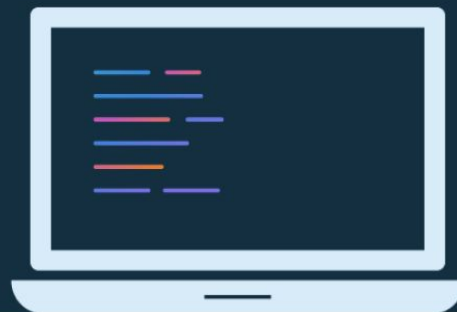




# Lezione 2.6: Architettura delle app (UI layer)



[Deeper understanding - Kate Bush](#)



# Questa lezione

## Lezione 2.6: Architettura delle app (UI layer)

- Architettura di un'app
- ViewModel
- ViewModel e Data binding
- LiveData
- Trasformazioni di LiveData

# Architettura di un'app

# Effetti delle scorciatoie

- Fattori esterni, come scadenze fisse vicine (ad esempio la data di consegna di un progetto), possono portare a scelte sbagliate di progettazione
- Le decisioni prese però hanno conseguenze nel futuro in termini di manutenibilità nel lungo termine
- È necessario bilanciare i due aspetti (consegna nei tempi e futura fatica nella manutenzione)

# Effetti delle scorciatoie

- Ogni patch (toppa) che inseriamo nel codice per risolvere il problema specifico senza curarci dell'architettura complessiva, produce *“technical debt”*
- Il costo del lavoro futuro di riprogettazione accumulato per aver evitato di risolvere il problema subito



# Esempi di scorciatoie rapide

- Progettare l'app per un device specifico
- Fare il copia e incolla di codice dal web senza approfondire
- Inserire tutta la logica applicativa in un solo Activity file
- Lasciare le stringhe hardcoded nel codice

# Perché c'è bisogno di una buona architettura?

- Definisce chiaramente la struttura della logica applicativa
- Rende più facile lo sviluppo e la collaborazione tra developer
- Rende il codice più facile da testare e l'app più scalabile
- Consente di sfruttare i benefici di problemi già risolti da altri (ad esempio nell'uso di best practice)
- Permette di risparmiare tempo e risorse nel lungo termine riducendo il *technical debt* a mano a mano che il progetto cresce

# Android Jetpack

- Una suite di librerie, tool e guide Android che incorporano best practice e forniscono retrocompatibilità alle nostre app, semplificando operazioni complesse
- Jetpack comprende le librerie con il package `androidx.*`
  - Esempi di librerie: activity, appcompat, cameraX, compose, datafragment, navigation, hilt, livedata, workmanager, room, ...



# “Guide to an app architecture”

Guida Android allo sviluppo “sostenibile”:

<https://developer.android.com/jetpack/guide>

Principi generali:

1. **Separation of concerns:** principio fondamentale di progettazione modulare secondo cui i diversi aspetti di un software devono essere progettati come componenti separate.

Questo per la progettazione mobile è importante perché le Activity ed i Fragment non sono componenti affidabili per memorizzare lo stato (possono essere distrutti in base ad eventi incontrollabili)

# “Guide to an app architecture”

Guida Android allo sviluppo “sostenibile”:

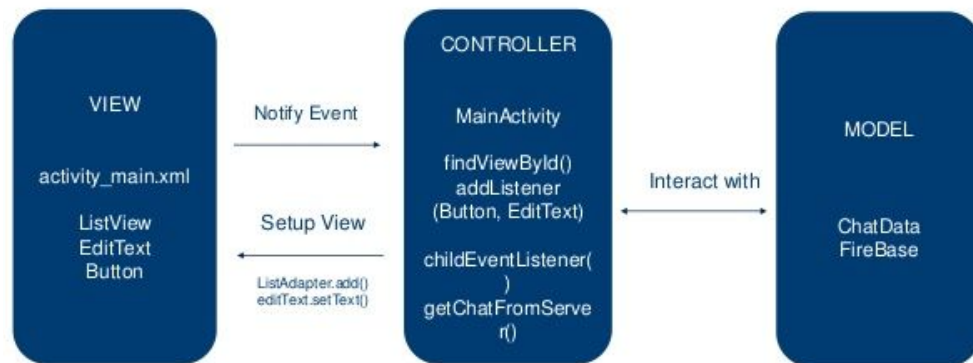
<https://developer.android.com/jetpack/guide>

Principi generali:

2. **Drive UI from a model:** i dati usati in un'interfaccia devono essere derivati da un modello di dati, cioè da componenti responsabili per la gestione dei dati, indipendenti dalle View. Esistono vari modi per gestire la persistenza (DB, file, shared preferences, ...)

# Diversi “pattern” per la separation of concerns

- La soluzione più vecchia (‘70) è il MVC (Model View Controller):
  - La View rappresenta il file XML di layout
  - Il controller è l’Activity, in cui inseriamo la logica
  - Il model rappresenta le classi in cui gestiamo i dati



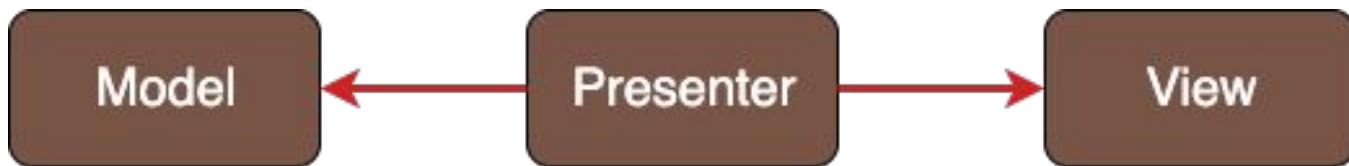
# Diversi “pattern” per la separation of concerns

Problemi:

- L'indipendenza delle componenti è molto limitata
- Tutto è vincolato al ciclo di vita delle Activity, che non può essere completamente controllato

# Diversi “pattern” per la separation of concerns

- Altra soluzione, MVP (Model View Presenter):
  - La View include i layout, le Activity ed i Fragment, e si occupa solo del rendering dell'interfaccia e di minima logica
  - Il presenter contiene la logica, gestisce gli eventi e fa da bridge con il Model che fornisce i dati

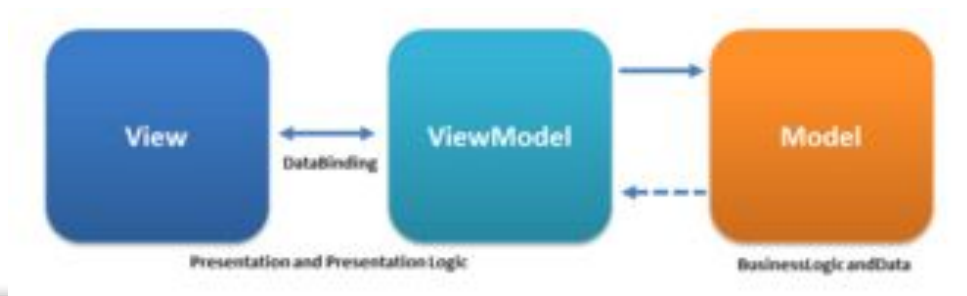


# Diversi “pattern” per la separation of concerns

- Pro:
  - Semplice da implementare
- Contro:
  - Non risolve ancora il fatto che il lifecycle del presenter è vincolato a quello delle view

# Diversi “pattern” per la separation of concerns

- MVVM (Model-View-ViewModel):
  - View e Model sono gli stessi dei pattern precedenti
  - Il ViewModel è un componente che gestisce la logica ma è disaccoppiato dal lifecycle delle view. In sintesi: aggiorna dei dati dentro un LiveData, il quale viene osservato dalla View



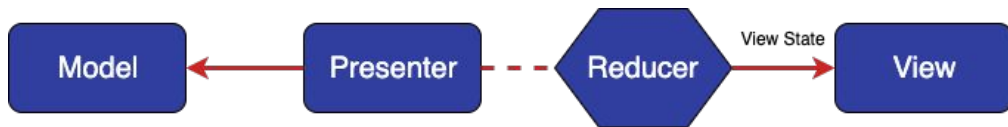
# Diversi “pattern” per la separation of concerns

- Pro:
  - Il ViewModel sopravvive al lifecycle delle View
  - Più View possono essere legate allo stesso ViewModel
- Contro:
  - Più complesso da implementare, richiede componenti specifiche delle librerie di Android



# Diversi “pattern” per la separation of concerns

- MVI (Model-View-Interactor):
  - Evoluzione del MVP
  - Il presenter comunica con la View solo attraverso un ViewState che può modificare
  - Lo StateReducer gestisce il merge tra il vecchio ViewState e quello nuovo



# Diversi “pattern” per la separation of concerns

- Pro:
  - Il ViewState rende più chiara l'interazione con la View
- Contro:
  - Lo State introduce un ulteriore livello di complessità da gestire
  - Stessi limiti del MVP relativi al lifecycle

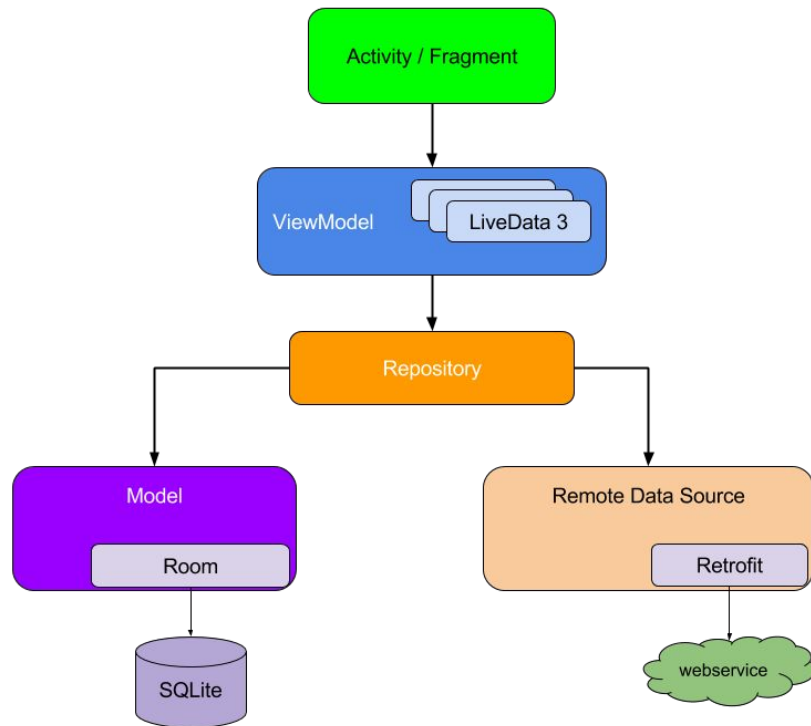
# Diversi “pattern” per la separation of concerns

- Android utilizza l’approccio MVVM per costruire app modulari e manutenibili
- Le componenti Jetpack aiutano il programmatore a costruire app robuste, facili da testare e manutenibili

# Architettura raccomandata di un'app

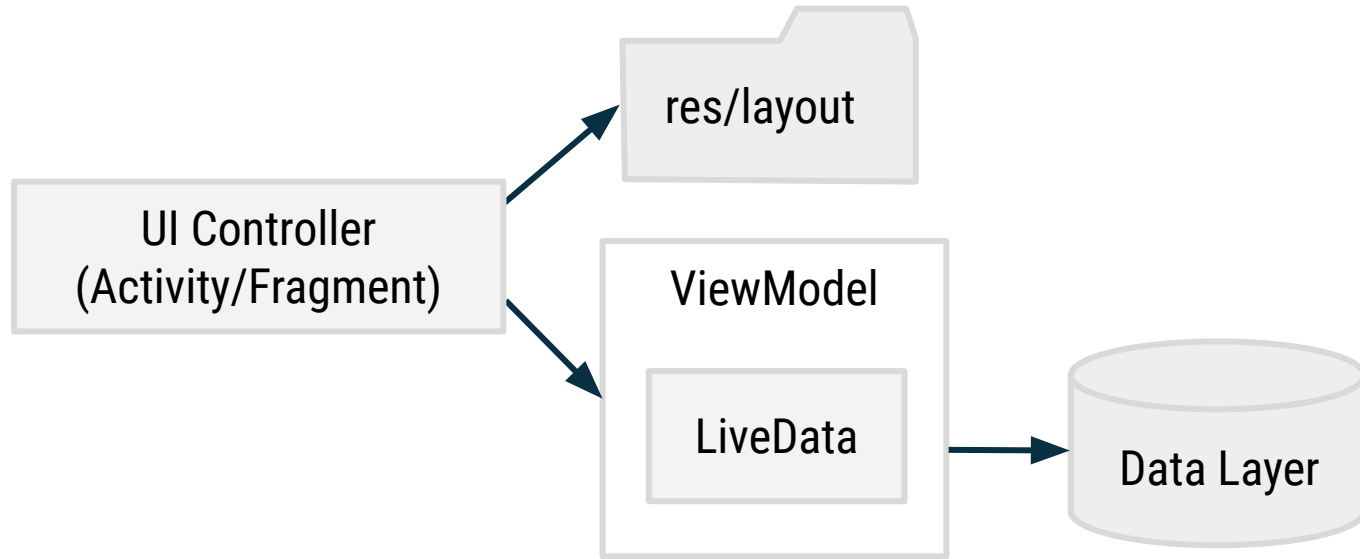
Principio: ogni componente interagisce solo con quelle sopra e sotto.

Questa architettura rappresenta un punto di partenza per la maggior parte dei casi, ma va personalizzata per le specifiche esigenze.



# ViewModel

# ViewModel



# Gradle: lifecycle extensions

In `app/build.gradle` file:

```
dependencies {  
    implementation  
    "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"  
    implementation "androidx.activity:activity-ktx:$activity_version"  
}
```

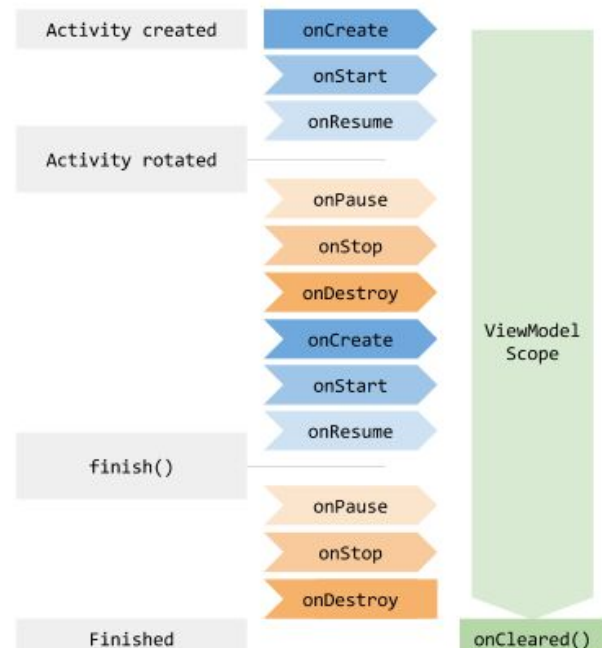
# ViewModel

- Prepara i dati per la UI
- Non deve contenere riferimenti ad Activity, Fragment, o View nel layout
- Legato ad un lifecycle (di un'Activity o Fragment)
- Permette ai dati di sopravvivere malgrado un cambiamento di configurazione
- Esiste in memoria finché il lifecycle non termina in modo definitivo



# Lifetime di un ViewModel

Nota: lo “scope” del ViewModel va dalla creazione dell’Activity (o Fragment) collegato, fino alla sua distruzione finale, sopravvivendo ad eventuali distruzioni temporanee (e dunque mantenendo i dati)



# Esempio: contatore per gioco a squadre



# ViewModel class

```
abstract class ViewModel
```

## Summary

### Public constructors

```
<init>()
```

ViewModel is a class that is responsible for preparing and managing the data for an [Activity](#) or a [Fragment](#).

### Protected methods

open [Unit](#)

```
onCleared()
```

This method will be called when this ViewModel is no longer used and will be destroyed.

### Extension properties

From [androidx.lifecycle](#)

[CoroutineScope](#)

[viewModelScope](#)

[CoroutineScope](#) tied to this [ViewModel](#).

# Implementare un ViewModel

```
class ScoreViewModel : ViewModel() {  
    var scoreA : Int = 0  
    var scoreB : Int = 0  
    fun incrementScore(isTeamA: Boolean) {  
        if (isTeamA) {  
            scoreA++  
        }  
        else {  
            scoreB++  
        }  
    }  
}
```

# Usare un ViewModel

```
class MainActivity : AppCompatActivity() {  
    // Delegate provided by androidx.activity.viewModels  
    val viewModel: ScoreViewModel by viewModels()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        val scoreViewA: TextView = findViewById(R.id.scoreA)  
        scoreViewA.text = viewModel.scoreA.toString()  
    }  
}
```

Usiamo la variabile nel ViewModel

# Usare un ViewModel

Within `MainActivity onCreate()`:

```
val scoreViewA: TextView = findViewById(R.id.scoreA)
val plusOneButtonA: Button = findViewById(R.id.plusOne_teamA)

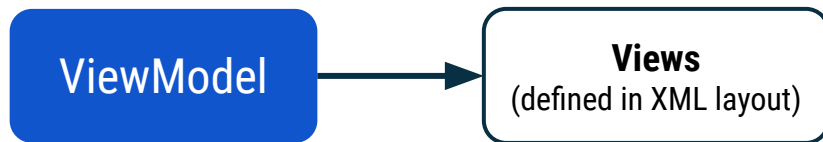
plusOneButtonA.setOnClickListener {
    viewModel.incrementScore(true)
    scoreViewA.text = viewModel.scoreA.toString()
}
```

↑  
Più avanti vedremo come evitare di dover aggiornare manualmente la `TextView`

# ViewModel e Data binding

# ViewModel e data binding

- App architecture without data binding





# Data binding in XML rivisitato

Specifichiamo il ViewModel nel tag `data` del binding

```
<layout>
  <data>
    <variable>
      name="viewModel"
      type="com.example.kabaddikounter.ScoreViewModel" />
    </data>
  <ConstraintLayout ../>
</layout>
```

Nota: questa dichiarazione non lega ancora l'istanza di ViewModel al layout

# Agganciare un ViewModel ai dati

```
class MainActivity : AppCompatActivity() {  
  
    val viewModel: ScoreViewModel by viewModels()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        val binding: ActivityMainBinding = DataBindingUtil.setContentViews(this,  
            R.layout.activity_main)  
  
        binding.viewModel = viewModel  
        ...  
    }  
}
```

Così si aggancia la variabile di layout alla reale istanza di ViewModel

# Usare un ViewModel da un data binding

In `activity_main.xml`:

```
<TextView
    android:id="@+id/scoreViewA"
    android:text="@{viewModel.scoreA.toString()}" />
    ...
```

# ViewModel e data binding

```
override fun onCreate(savedInstanceState: Bundle?) {  
    ...  
    val binding: ActivityMainBinding = DataBindingUtil setContentView(this,  
        R.layout.activity_main)  
  
    binding.plusOneButtonA.setOnClickListener {  
        viewModel.incrementScore(true)  
        binding.scoreViewA.text = viewModel.scoreA.toString() }  
}
```

Nota: il fatto di aver legato la variabile di layout all'istanza di ViewModel significa che la TextView avrà il valore dal ViewModel in fase di istanziamento. Ma **ciò non significa che appena il valore cambia, la TextView verrà aggiornata!** Dovremo quindi ancora farlo manualmente.

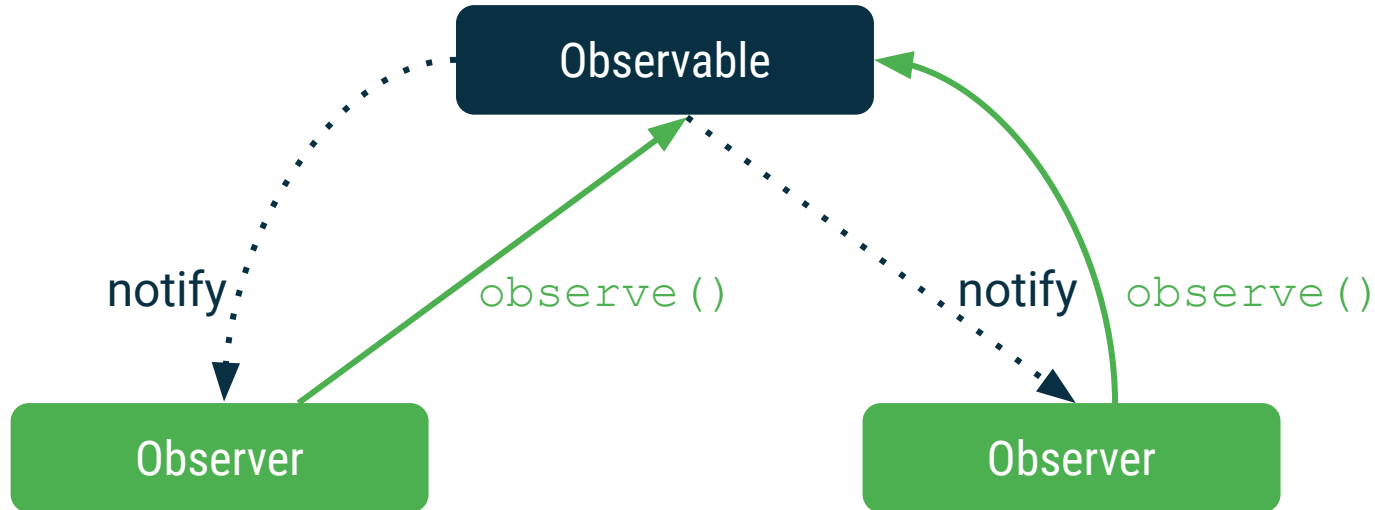
# LiveData

# “Observer” design pattern

Un pattern che prevede un oggetto (osservato) che mantiene una lista di oggetti dipendenti (osservatori) a cui notificare ogni cambiamento di stato

- Osservatori ricevono i cambiamenti di stato ed eseguono dei metodi appropriati
- Gli osservatori possono essere aggiunti o rimossi in qualunque momento

# “Observer” design pattern



# LiveData

- Un componente lifecycle-aware che gestisce dati e che può essere osservato
- E' un "wrapper" che può essere usato su qualsiasi tipo, incluse le liste (ad esempio, `LiveData<Int>` conterrà un `Int`)
- Spesso usato dai ViewModel per gestire singole proprietà
- Osservatori (Activity o Fragment) possono essere aggiunti o rimossi
  - `observe(owner: LifecycleOwner, observer: Observer)`
  - `removeObserver(observer: Observer)`



# LiveData versus MutableLiveData

LiveData<T>	MutableLiveData<T>
<ul style="list-style-type: none"><li>● <code>getValue()</code></li></ul>	<ul style="list-style-type: none"><li>● <code>getValue()</code></li><li>● <code>postValue(value: T)</code></li><li>● <code>setValue(value: T)</code></li></ul>

T is the type of data that's stored in `LiveData` or `MutableLiveData`.

# Use LiveData in ViewModel

```
class ScoreViewModel : ViewModel() {  
  
    private val _scoreA = MutableLiveData<Int>(0)  
    val scoreA: LiveData<Int>  
        get() = _scoreA  
  
    fun incrementScore(isTeamA: Boolean) {  
        if (isTeamA) {  
            _scoreA.value = _scoreA.value!! + 1  
        }  
        ...  
    }  
}
```

# Aggiungere un observer con LiveData

Creare un click listener per incrementare lo score nel `ViewModel`:

```
binding.plusOneButtonA.setOnClickListener {  
    viewModel.incrementScore(true)  
}
```

Creare un observer per aggiornare lo score del team A a schermo:

```
val scoreA_Observer = Observer<Int> { newValue ->  
    binding.scoreViewA.text = newValue.toString()  
}
```

Aggiungere un observer sul `LiveData` `scoreA` nel `ViewModel`:

```
viewModel.scoreA.observe(this, scoreA_Observer)
```

# Two-way data binding

- In questo modo abbiamo realizzato un “two-way binding” con il `ViewModel` ed i `LiveData`.
- Soluzione alternativa: effettuare il binding del `LiveData` direttamente nel layout XML elimina la necessità di definire esplicitamente un observer nel codice

# Esempio di layout XML

```
<layout>
  <data>
    <variable>
      name="viewModel"
      type="com.example.kabaddikounter.ScoreViewModel" />
    </data>
    <ConstraintLayout ...>
      <TextView ...
        android:id="@+id/scoreViewA"
        android:text="@{viewModel.scoreA.toString()}" />
      ...
    </ConstraintLayout>
</layout>
```

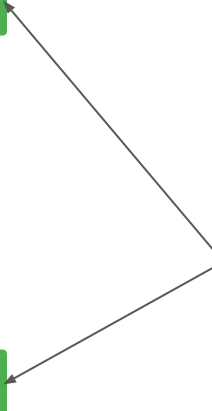
# Esempio di Activity

```
class MainActivity : AppCompatActivity() {  
    val viewModel: ScoreViewModel by viewModels()  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        val binding: ActivityMainBinding = DataBindingUtil  
            .setContentView(this, R.layout.activity_main)  
  
        binding.viewModel = viewModel  
        binding.lifecycleOwner = this  
  
        binding.plusOneButtonA.setOnClickListener {  
            viewModel.incrementScore(true)  
        }  
        ...  
    }  
}
```

# Esempio di ViewModel

```
class ScoreViewModel : ViewModel() {  
    private val _scoreA = MutableLiveData<Int>(0)  
    val scoreA : LiveData<Int>  
        get() = _scoreA  
    private val _scoreB = MutableLiveData<Int>(0)  
    val scoreB : LiveData<Int>  
        get() = _scoreB  
    fun incrementScore(isTeamA: Boolean) {  
        if (isTeamA) {  
            _scoreA.value = _scoreA.value!! + 1  
        } else {  
            _scoreB.value = _scoreB.value!! + 1  
        }  
    }  
}
```

Ora le modifiche  
sono riflesse nella UI



# Transformare LiveData



# Manipolare LiveData con trasformazioni

Un `LiveData` può essere trasformato in un nuovo `LiveData`. Ad esempio, possono essere usate le seguenti operazioni, che prendono in ingresso l'oggetto `LiveData` da trasformare e la funzione da applicare

- `map()`
- `switchMap()`

# LiveData con trasformazioni: map

Ad esempio, supponiamo di voler ritornare lo scoreA come numero se questo è  $\leq 10$ , e come stringa se è  $> 10$ .

```
val result: LiveData<String> = Transformations.map(viewModel.scoreA) {  
    x -> if (x > 10) "A Wins" else ""  
}
```

# Riassunto

# Riassunto

In questa lezione abbiamo imparato a:

- Seguire principi di progettazione, in particolare la “separation-of-concerns” per avere app più facilmente manutenibili e per ridurre il technical debt
- Creare un `ViewModel` per gestire i dati separatamente da un UI controller
- Usare `ViewModel` con il data binding per avere un’interfaccia reattiva con meno codice
- Usare gli observer per avere automaticamente aggiornamenti dai `LiveData`

# Link

- [Guide to app architecture](#)
- [Android Jetpack](#)
- [ViewModel Overview](#)
- [Android architecture sample app](#)
- [ViewModelProvider](#)
- [Lifecycle Aware Data Loading with Architecture Components](#)
- [ViewModels and LiveData: Patterns + AntiPatterns](#)