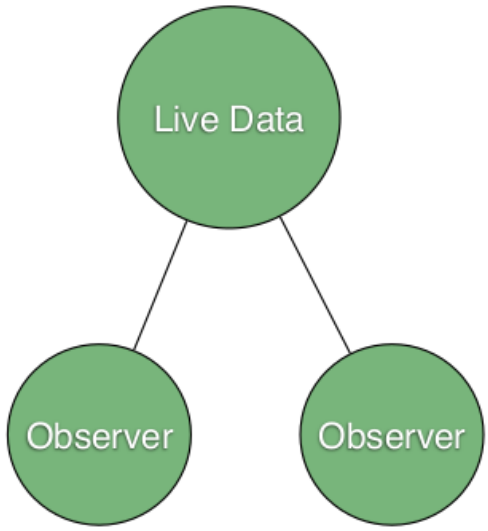
8-Live Data

Live Data

La classe che gestisce i dati `LiveData` è osservabile da entità esterne, e l'unica caratteristica che presenta è che è consapevole del suo ciclo di vita, che rispetto a quello di altri componenti come:

- activities
- fragments
- services

> Nota
> Attraverso un LiveData è possibile aggiornare solo gli osservatori che sono in uno stato di vita attivo



Live Data & Lifecycle Owners

LiveData è rappresentato dalla classe `Observer` e può essere associato a uno stato tra:

- `CREATED`
- `DESTROYED`
- `INITIALIZED`
- `STARTED`
- `RESUMED`

> Nota
> Le callback di notifica vengono inviate solo agli osservatori attivi

È possibile registrare un osservatore associato a un componente che implementa l'interfaccia `LifecycleOwner` (ad esempio Activity, Fragment e Service) e la relazione con quest'ultimo consente di gestire correttamente lo stato di un oggetto `LiveData` e impostarlo su `DESTROYED` quando necessario.

Live Data Advantages

I vantaggi sono:

- "Assicura che l'interfaccia utente corrisponda allo stato dei dati", le notifiche vengono inviate agli osservatori quando lo stato del ciclo di vita cambia. È possibile consolidare il codice in modo da aggiornare l'interfaccia utente solo quando avviene un cambiamento.
- "Nessuna perdita di memoria", gli osservatori sono legati agli oggetti nel ciclo di vita e vengono liberati quando il ciclo di vita associato viene distrutto.
- "Nessun arresto anomalo dovuto ad attività interrotte" e "Corrette modifiche di configurazione", quando il ciclo di vita di un osservatore è inattivo (ad esempio, un'attività nello stack di retroazione), non riceverà eventi associati a LiveData.
- "Nessuna gestione manuale del ciclo di vita" e "Dati sempre aggiornati", l'interfaccia utente osserva i dati rilevanti senza interrompere o riprendere le osservazioni. LiveData gestisce automaticamente le osservazioni poiché è consapevole dei cambiamenti dello stato del ciclo di vita mentre osserva
- "Condivisione delle risorse", nn oggetto LiveData può essere implementato utilizzando il pattern singleton per essere condiviso nell'app. Questo approccio consente all'oggetto LiveData di essere connesso al sistema di servizio solo una volta e qualsiasi osservatore esterno che ha bisogno della risorsa può semplicemente osservare l'oggetto LiveData.

Extend Live Data Object

LiveData è un wrapper che può essere utilizzata con ogni tipo di dato, includendo ad esempio oggetti che implementano collezioni, come le liste.

```
public class MyLiveData extends LiveData<MyDataStructure> {
	private MyDataManager dataManager;
	private MyDataListener listener = new MyDataListener() {
		@Override
		public void onDataChanged(MyDataStructure newData) {
			// Metodo che aggiorna il valore dell'istanza corrente di LiveData.
			// Questo cambiamento attiva la notifica verso tutti gli osservatori
attivi che riceveranno i dati aggiornati.
			setValue(newData);
		}
	});
	public MyLiveData() {
		dataManager = new MyDataManager();
	}
	// Metodo chiamato quando l'oggetto LiveData ha un osservatore attivo.
	// In questo momento, è necessario iniziare a osservare i cambiamenti dei dati da
questo metodo
	@Override
	protected void onActive() {
		dataManager.requestDataUpdates(listener);
	}
	// Metodo chiamato quando l'oggetto LiveData non ha alcun osservatore attivo.
	// L'assenza di osservatori significa che non c'è motivo di mantenere attivo il
listener verso il Data Manager
	@Override
	protected void onInactive() {
		dataManager.removeUpdates(listener);
	}
}
```

Define Your Listener

Bisogna definire una propria interfaccia DataListener per creare un collegamento comune tra LiveData e la propria classe Data Manager.

Guardando l'esempio precedente, questa è l'interfaccia del Listener associata::

```
public interface MyDataListener {
	void onDataChanged(MyDataStructure newData);
}
```

Use the Listener in Data Manager

Il proprio DataManager dovrebbe essere consapevole degli ascoltatori attivi e chiamarli quando c'è una notifica significativa da attivare.

DataManager implementato come Singleton

```
public class MyDataManager {
	private MyDataStructure myData = null;
	private List<MyDataListener> listenerList = null;
	private static LogDescriptorManager instance = null;
	private LogDescriptorManager(){
		this.listenerList = new ArrayList<>();
	}
	public static LogDescriptorManager getInstance(){
		if(instance == null)
```

```
                instance = new LogDescriptorManager();
                return instance;
        }
        // Metodo generico per aggioranre i dati
        // Dopo l'aggiornamento ogni ascoltatore viene notificato
        public void updateData(MyDataStructure newData){
                this.myData = newData;
                notifyListeners();
        }
        // Metodo che per notificare a tutti i listener registrati per un aggioranemnto dei
dati
        private void notifyListeners() {
                for(int i=0; i<this.listenerList.size(); i++)
                        if(this.listenerList.get(i) != null)
                                this.listenerList.get(i).onDataChanged(this.myData);
        }
        // Metodo per aggiungere un ascoltatore
        public void requestDataUpdates(MyDataListener listener) {
                if(listener != null) {
                        this.listenerList.add(listener);
                        // Per una nuova registrazione se c'è un dato disponibile il listener
                        // viene notificato immediatamente
                        if(this.myData != null)
                                listener.onDataChanged(this.myData);
                }
        }

        // Metodo per rimuovere un ascoltatore
        public void removeUpdates(MyDataListener listener) {
                if(listener != null)
                this.listenerList.remove(listener);
        }
}
```

Register an Observer in your App Componentù

fragment

```
public class MyFragment extends Fragment {
        @Override
        public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
                super.onViewCreated(view, savedInstanceState);
                LiveData<MyDataStructure> myLiveData = new MyLiveData();
                // Il metodo observe() utilizza la reference LifecycleOwner associata al
fragment.
                // L'osservatore è legato all'oggetto Lifecycle associato al proprietario.
                // Se l'oggetto Lifecycle non è in uno stato attivo, l'osservatore non viene
chiamato
                // anche se il valore cambia.
                // Quando l'oggetto Lifecycle viene distrutto, l'osservatore verrà rimosso
                myLiveData.observe(getViewLifeycleOwner(), newData -> {
                        // Update the UI.
                });
        }
}
```

activity

```
public class MyFragment extends Fragment {
@Override
        public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
                super.onViewCreated(view, savedInstanceState);
                LiveData<MyDataStructure> myLiveData = new MyLiveData();
                // Nell'activity è possibile usare direttamente il riferimento a this
                // invece di getViewLifeycleOwner() usato nel fragment
                myLiveData.observe(this, newData -> {
                        // Update the UI.
                });
        }
}
```

LiveData as a Singleton

Come detto sopra gli oggetti LiveData sono consapevoli del ciclo di vita e possono essere condivisi tra diverse activity, fragment e services.

Per mantenere il codice semplice, è possibile implementare la classe LiveData come un singleton.

```
public class MyLiveData extends LiveData<MyDataStructure> {

	private static MyLiveData sInstance;
	private MyDataManager dataManager;

	private MyDataListener listener = new MyDataListener() {
		@Override
		public void onDataChanged(MyDataStructure newData) {
			setValue(newData);
		}
	};
	// metodo publico getInstance che controlla se un istanza è disponibile o no
	// Se non la fosse la crea e la ritorna
	public static MyLiveData getInstance() {
		if (sInstance == null)
			sInstance = new MyLiveData();

		return sInstance;
	}

	// costruttore privato
	private MyLiveData() {
		dataManager = new MyDataManager();
	}
	@Override
	protected void onActive() {
		dataManager.requestDataUpdates(listener);
	}
	@Override
	protected void onInactive() {
		dataManager.removeUpdates(listener);
	}
}
```

LiveData & Room

```
@Dao
public interface LogDao {
	// Tutti i metodi dedicati sono stati aggiunti ad osservare le variazioni sulla
LogList
	@Query("SELECT * FROM logs ORDER BY id DESC")
	LiveData<List<LogDescriptor>> getAllLiveData();

	@Query("SELECT * FROM logs ORDER BY id DESC")
	List<LogDescriptor> getAll();

	@Query("SELECT * FROM logs WHERE id IN (:userIds)")
	List<LogDescriptor> loadAllByIds(int[] userIds);

	@Insert
	void insertAll(LogDescriptor... logs);

	@Delete
	void delete(LogDescriptor logDescriptor);
}


private void observeLogData(){
	// Oserva la lista LiveData nella UI
	this.logDescriptorManager.getLogLiveDataList().observe(getViewLifecycleOwner(), new
Observer<List<LogDescriptor>>() {
		@Override
		public void onChanged(List<LogDescriptor> logDescriptors) {
			if(logDescriptors != null){
				Log.d(TAG, "Update Log List Received ! List Size: " +
logDescriptors.size());
				refreshRecyclerView(logDescriptors, 0);
			}
			else
				Log.e(TAG, "Error observing Log List ! Received a null Object
!");
		}
	});
}

// In questo esempio un metodo specifico è stato creato per gestire gli aggiornamenti di UI
// provenienti da callback per aggiornamenti di dati
private void refreshRecyclerView(List<LogDescriptor> updatedList, int scrollPosition){
```

```
        mAdapter.setDataset(updatedList);
        mAdapter.notifyDataSetChanged();
        if(scrollPosition >= 0)
                mLayoutManager.scrollToPosition(scrollPosition);
}
```

Dependencies

Dipendenze Live Data & ViewModel

```
dependencies {
        def lifecycle_version = "2.2.0"
        def arch_version = "2.1.0"
        // ViewModel
        implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"
        // LiveData
        implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
        // Lifecycles only (without ViewModel or LiveData)
        implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"
        // Saved state module for ViewModel
        implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lifecycle_version"
        // Annotation processor
        annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"

        // alternately - if using Java8, use the following instead of lifecycle-compiler
        implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"
        // optional - helpers for implementing LifecycleOwner in a Service
        implementation "androidx.lifecycle:lifecycle-service:$lifecycle_version"
        // optional - ProcessLifecycleOwner provides a lifecycle for the whole application
process
        implementation "androidx.lifecycle:lifecycle-process:$lifecycle_version"
        // optional - ReactiveStreams support for LiveData
        implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"
        // optional - Test helpers for LiveData
        testImplementation "androidx.arch.core:core-testing:$arch_version"
}
```

compatibilità con Java 1.8

```
compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
}
```