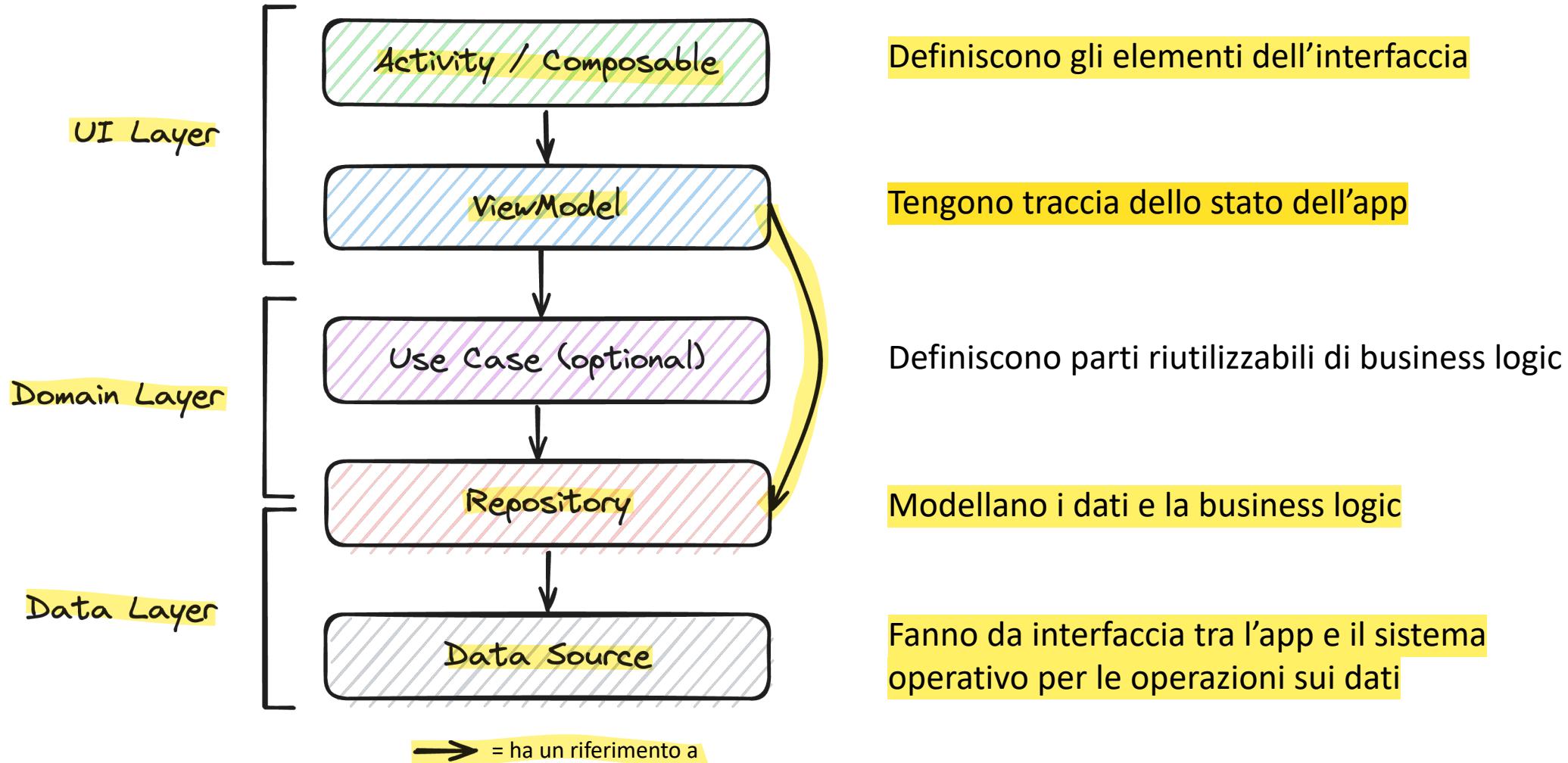


# **ViewModel , DataStore, Dependency Injection**

Architettura e stato delle app Compose

**Laboratorio 3**

# Architettura di un'app Compose



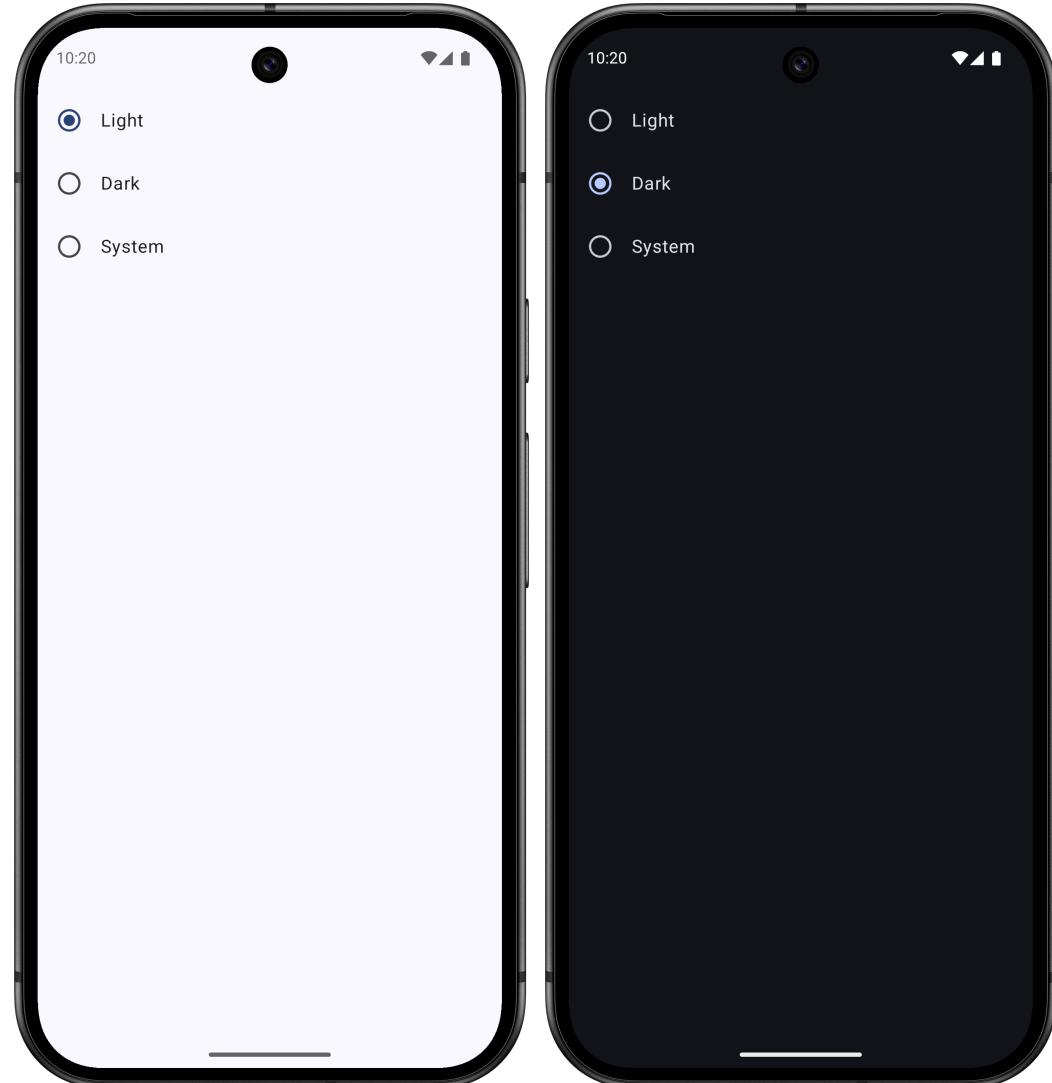
# Laboratorio di oggi

---

- Esercitazione guidata
  1. Creazione di una semplice app per cambiare il tema tra chiaro, scuro e automatico
  2. Aggiunta all'app di un ViewModel per gestire lo stato dell'interfaccia
  3. Salvataggio dell'opzione selezionata tramite DataStore
    - Con dependency injection
- Esercitazione libera
  4. Aggiunta all'app TravelDiary di ViewModel, DataStore e dependency injection

# 1. App con toggle per il tema

- Creiamo una semplice applicazione che permetta, tramite dei radio button, di selezionare il tema tra chiaro, scuro e di sistema (che segue cioè il tema del dispositivo)
- Step
  1. Enum per rappresentare le tre opzioni
  2. Variabile per memorizzare il tema selezionato
  3. Radio buttons
  4. Applicazione del tema
  5. Fix vari



# 1.1. Enum per rappresentare le tre opzioni

---

- Possiamo utilizzare un enum per definire i tre temi selezionabili

```
enum class Theme { Light, Dark, System }
```

## 1.2. Variabile per memorizzare il tema selezionato

---

```
var selectedTheme by remember { mutableStateOf(Theme.System) }
```

- Tramite **remember** l'app non perde traccia del tema in seguito alle recomposition
- Tramite **mutableStateOf** l'app fa scattare una recomposition al cambiare del valore di **selectedTheme**
- Scegliamo come tema di default quello di sistema

# 1.3. Radio buttons

```
Column(Modifier.selectableGroup()) {
    Theme.entries.forEach { theme ->
        Row(
            Modifier
                .fillMaxWidth()
                .height(56.dp)
                .selectable(
                    selected = (theme == selectedTheme),
                    onClick = { selectedTheme = theme },
                    role = Role.RadioButton
                )
                .padding(horizontal = 16.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            RadioButton(
                selected = (theme == selectedTheme),
                onClick = null
            )
            Text(
                text = theme.toString(),
                style = MaterialTheme.typography.bodyLarge,
                modifier = Modifier.padding(start = 16.dp)
            )
        }
    }
}
```

- Codice dalla [documentazione ufficiale](#) e leggermente modificato

# 1.3. Radio buttons

```
Column(Modifier.selectableGroup()) {  
    Theme.entries.forEach { theme ->  
        Row(  
            Modifier  
                .fillMaxWidth()  
                .height(56.dp)  
                .selectable(  
                    selected = (theme == selectedTheme),  
                    onClick = { selectedTheme = theme },  
                    role = Role.RadioButton  
                )  
                .padding(horizontal = 16.dp),  
            verticalAlignment = Alignment.CenterVertically  
        ) {  
            RadioButton(  
                selected = (theme == selectedTheme),  
                onClick = null  
            )  
            Text(  
                text = theme.toString(),  
                style = MaterialTheme.typography.bodyLarge,  
                modifier = Modifier.padding(start = 16.dp)  
            )  
        }  
    }  
}
```

- Codice dalla [documentazione ufficiale](#) e leggermente modificato

Creiamo un radio button per ogni elemento dell'enum

Definiamo quando è selezionato

Definiamo cosa succede al click

onClick = null è consigliato dalla documentazione per motivi di accessibilità

## 1.4. Applicazione del tema

- Nella **MainActivity**, impostiamo il parametro **darkTheme** del composable che definisce il tema

```
ThemeToggleTheme(  
    darkTheme = when (selectedTheme) {  
        Theme.Light -> false  
        Theme.Dark -> true  
        Theme.System -> isSystemInDarkTheme()  
    }  
) {  
    //...
```

## 1.5. Fix per i cambi di configurazione

- C'è un problema: in seguito a un cambio di configurazione, il tema dell'app viene resettato a quello di sistema
  - Ad esempio se ruotiamo il dispositivo
- Questo perché il valore memorizzato da **remember** sopravvive alle recomposition, ma non ai cambi di configurazione
- Utilizzando **rememberSaveable** possiamo gestire anche questa casistica

```
var selectedTheme by rememberSaveable { mutableStateOf(Theme.System) }
```

# 1.5. Fix per il colore della status bar

- Abbiamo un altro problema: il contenuto della status bar diventa bianco se impostiamo il tema scuro a livello di sistema, mentre rimane nero se lo facciamo tramite l'app
- Soluzione: impostare il colore manualmente in seguito ad ogni recomposition

ui.theme.Theme.kt - ThemeToggleTheme

```
val view = LocalView.current
if (!view.isInEditMode) {
    SideEffect {
        val window = (view.context as Activity).window
        window.statusBarColor = colorScheme.primaryContainer.toArgb()
        WindowCompat
            .getInsetsController(window, view)
            .isAppearanceLightStatusBars = !darkTheme
    }
}
```



# 1.5. Fix per il colore della status bar

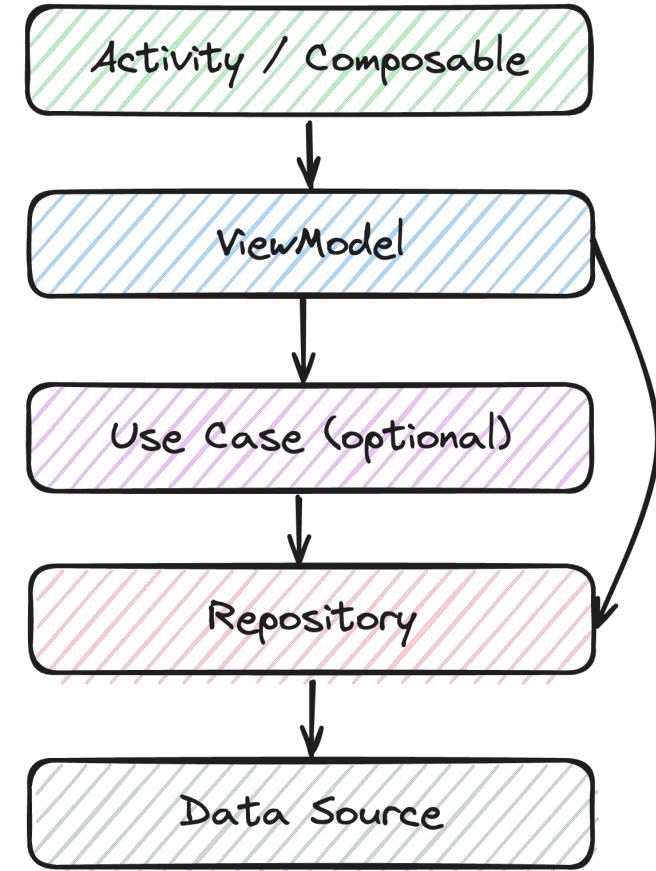
---

ui.theme.Theme.kt - ThemeToggleTheme

```
val view = LocalView.current
// Salta l'operazione se stiamo visualizzando il componente tramite dei devtools (es. preview)
if (!view.isInEditMode) {
    SideEffect { // Esegue il blocco al termine di ogni recomposition
        val window = (view.context as Activity).window
        // Cambio del colore della status bar per Android <= 14
        window.statusBarColor = colorScheme.primaryContainer.toArgb()
        // Cambio del colore della status bar per Android 15+
        WindowCompat
            .getInsetsController(window, view)
            .isAppearanceLightStatusBars = !darkTheme
    }
}
```

## 2. Aggiunta ViewModel

- L'app che abbiamo creato è semplice e funzionale
- Tuttavia, con l'implementazione di nuove feature sarà sempre più complicata da manutenere e testare
  - Interfaccia, stato e dati coesistono nello stesso file, all'interno delle stesse funzioni
- **Soluzione:** definiamo una struttura con una migliore *separation of concerns*
  - Il primo passaggio è l'aggiunta di un **ViewModel**



# Recap: ViewModel

- Incapsula ed espone lo stato dell’interfaccia utente di una parte dell’app (es. un’Activity, uno screen, ...)
- Espone metodi per modificare lo stato
- Sopravvive alle recomposition e ai cambi di configurazione del dispositivo (es. rotazione)
- Il suo ciclo di vita è legato automaticamente a quello del suo owner (es. un’Activity, una parte del grafo di navigazione, ...)

Poiché il ViewModel è consapevole del ciclo di vita delle Activity, evita problemi come perdite di memoria e aggiornamenti alla UI quando questa non è attiva.

# Recap: StateFlow

StateFlow è un tipo speciale di observable (un "hot stream") che tiene sempre in memoria un unico valore accessibile tramite la proprietà `value`. Ogni volta che modifichi questo valore, l'aggiornamento viene immediatamente propagato a tutti i subscriber. Questo comportamento lo rende particolarmente adatto per tracciare i cambiamenti di stato all'interno di un ViewModel, dove è fondamentale mantenere un'unica fonte di verità e garantire che la UI rifletta sempre lo stato corrente.

- È un observable con una singola proprietà **value**, che può essere modificata facendo scattare un'aggiornamento a tutti i subscriber
  - StateFlow espone una proprietà `value` che contiene l'ultimo stato emesso. Quando cambi il suo contenuto (ad esempio, con un'assegnazione diretta o tramite operatori specifici), l'aggiornamento viene automaticamente notificato a tutte le parti dell'app che lo stanno osservando.
- È uno dei metodi più indicati per tracciare i cambiamenti di stato all'interno di un ViewModel
  - Utilizzare StateFlow nel ViewModel consente di gestire in maniera reattiva lo stato della UI. Qualsiasi modifica nello stato, come ad esempio un cambiamento nei dati o nelle impostazioni, viene propagata immediatamente alla UI, garantendo aggiornamenti coerenti e sincronizzati anche in presenza di cambi di configurazione (come la rotazione dello schermo).
- Fornisce vari operatori per l'aggiornamento di **value**, tutti atomici e utilizzabili in contesti concorrenti e multi-thread
  - StateFlow mette a disposizione operatori come `update`, `updateAndGet` e `getAndUpdate`, che eseguono le modifiche in maniera atomica. Questi operatori usano internamente meccanismi come il `compareAndSet` per assicurarsi che, anche se più thread tentano di aggiornare lo stato contemporaneamente, ogni modifica avvenga in modo sicuro e senza condizioni di race.

## 2. Aggiunta ViewModel

---

- Step
  1. Installazione dipendenze
  2. Modifica della struttura del progetto
  3. Creazione classe ViewModel
  4. Modifica della schermata dell'app
  5. Creazione istanza ViewModel

## 2.1. Installazione dipendenze

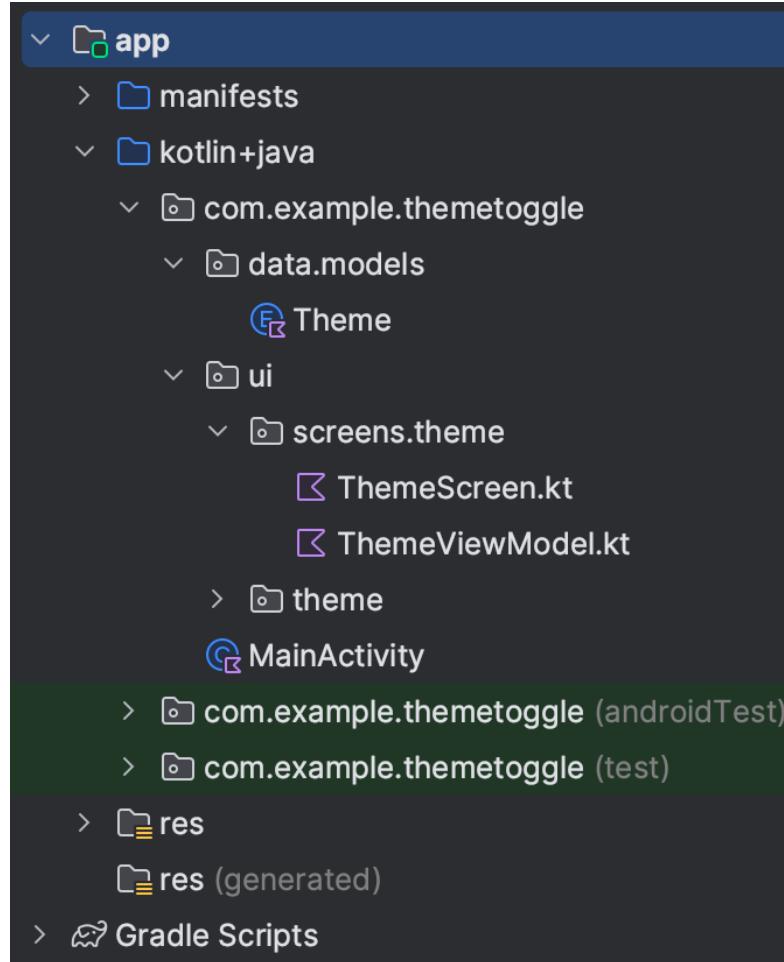
---

- Aggiungere le seguenti dipendenze al file **build.gradle.kts** (**modulo :app**)

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7")
implementation("androidx.lifecycle:lifecycle-runtime-compose:2.8.7")
```

Replace with new library catalog  
declaration, reusing version  
variable lifecycleRuntimeKtx

## 2.2. Struttura progetto



## 2.3. Creazione classe ViewModel

All'interno del ViewModel viene creato un `MutableStateFlow (_state)` che contiene il valore iniziale (qui impostato su `ThemeState(Theme.System)`). In questo modo lo stato può essere aggiornato internamente in maniera reattiva e sicura. Successivamente, viene esposto un `StateFlow` (tramite `asStateFlow()`) che è in sola lettura, garantendo che altre classi (ad esempio la UI) possano osservare lo stato senza poterlo modificare direttamente.

ThemeViewModel.kt

```
data class ThemeState(val theme: Theme)

class ThemeViewModel : ViewModel() {
    private val _state =
        MutableStateFlow(ThemeState(Theme.System))
    val state = _state.asStateFlow()

    fun changeTheme(theme: Theme) {
        _state.value = ThemeState(theme)
    }
}
```

Il metodo `changeTheme` consente di aggiornare lo stato del tema creando un nuovo oggetto `ThemeState` con il tema specificato. Questo approccio immutabile (ogni modifica genera una nuova istanza di `ThemeState`) è in linea con i principi della programmazione reattiva e consente una gestione più prevedibile delle modifiche.

Definire una data class (qui `ThemeState`) permette di raggruppare in modo chiaro e organizzato tutte le informazioni relative allo stato del ViewModel. Questo facilita la gestione, il test e l'estensione dello stato nel tempo.

• È convenzione raggruppare lo stato del ViewModel in una classe State

• Utilizziamo un **MutableStateFlow** come backing property per lo stato, così che possa essere modificato internamente al ViewModel, mentre esponiamo uno **StateFlow** non modificabile esternamente

• Esponiamo un metodo **changeTheme** per cambiare tema

## 2.4. Schermata dell'app

ThemeScreen.kt

```
@Composable
fun ThemeScreen(state: ThemeState, onThemeSelected: (Theme) -> Unit) {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
        Column(Modifier.padding(innerPadding).selectableGroup()) {
            Theme.entries.forEach { theme ->
                Row(
                    Modifier
                        .fillMaxWidth()
                        .height(56.dp)
                        .selectable(
                            selected = (theme == state.theme),
                            onClick = { onThemeSelected(theme) },
                            role = Role.RadioButton
                        )
                        .padding(horizontal = 16.dp),
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    RadioButton(selected = (theme == state.theme), onClick = null)
                    Text(
                        text = theme.toString(),
                        style = MaterialTheme.typography.bodyLarge,
                        modifier = Modifier.padding(start = 16.dp)
                    )
                }
            }
        }
    }
}
```

## 2.4. Schermata dell'app

ThemeScreen.kt

```
@Composable
fun ThemeScreen(state: ThemeState, onThemeSelected: (Theme) -> Unit) {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
        Column(Modifier.padding(innerPadding).selectableGroup()) {
            Theme.entries.forEach { theme ->
                Row(
                    Modifier
                        .fillMaxWidth()
                        .height(56.dp)
                        .selectable(
                            selected = (theme == state.theme),
                            onClick = { onThemeSelected(theme) },
                            role = Role.RadioButton
                        )
                        .padding(horizontal = 16.dp),
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    RadioButton(selected = (theme == state.theme), onClick = null)
                    Text(
                        text = theme.toString(),
                        style = MaterialTheme.typography.bodyLarge,
                        modifier = Modifier.padding(start = 16.dp)
                    )
                }
            }
        }
    }
}
```

Riceviamo come parametri:

- lo stato del ViewModel
- la funzione da richiamare alla modifica dello stato

## 2.5. Creazione istanza ViewModel

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            val themeViewModel = viewModel<ThemeViewModel>()
            val themeState by themeViewModel.state.collectAsStateWithLifecycle()

            ThemeToggleTheme(
                darkTheme = when (themeState.theme) {
                    Theme.Light -> false
                    Theme.Dark -> true
                    Theme.System -> isSystemInDarkTheme()
                }
            ) {
                ThemeScreen(themeState, themeViewModel::changeTheme)
            }
        }
    }
}
```

## 2.5. Creazione istanza ViewModel

MainActivity.kt

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContent {  
            val themeViewModel = viewModel<ThemeViewModel>()  
            val themeState by themeViewModel.state.collectAsStateWithLifecycle()  
  
            ThemeToggleTheme(  
                darkTheme = when (themeState.theme) {  
                    Theme.Light -> false  
                    Theme.Dark -> true  
                    Theme.System -> isSystemInDarkTheme()  
                }  
            ) {  
                ThemeScreen(themeState, themeViewModel::changeTheme)  
            }  
        }  
    }  
}
```

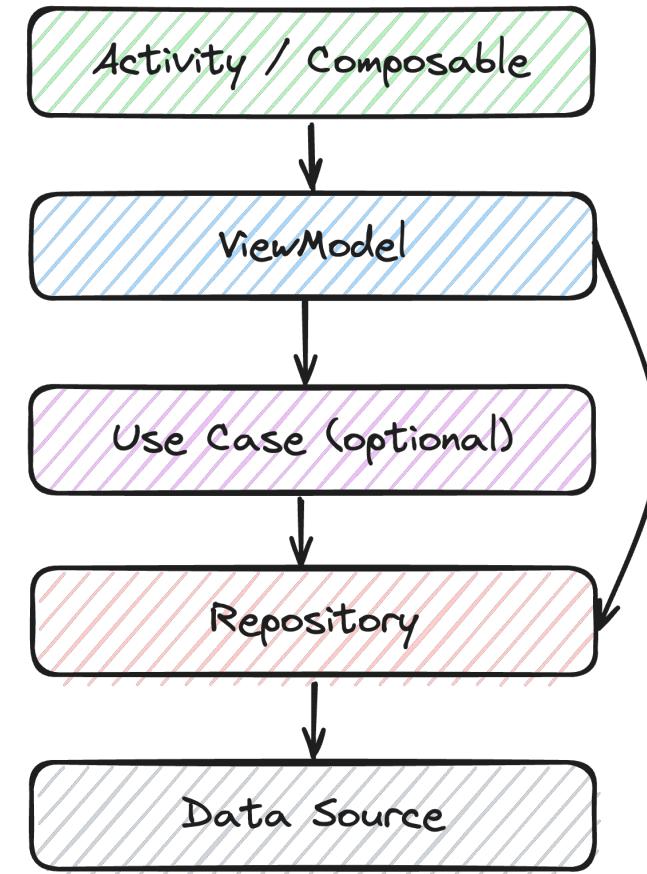
Creiamo l'istanza tramite l'helper `viewModel()`

Convertiamo lo `StateFlow` in `State`  
grazie a `collectAsStateWithLifecycle()`, convertiamo lo `StateFlow` esposto dal `ViewModel` in uno `State` di `Compose`, in modo che ogni aggiornamento dello stato provochi automaticamente la ricomposizione dell'interfaccia utente.

Passiamo lo state alla schermata, assieme alla funzione per modificare il tema

# 3. Aggiunta DataStore

- All'app che abbiamo creato manca una feature importante: il tema selezionato viene salvato solo in memory
  - Quindi va perso alla chiusura dell'app!
- Aggiungiamo un data layer tramite cui memorizzare il tema sul dispositivo
  - Tramite **DataStore**



# Recap: DataStore

---

- È una soluzione per lo storage di dati che permette di salvare coppie chiave-valore o oggetti tipizzati
- È perfettamente integrato nell'ecosistema di Compose, utilizzando i Flow e le coroutine di Kotlin
- Fornisce due API:
  - **Preferences DataStore**: per il salvataggio di coppie chiave-valore (lo utilizzeremo ora)
  - **Proto DataStore**: per memorizzare tipi di dati custom

# 3. Aggiunta DataStore

---

- Step - DataStore
  1. Installazione dipendenze per DataStore
  2. Creazione repository
  3. Aggiunta della repository al ViewModel
- Step - Dependency Injection
  4. Installazione dipendenze per dependency injection con Koin
  5. Creazione AppModule
  6. Avvio di Koin
  7. Creazione del ViewModel tramite Koin

## 3.1. Installazione dipendenze per DataStore

---

- Aggiungere la seguente dipendenza al file **build.gradle.kts** (**modulo :app**)

```
implementation("androidx.datastore:datastore-preferences:1.1.3")
```

Replace with new library  
catalog declaration

## 3.2. Creazione repository

---

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {
    companion object {
        private val THEME_KEY = stringPreferencesKey("theme")
    }

    val theme = dataStore.data
        .map { preferences ->
            try {
                Theme.valueOf(preferences[THEME_KEY] ?: "System")
            } catch (_: Exception) {
                Theme.System
            }
        }

    suspend fun setTheme(theme: Theme) =
        dataStore.edit { it[THEME_KEY] = theme.toString() }
}
```

## 3.2. Creazione repository

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {  
    companion object {  
        private val THEME_KEY = stringPreferencesKey("theme")  
        } Questa chiave serve per identificare il valore  
        del tema all'interno delle Preferences.  
  
        val theme = dataStore.data  
            .map { preferences ->  
                try {  
                    Si cerca di leggere il valore associato a THEME_KEY.  
                    Theme.valueOf(preferences[THEME_KEY] ?: "System")  
                } catch (_: Exception) {  
                    Theme.System  
                }  
            }  
  
        suspend fun setTheme(theme: Theme) =  
            dataStore.edit { it[THEME_KEY] = theme.toString() }  
    }
```

In questo modo il repository non si preoccupa  
di come venga creato il DataStore, ma si  
limita a utilizzarlo per leggere e scrivere i dati.

Accettiamo il DataStore  
come parametro

Sottoscriviamo al Flow  
del DataStore

Convertiamo la stringa del  
tema in enum

Se la conversione produce  
errori (es. la stringa non  
corrisponde a un valore  
dell'enum) utilizziamo il  
tema di sistema

## 3.2. Creazione repository

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {  
    companion object {  
        private val THEME_KEY = stringPreferencesKey("theme")  
    }  
  
    val theme = dataStore.data  
        .map { preferences ->  
            try {  
                Theme.valueOf(preferences[THEME_KEY] ?: "System")  
            } catch (_: Exception) {  
                Theme.System  
            }  
        }  
  
    suspend fun setTheme(theme: Theme) = ←  
        dataStore.edit { it[THEME_KEY] = theme.toString() }  
}
```

Definiamo una suspend function per modificare il valore nello store da una coroutine

## 3.3. Aggiunta repository al ViewModel

- Modifichiamo il ViewModel per leggere e scrivere i dati dalla repository

```
data class ThemeState(val theme: Theme)

class ThemeViewModel(
    private val repository: ThemeRepository
) : ViewModel() {
    val state = repository.theme.map { ThemeState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = ThemeState(Theme.System)
    )

    fun changeTheme(theme: Theme) = viewModelScope.launch {
        repository.setTheme(theme)
    }
}
```

### 3.3. Aggiunta repository al ViewModel

started: Usiamo `SharingStarted.WhileSubscribed()`, che attiva il collecting quando almeno un osservatore è presente e sospende l'operazione quando non ci sono subscriber, ottimizzando l'uso delle risorse.

initialValue: Forniamo un valore iniziale (`ThemeState(Theme.System)`) da utilizzare finché non arrivano gli aggiornamenti dal repository.

- Modifichiamo il ViewModel per leggere e scrivere i dati dalla repository

```
data class ThemeState(val theme: Theme)

class ThemeViewModel(
    private val repository: ThemeRepository
) : ViewModel() {
    val state = repository.theme.map { ThemeState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = ThemeState(Theme.System)
    )

    fun changeTheme(theme: Theme) = viewModelScope.launch {
        repository.setTheme(theme)
    }
}
```

Il Flow `repository.theme` emette valori di tipo `Theme`. Con l'operatore `map` trasformiamo ciascun valore in un oggetto `ThemeState`. Così la UI non osserva direttamente l'enum, ma uno stato completo.

Accettiamo la repository come parametro

Con questo operatore convertiamo il Flow in un StateFlow

Convertiamo il Flow del tema in uno StateFlow di ThemeState

Utilizziamo una coroutine nello scope del ViewModel per modificare il tema nella repository

# Dependency injection

---

- **Problema:** come passiamo i parametri necessari ai costruttori di ViewModel (che necessita la repository) e repository (a cui serve il DataStore)?
- Potremmo farlo manualmente, ma:
  - Le istanze passate sarebbero dei singleton?
  - Sarebbe semplice fare refactoring del codice?
  - Sarebbe semplice testare separatamente i vari componenti?
  - ...
- **Soluzione: dependency injection**

# Dependency injection

---

- La **dependency injection (DI)** è una tecnica molto utilizzata che permette a un sistema di avere una buona architettura, fornendo i seguenti vantaggi:
  - Riusabilità del codice
  - Facilità di refactoring
  - Facilità di testing

# Gestione delle dipendenze

---

- Le classi spesso richiedono riferimenti ad altre classi. Per esempio, una classe **Car** potrebbe aver bisogno di un riferimento a una classe **Engine**. Queste classi necessarie sono chiamate *dipendenze* e, in questo esempio, la classe **Car** dipende dalla presenza di un'istanza della classe **Engine** per funzionare
- Una classe può ottenere un oggetto di cui ha bisogno in tre modi:
  - a. La classe stessa crea un'istanza della dipendenza di cui ha bisogno
  - b. La classe ottiene un riferimento alla dipendenza da qualche altra parte, tramite API come **Context** e **getSystemService**
  - c. La dipendenza viene fornita come parametro. L'applicazione può passare queste dipendenze quando la classe viene istanziata o passarle ai metodi della classe che ne hanno bisogno

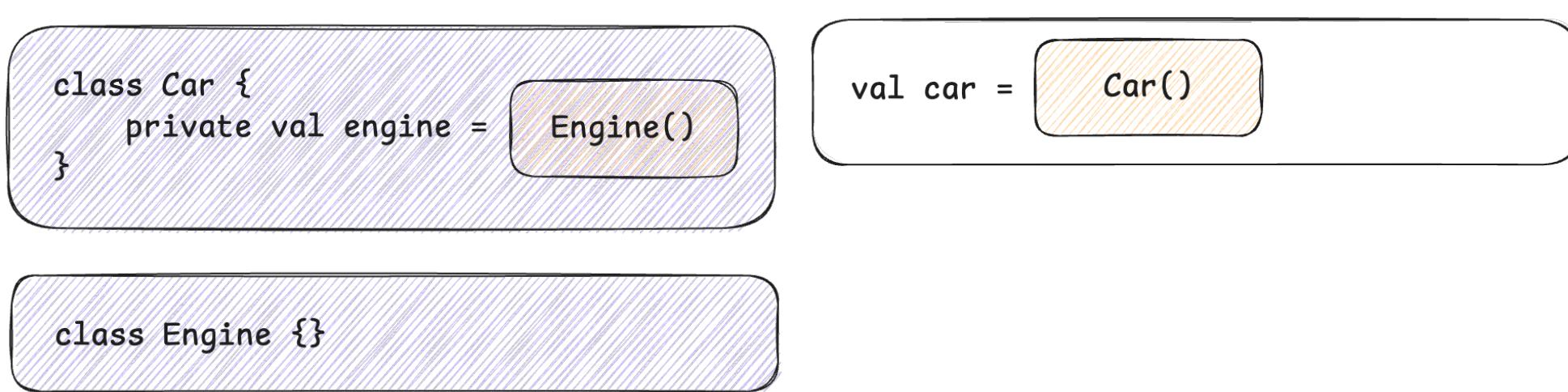
# Gestione delle dipendenze

- Le classi spesso richiedono riferimenti ad altre classi. Per esempio, una classe **Car** potrebbe aver bisogno di un riferimento a una classe **Engine**. Queste classi necessarie sono chiamate *dipendenze* e, in questo esempio, la classe **Car** dipende dalla presenza di un'istanza della classe **Engine** per funzionare
- Una classe può ottenere un oggetto di cui ha bisogno in tre modi:
  - a. La classe stessa crea un'istanza della dipendenza di cui ha bisogno
  - b. La classe ottiene un riferimento alla dipendenza da qualche altra parte, tramite API come **Context** e **getSystemService**
  - c. La dipendenza viene fornita come parametro. L'applicazione può passare queste dipendenze quando la classe viene istanziata o passarle ai metodi della classe che ne hanno bisogno

Dependency injection

# Senza dependency injection

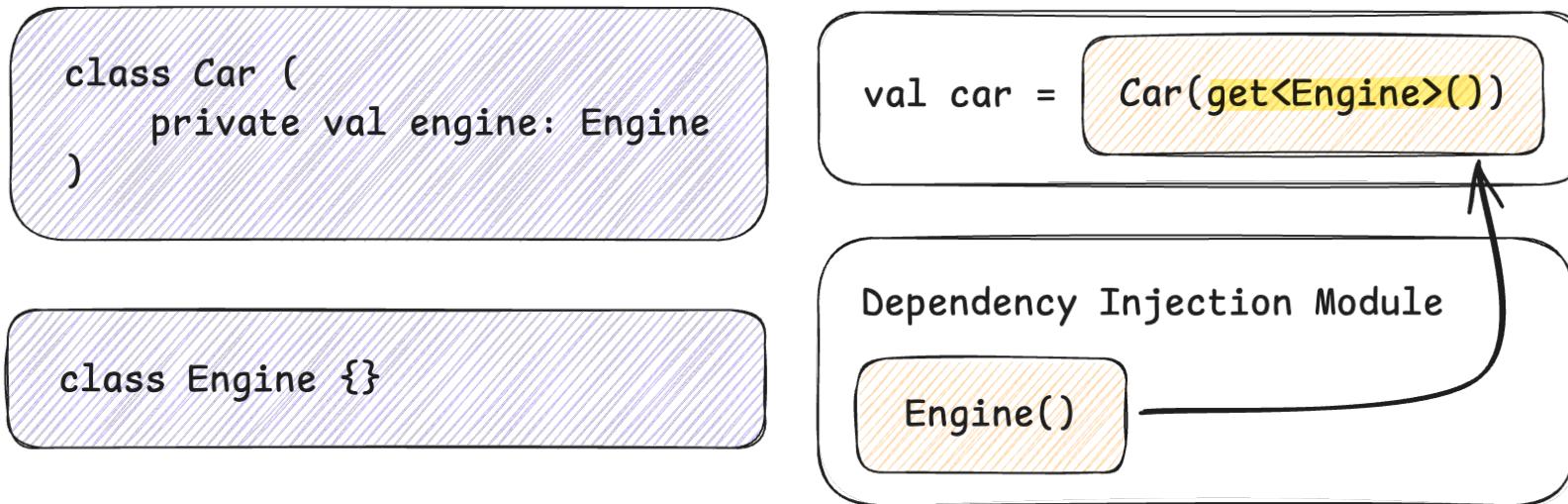
-  Class declaration
-  Object instantiation



- **Car** e **Engine** sono **tightly coupled**
- Non possiamo istanziare **Car** senza creare anche un oggetto **Engine**
- Non possiamo quindi testare **Car** separatamente da **Engine**

# Con dependency injection

- Class declaration
- Object instantiation



- **Car** e **Engine** possono essere istanziati e testati separatamente
- In più, tutte le dipendenze del programma sono raggruppate all'interno di un modulo di dependency injection, semplificandone la gestione all'aumentare delle dimensioni del sistema

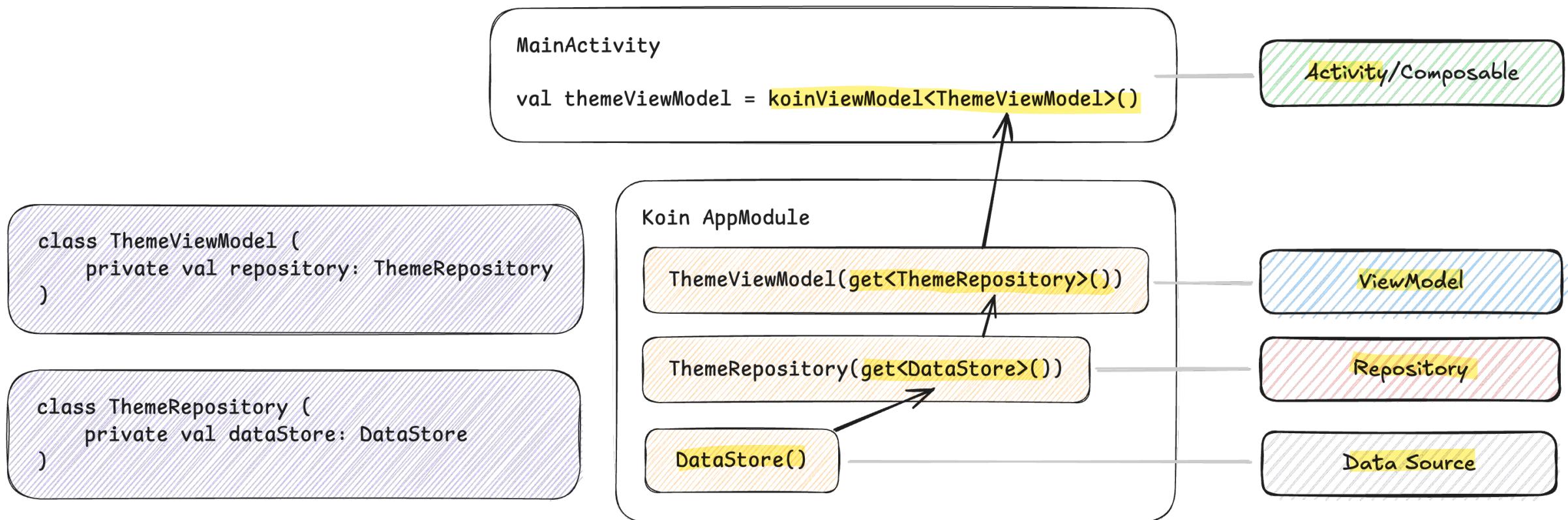
# Dependency injection con Koin

---

- **Koin** è una libreria open source, semplice e modulare per la gestione automatica della dependency injection
- Non è specifica per Android, può funzionare praticamente in qualsiasi progetto Kotlin
- Tramite Koin, è possibile organizzare le dipendenze della nostra app in **moduli**
- Ogni modulo può definire varie tipologie di oggetto. Nel nostro caso utilizzeremo:
  - **single** per la creazione di generici singleton
  - **viewModel** per istanziare appunto i singleton contenenti i ViewModel

# DI con Koin - Esercizio di oggi

- Class declaration
- Object instantiation



## 3.4. Installazione dipendenze per DI

---

- Aggiungere la seguente dipendenza al file **build.gradle.kts** (**modulo :app**)

```
implementation("io.insert-koin:koin-androidx-compose:4.0.2")
```

Replace with new library  
catalog declaration

## 3.5. Creazione AppModule

AppModule.kt

```
val Context.dataStore  
    by preferencesDataStore("theme")  
  
val appModule = module {  
    single { get<Context>().dataStore }  
  
    single { ThemeRepository(get()) }  
  
    viewModel { ThemeViewModel(get()) }  
}
```

- Il DataStore va creato come extension di **Context**, di conseguenza dobbiamo inizializzarlo top-level
- Con la funzione **get()** possiamo dire a Koin di andare a cercare la dipendenza in questione all'interno dei suoi moduli
- Ma come fa Koin a reperire un'istanza di **Context**?
  - Gli viene fornita in fase di avvio dell'app (vedi prossima slide)

## 3.6. Avvio di Koin

- La DI di Koin va avviata a livello globale di applicazione
- Per farlo, possiamo creare una classe nella root del progetto che eredita da **Application** e fare override del metodo **onCreate**
- La classe va aggiunta al file manifest

ThemeToggleApplication.kt

```
class ThemeToggleApplication : Application() {  
    override fun onCreate() {  
        super.onCreate()  
  
        startKoin {  
            androidLogger()  
            androidContext(this@ThemeToggleApplication)  
            modules(appModule)  
        }  
    }  
}
```

AndroidManifest.xml

```
<application  
    android:name=".ThemeToggleApplication"
```

## 3.7. Creazione ViewModel tramite Koin

- L'ultimo passaggio è quello di sostituire l'helper **viewModel** con quello fornito da Koin per istanziare il ViewModel

```
val themeViewModel = viewModel<ThemeViewModel>()
```



```
val themeViewModel = koinViewModel<ThemeViewModel>()
```

## 4. TravelDiary - Struttura avanzata progetto

- Modificare l'architettura dell'app TravelDiary in base a quanto visto negli esercizi di oggi:
  - Configurare la dependency injection con Koin
  - Memorizzare lo stato di **AddTravelScreen** e **SettingsScreen** in dei corrispondenti ViewModel:
    - **AddTravelViewModel**
    - **SettingsViewModel**
  - Memorizzare l'username della pagina Settings nel DataStore tramite repository
    - **SettingsRepository**

# 4. TravelDiary - Struttura avanzata progetto

---

- **Hint:** navigation e creazione ViewModel
  - Il punto migliore per la creazione del ViewModel di una schermata è il blocco **composable** della rotta corrispondente

```
composable<TravelDiaryRoute.Settings> {
    val settingsVm = koinViewModel<SettingsViewModel>()
    SettingsScreen(settingsVm.state, settingsVm::setUsername, navController)
}
```

- Non dimenticate **.collectAsStateWithLifecycle()** se lo stato del ViewModel è uno **StateFlow** (non lo è nell'esempio sopra)

# 4. TravelDiary - Struttura avanzata progetto

---

- **Hint:** ViewModel con molti metodi
  - In base all'implementazione, **AddTravelViewModel** potrebbe contenere vari metodi, ad esempio: **setDestination**, **setDate**, **setDescription**
  - In questi casi, può essere una buona idea raggruppare tutti i metodi in un oggetto **actions** all'interno del ViewModel, così da non doverli passare individualmente dal ViewModel allo screen

AddTravelViewModel.kt

```
interface AddTravelActions {  
    fun setDestination(title: String)  
    fun setDate(date: String)  
    fun setDescription(description: String)  
}
```

Navigation.kt

```
AddTravelScreen(  
    state,  
    addTravelVm.actions,  
    navController  
)
```

# 4. TravelDiary - Struttura avanzata progetto

---

- **Hint:** TextField e Flow (schermata Settings)
  - Modificare il valore contenuto in un TextField mentre l'utente sta scrivendo resetta la posizione del cursore, creando una bad user experience
  - Per questo motivo, il valore di un TextField non dovrebbe mai provenire da un Flow, che può emettere nuovi valori in qualsiasi momento

# 4. TravelDiary - Struttura avanzata progetto

---

- **Hint:** TextField e Flow (schermata Settings) pt. 2
  - Come facciamo allora a sincronizzare il valore di un TextField con il DataStore?
    - Alla creazione del ViewModel, leggiamo il valore presente nel DataStore e lo salviamo in una variabile non Flow (es. **mutableStateOf**)
    - Ogni volta che il valore della variabile viene modificato, lo scriviamo nel DataStore
  - Come eseguiamo codice alla creazione del ViewModel?
    - Con il blocco **init** di Kotlin

```
class SettingsViewModel (private val repository: SettingsRepository) : ViewModel() {  
    // ...  
  
    init {  
        println("SettingsViewModel created")  
    }  
}
```



---

**Tocca a voi!**



# Riferimenti

---

- State in Compose  
<https://developer.android.com/jetpack/compose/state>
- ViewModel  
<https://developer.android.com/topic/libraries/architecture/viewmodel>
- StateFlow  
<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow>
- Dependency Injection  
<https://developer.android.com/training/dependency-injection>
- DataStore  
<https://developer.android.com/topic/libraries/architecture/datastore>
- Koin su Android  
<https://insert-koin.io/docs/quickstart/android/>