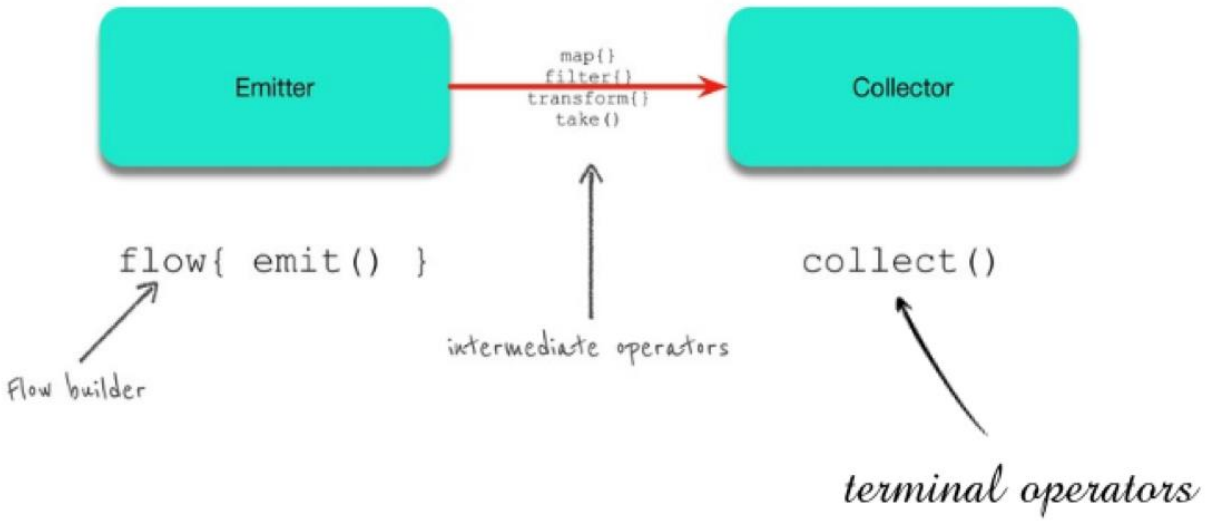


Some of the slides are based on
[Kotlin Flows in practice YouTube Video]

Back

What is Flow?

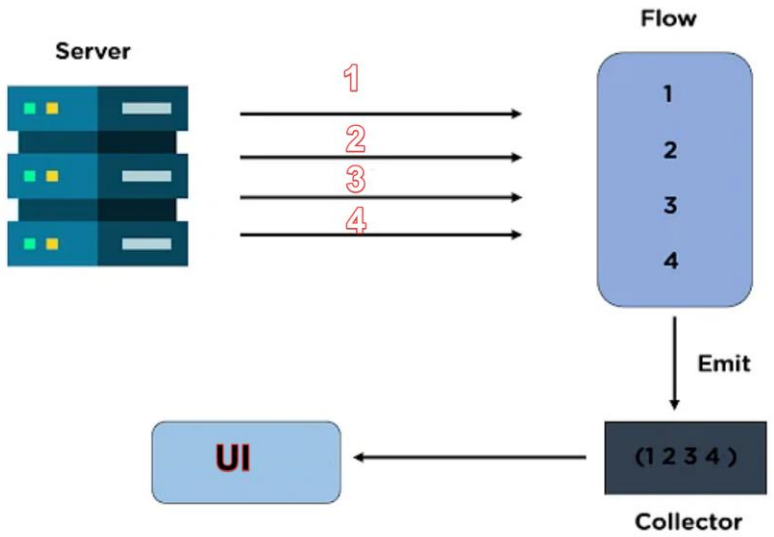
- Stream of values produced one by one over time instead of all at once
  - as opposed to functions that return only a single value
  - Values could be generated **asynchronously** from network requests or database calls
- Flow as an **Emitter** and **Collector**



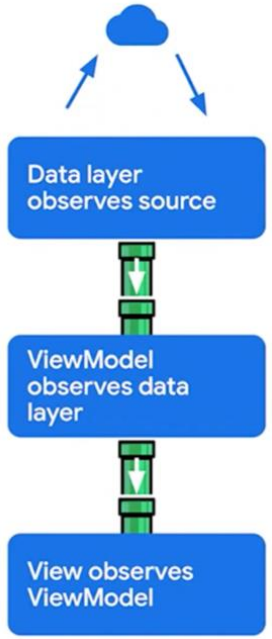
Flows are **cold** i.e., values are produced after collect is called

Flow motivation

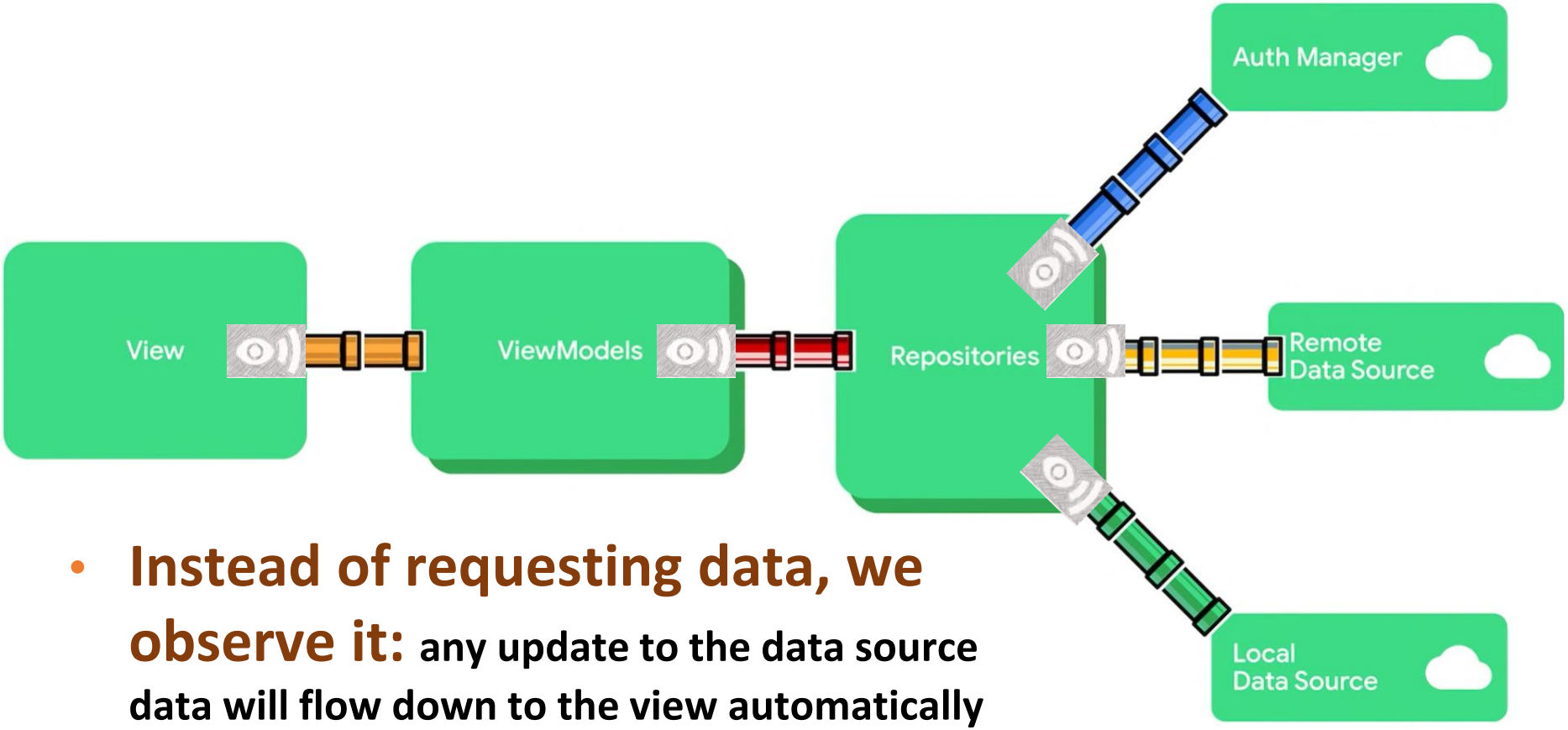
- Flow is an asynchronous stream of data
  - It returns multiple asynchronously computed values
  - e.g., you can use a flow to receive live updates from a database then use it to update the UI



- Makes the app responsive as the data is received from the server asynchronously in the background using as Flow to avoid blocking specially when the data fetching is long running



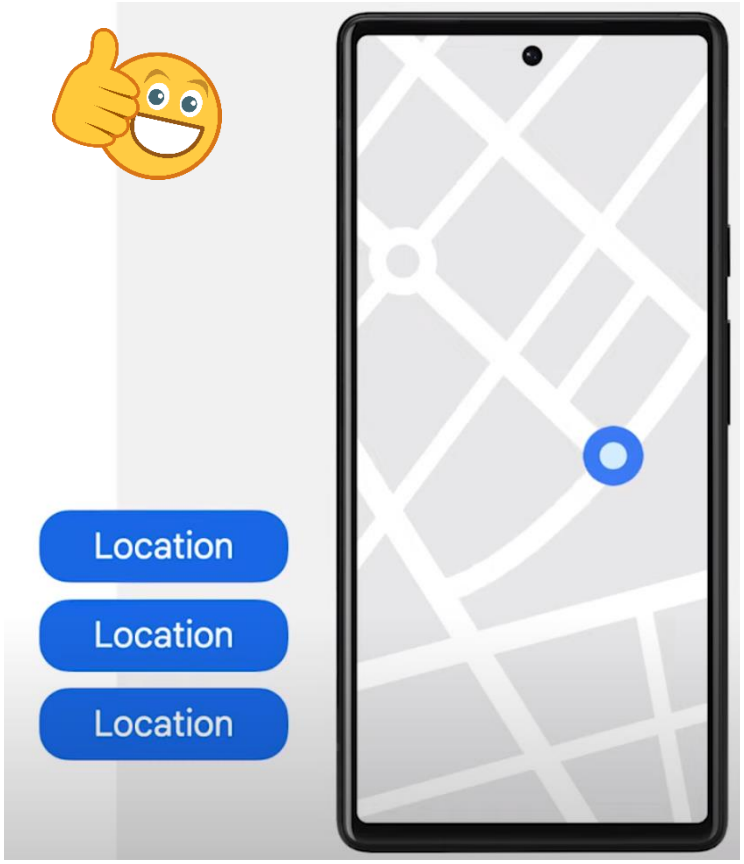
Flows are used to keep the View in synch with the data sources



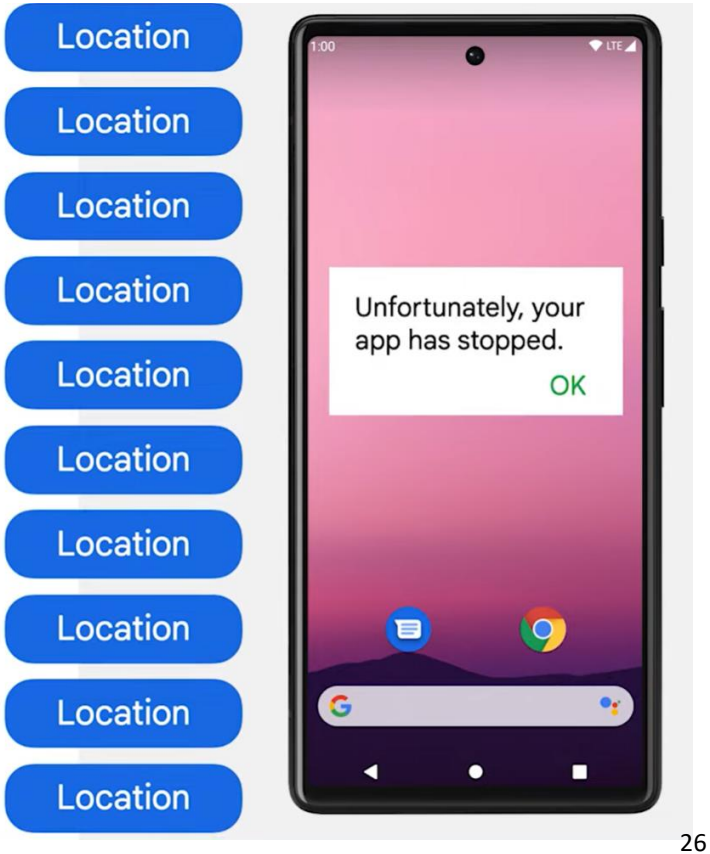
- **Instead of requesting data, we observe it:** **any update to the data source data will flow down to the view automatically**
- Repo observes data changes from data sources
- ViewModel observes data changes from the Repo
- View observes data changes from the ViewModel

Non lifecycle aware flow collection

- Non lifecycle aware flow collection means keep collecting the flow updates even if the UI is NOT visible
  - Waste resources such as battery and network bandwidth
  - May cause application crash as updating the UI in the background may throw an exception or show dialogue

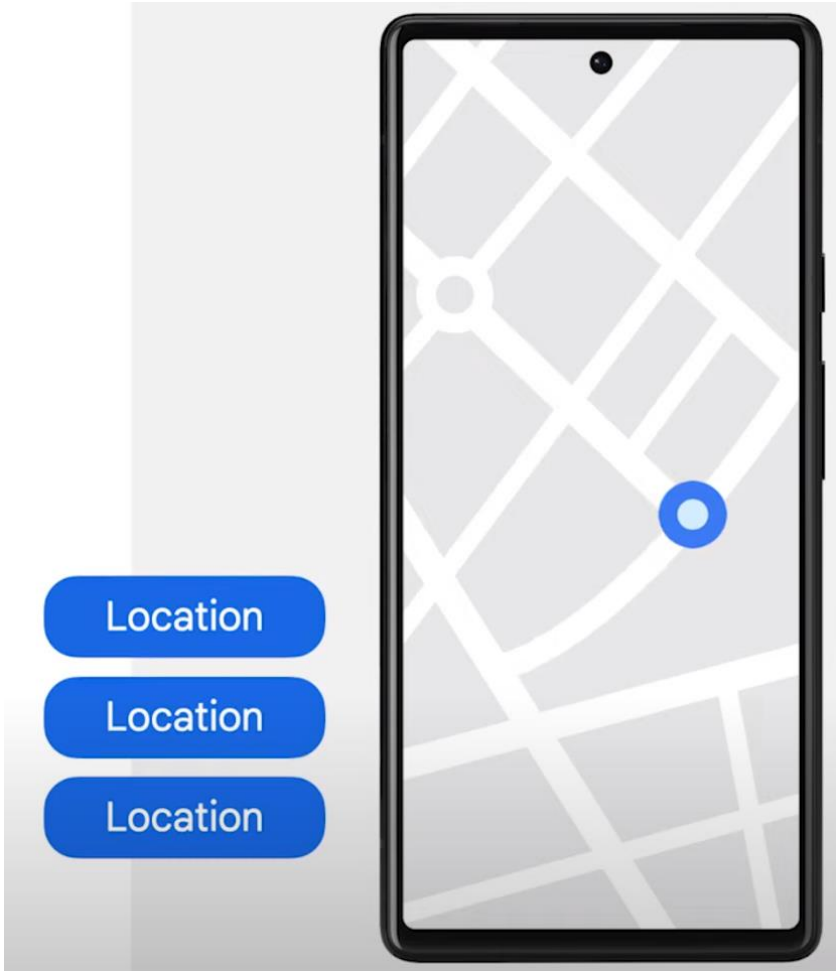


Analogy: should close the tap while brushing teeth or going for a nap!



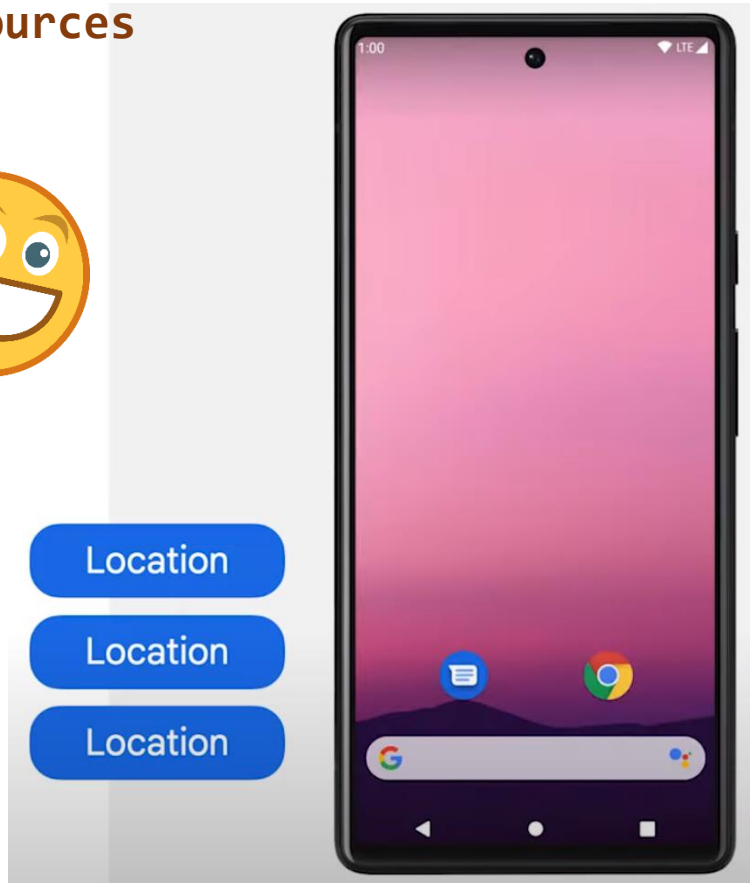
Lifecycle aware flow collection

Collect the data from the flow ONLY when the UI is visible



Stop collecting the flow updates when the UI is NOT visible:

No more location updates are received when the app is in the background to save system resources



In the View use collectAsStateWithLifecycle

- `collectAsStateWithLifecycle()` collects values from a Flow and stores the latest value into a Compose State variable
  - Causing recomposition when a new value is received from the Flow
- It does so in a lifecycle-aware manner, allowing the app to save unneeded app resources when the app is in the background
  - collectAsStateWithLifecycle start collecting values from the flow only when the UI is visible and stop collecting values from the flow when the UI is not visible on the screen
- Requires adding this dependency to build.gradle

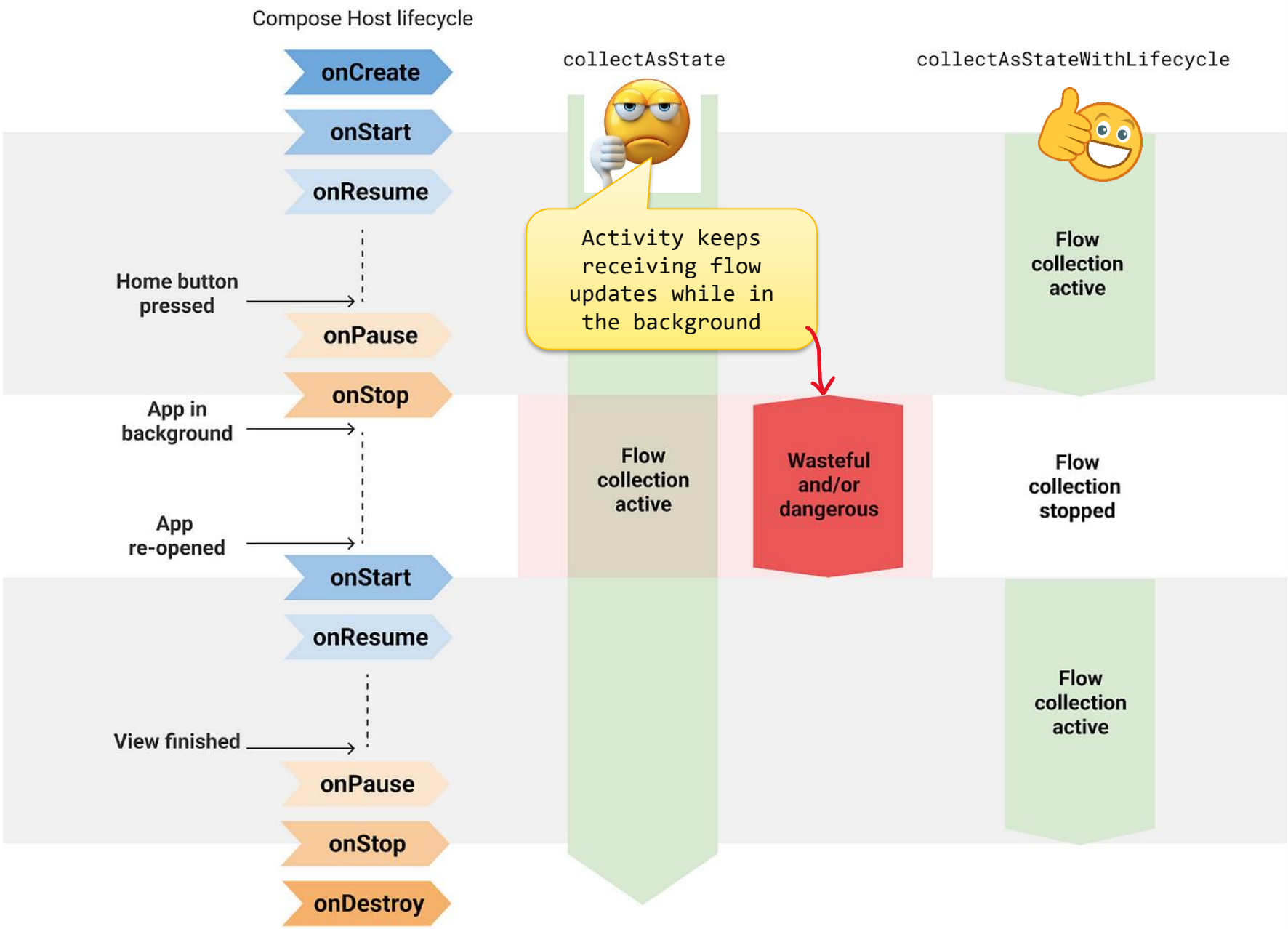
```
implementation("androidx.lifecycle:lifecycle-runtime-compose:@version")
```


```
object WeatherRepository {
    private val weatherConditions = listOf("Sunny", "Windy", "Rainy", "Snowy")
    fun getWeather(): Flow<String> =
        flow {
              while (true) {
                  delay(3000)
                  emit( weatherConditions.shuffled().first() )
               }
         }
}
```

```
class WeatherViewModel : ViewModel() {
    val weatherFlow: Flow<String> = WeatherRepository.getWeather()
}
```

```
@Composable
fun ScoreScreen() {
    val weatherViewModel = viewModel<WeatherViewModel>()
    val weatherFlow = weatherViewModel.weatherFlow.
        collectAsStateWithLifecycle (initialValue = "")
    // Recomposes whenever weatherFlow changes
    Text(
      text = "Weather: ${weatherFlow.value}" )
    }  ... }
```

In the View, collect Flow using **collectAsStateWithLifecycle**



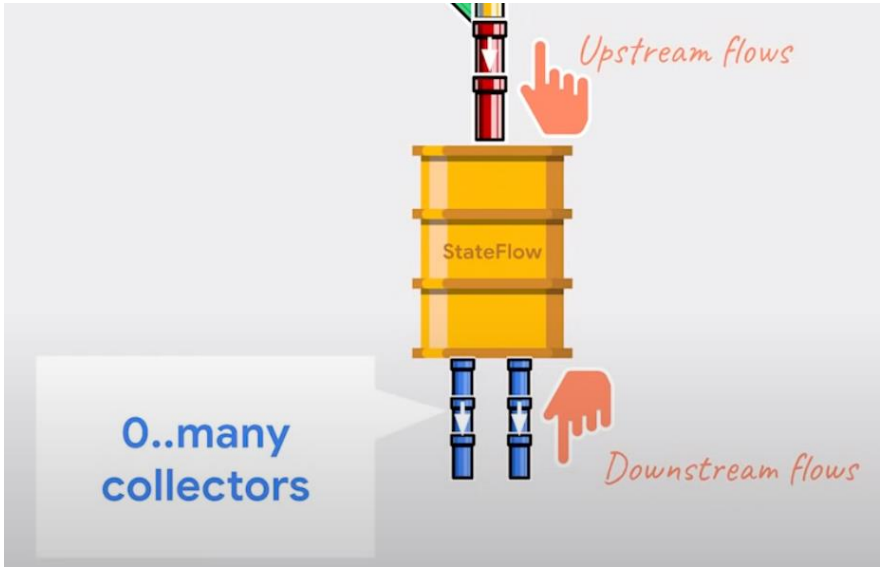
In the ViewModel use `stateIn`

- ViewModel can collect values from a Flow returned by the model and convert it into a StateFlow using **stateIn** intermediate operator
- The started parameter is used to specify the strategy that controls when the flow collection is started and stopped
- ***WhileSubscribed*** allows you to configure a delay in milliseconds between the time the last UI collector disappears and the time we stop the upstream flows
  - E.g., You can use **WhileSubscribed(5000)** to keep the upstream flow active for 5 seconds more after the disappearance of the last collector. That avoids restarting the upstream flow after a configuration change
  - After 5 seconds if the app remains in the background, Flow collection is stopped to save battery and other resources. Only when the user opens the app again, the upstream flows are automatically restarted

```
val newsFlow: StateFlow<String> = DataRepository.getNews().stateIn(
    scope = viewModelScope, initialValue = "",
    started = WhileSubscribed(5000)
)
```

StateFlow

- Flows are **cold** by default. This means that each time we collect the Flow (aka subscribe), the code in its builder will get executed
  - This is something that you might not want to do when the app goes through a configuration change. To solve this, StateFlow can be used



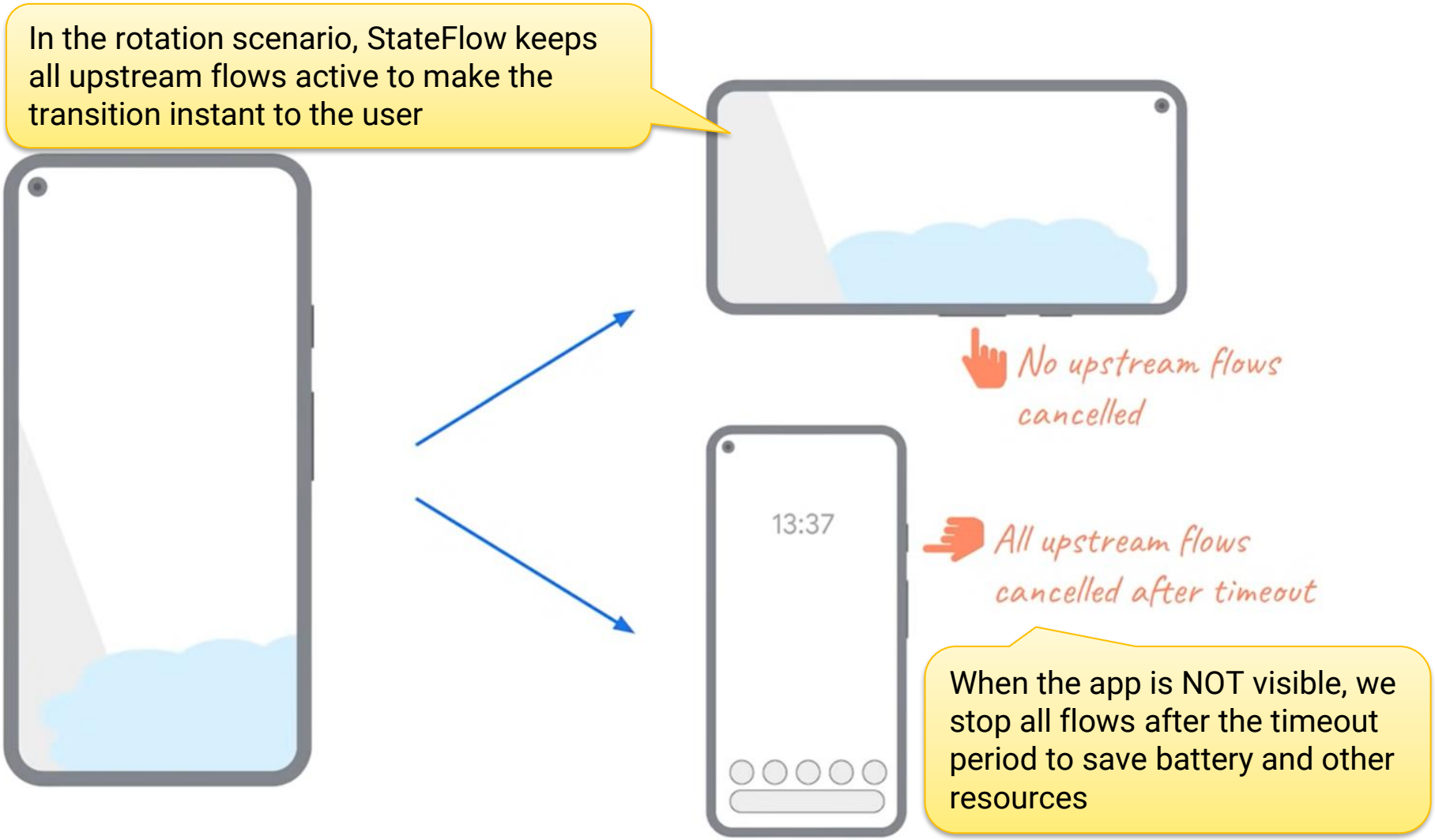
- ***StateFlow*** is a **hot** observable state-holder that receives updates from the upstream flow and stores the latest value
  - A StateFlow is hot: collecting from the flow doesn't trigger any producer code
  - A StateFlow remains active and in memory during a configuration change (i.e., survives a config change)
- Multiple screens can get its current value or subscribe to it to receive state updates
  - When a new consumer starts collecting from the state flow, it receives the last state and any subsequent state updates

StateFlow

StateFlow stores the latest flow value and makes it available to the UI without needing to re-collect from the upstream flow after a figuration change

- You can convert any flow to a ***StateFlow*** using the ***stateIn*** operator
- It holds data even if there are no collectors
- You can collect multiple times from it without triggering any producer code
- Can be shared by multiple screens

.stateIn WhileSubscribed(5000)



- When a StateFlow stops being collected, we don't immediately stop all the upstream flows. Instead, we wait for some time (5 seconds).
- If the flow is collected again before the timeout, no upstream flows are canceled.

stateIn with SharingStarted.Lazily

- In same cases you may want to keep collecting flow updates even when the UI is NOT visible
  - Use `started = SharingStarted.Lazily` to do so
- The example below keep collecting time remaining updates even when the app is in the background

```
val timeRemainingFlow: StateFlow<String> =
    DataRepository.countDownTimer(5)
        .stateIn(
            scope = viewModelScope,
            // Sharing is started when the first subscriber appears and never stops.
            started = SharingStarted.Lazily,
            initialValue = ""
        )
```

Flow Operators

- Flow has operators similar to collections such as map, filter and reduce that can be used to transform the flow

```
(1..5).asFlow()
      .filter { it % 2 == 0 }
      .map { it * it }
      .collect { println(it.toString()) }

val result =(1..5).asFlow()
                  .reduce { a, b -> a + b }
println("result: $result")
```

Using .map to produced a derived flow

- This example produces a new flow that converts the incoming Fahrenheit temperature to Celsius temperature
  - This gets auto executed whenever a flow update is received

```
val weatherCelsiusFlow = weatherFlow.map {
    it?.let { weather ->
        Weather(
            weather.condition,
            (weather.temperature - 32) * 5 / 9,
            "Celsius",
        )
    } }
```

Resources

- State and Jetpack Compose
  - https://developer.android.com/jetpack/compose/state
- Kotlin flows on Android
  - Kotlin Flows in practice YouTube Video
  - https://developer.android.com/kotlin/flow
  - https://developer.android.com/kotlin/flow/stateflow-and-sharedflow
- MVVM
  - https://developer.android.com/topic/architecture