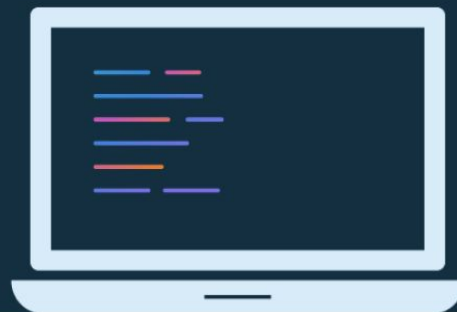


Lezione 2.7: Architettura delle app (persistenza)



[Deep blue - Arcade fire](#)



Questa lezione

Lezione 2.7: Architettura delle app (persistenza)

- Memorizzare dati
- La libreria Room per la persistenza
- Asynchronous programming
- Coroutines

Memorizzare dati

Modi per memorizzare dati su un app

- App-specific storage: memorizza i file che servono per l'app (file di testo, JSON, file multimediali)
- Shared storage: file che devono essere condivisi con altre app
- Preferences: dati privati e primitivi nella forma key-value
- Database: per memorizzare dati strutturati in modo privato

App-specific storage

A volte è necessario memorizzare file in modo privato, a cui nessun'altra app sarà in grado di accedere:

- **Storage interno:** directory private per memorizzare file e cache data. Dalle API 10 sono criptate.
- **Storage esterno:** directory per file e cache data per l'app. Se i dati vanno condivisi con altre app si dovrebbero memorizzare nelle directory condivise dello storage esterno

Nota: quando l'app viene disinstallata i file vengono eliminati.

App-specific storage

Storage interno: utile per memorizzare pochi file utili per il funzionamento dell'app che devono rimanere privati. Si può usare l'API `File` per accedere, leggere e scrivere:

```
val file = File(context.filesDir, filename)
```

O, in alternativa, si può leggere/scrivere con `FileOutputStream`:

```
val fileContents = "Hello world!"  
context.openFileOutput("myfile", Context.MODE_PRIVATE).use {  
    it.write(fileContents.toByteArray())}
```

App-specific storage

Qualche metodo utile:

- Lista di file: `var files: Array<String> = context.listFiles()`
- Creare/accedere ad un cache file:
`File.createTempFile(filename, null, context.cacheDir)`
`val cacheFile = File(context.cacheDir, filename)`
- Eliminare un cache file: `context.deleteFile(cacheFileName)`

Per una guida dettagliata:

<https://developer.android.com/training/data-storage/app-specific#kotlin>

Preferences

Coppie key-value memorizzate in file sul device. Sono utili per salvare le impostazioni dell'app invece di usare un database:

- API `SharedPreferences` per leggere, salvare, modificare, cancellare delle preferenze

Preferences

In cosa differisce dall'approccio con `saveInstanceState`?

- I dati rimangono anche se l'app viene chiusa o il dispositivo riavviato
- Utile per dati che devono persistere tra diverse sessioni
 - tipicamente usato per le impostazioni dell'utente

Preferences

E' possibile creare un nuovo file o accedere ad uno esistente con:

```
val sharedPref = applicationContext.getSharedPreferences(  
    getString(R.string.preference_file_key), Context.MODE_PRIVATE)
```

Il nome dovrebbe essere univoco, es: `com.example.myapp.PREFERENCE_FILE_KEY`

Questo è valido a livello di app, ma se invece vogliamo un file di preferenze per un'activity usiamo questo metodo (senza nome del file):

```
val sharedPref = this.getSharedPreferences(Context.MODE_PRIVATE)
```

Preferences: scrittura

Per scrivere:

```
val sharedPref = this.getPreferences(Context.MODE_PRIVATE)  
    oppure = applicationContext.getSharedPreferences(...)
```

```
val editor = sharedPref.edit()  
editor.putInt(getString(R.string.saved_high_score_key),  
    newHighScore)  
editor.apply()
```

Se terminiamo con `apply()` -> scrittura sincrona in memoria ma asincrona su disco. Se terminiamo con `commit()` è sincrona (bloccante!)

- 1) Creare un nuovo progetto
- 2) Creare un'activity
- 3) In onCreate, creare un file di SharedPreferences
- 4) Inserire un intero
- 5) Recuperare quell'intero
- 6) Visualizzarlo a video in una TextView

Preferences: lettura

Per leggere:

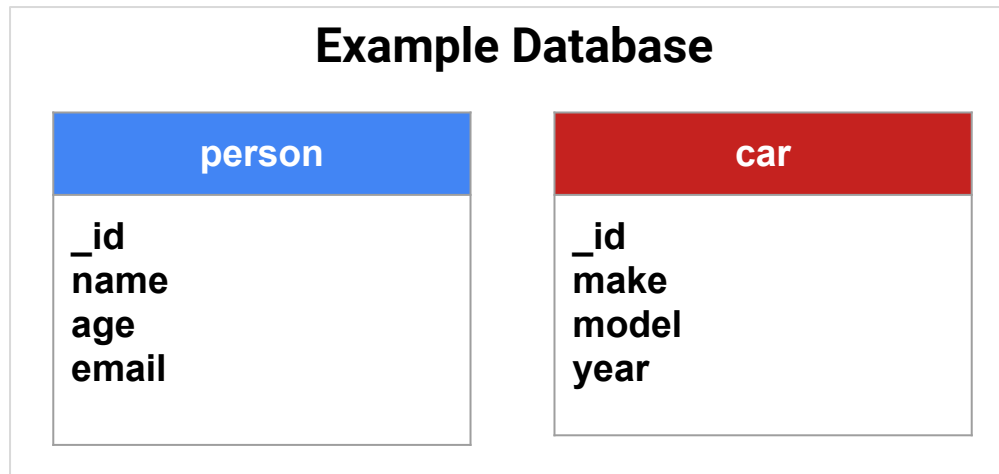
```
val sharedPref = this.getPreferences(Context.MODE_PRIVATE)  
    oppure = applicationContext.getSharedPreferences(...)
```

```
val highScore =  
sharedPref.getInt(getString(R.string.saved_high_score_key),  
defaultValue)
```

Cos'è un database (relazionale)?

Una collezione di dati strutturati che possono essere facilmente acceduti, cercati ed organizzati, composto da:

- Tabelle (relazioni)
- Righe (tuple)
- Colonne (domini)

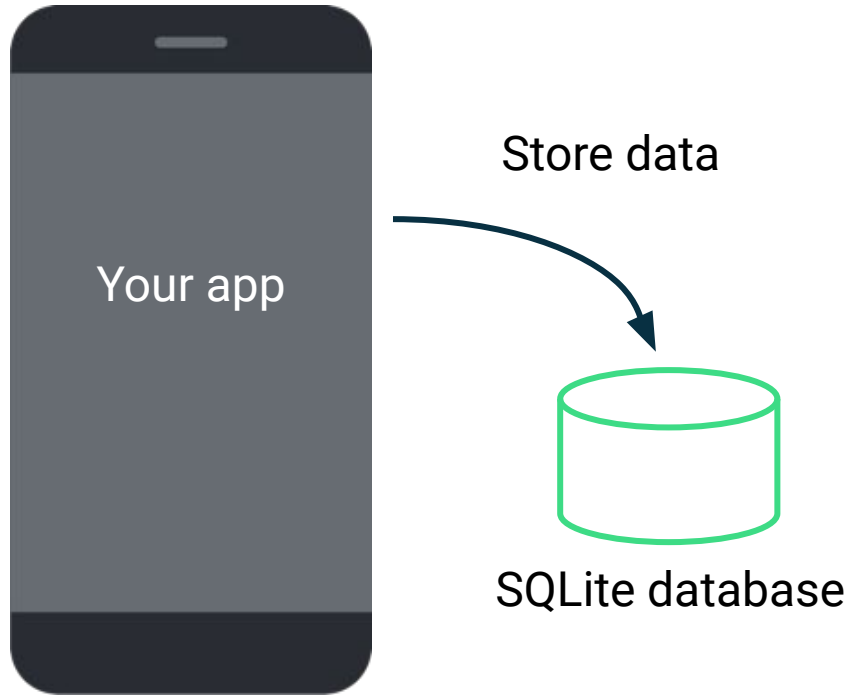


Structured Query Language (SQL)

Usiamo SQL per accedere e modificare un database relazionale

- Creare nuove tabelle
- Eseguire query per estrarre dati
- Inserire nuovi dati
- Aggiornare dati
- Eliminarli

SQLite in Android



SQLite in Android

Colors

_id	hex_color	name
1	#FF0000	red
2	#0000FF	blue
3	#FFBF00	amber

Esempio di istruzioni SQLite

Create

```
INSERT INTO colors VALUES ("#FF0000","red");
```

Read

```
SELECT * from colors;
```

Update

```
UPDATE colors SET hex_color="#DD0000" WHERE name="red";
```

Delete

```
DELETE FROM colors WHERE name = "red";
```

Interagire direttamente con un database

- Nessuna verifica della correttezza delle query a compile-time
- Richiede la scrittura di codice per convertire il risultato delle query SQL in oggetti
- Alcune librerie (ad esempio Room di Android Jetpack) facilitano lo storage, fornendo un livello di astrazione che ci permette di interagire direttamente con un database

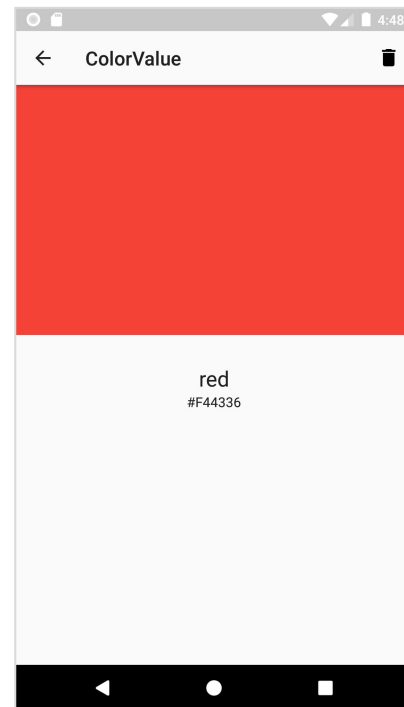
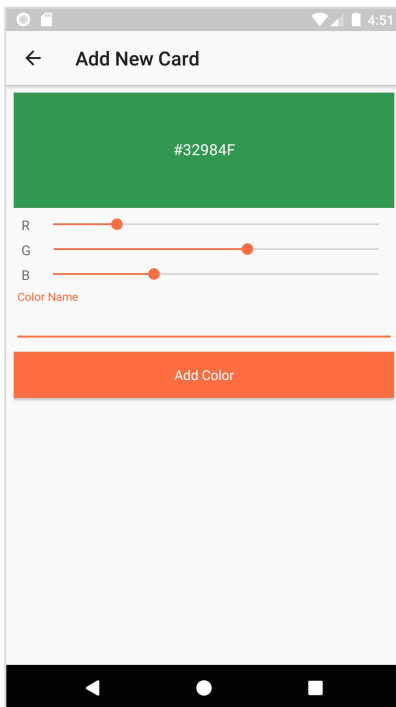
La libreria Room per la persistenza

Room

La libreria fornisce un livello di astrazione sopra SQLite per consentire un accesso al database più robusto, sfruttando tutta l'espressività del linguaggio di query

Funge da libreria per il “Object Relational Mapping” che converte dati dalla loro rappresentazione nel database in oggetti che possiamo manipolare da codice (o al contrario salvando oggetti, che erano stati creati da codice, nel database)

ColorValue app

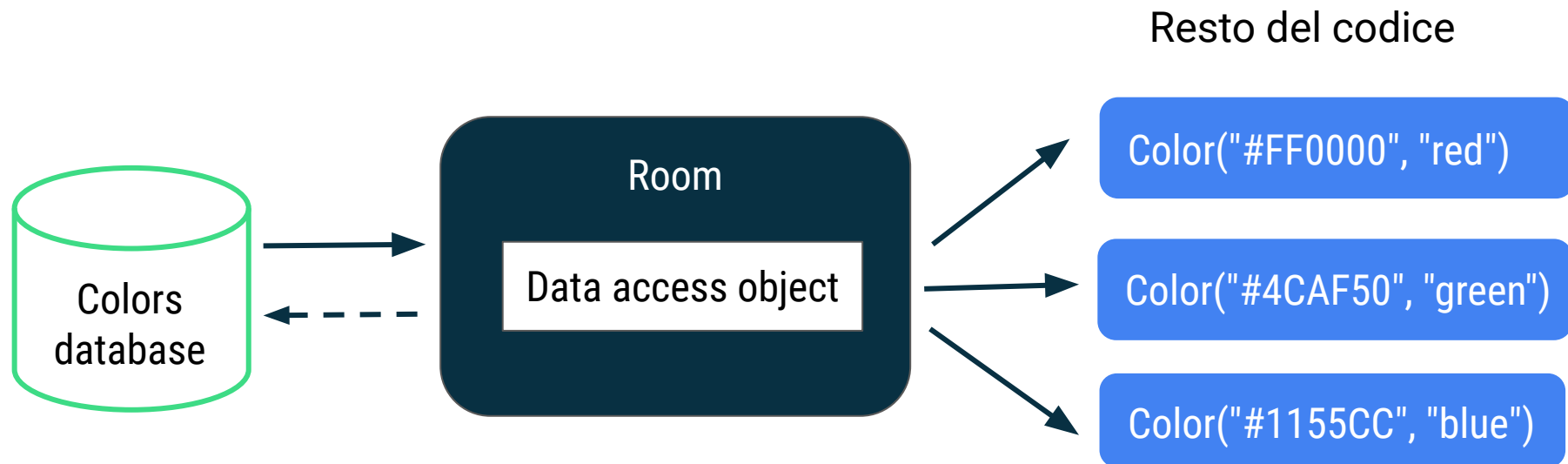


Aggiungiamo le dipendenze Gradle

```
dependencies {  
    implementation "androidx.room:room-runtime:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
  
    // Kotlin Extensions and Coroutines support for Room  
    implementation "androidx.room:room-ktx:$room_version"  
  
    // Test helpers  
    testImplementation "androidx.room:room-testing:$room_version"  
}
```

Nota: aggiungere `apply plugin: 'kotlin-kapt'` nel file Gradle

Room

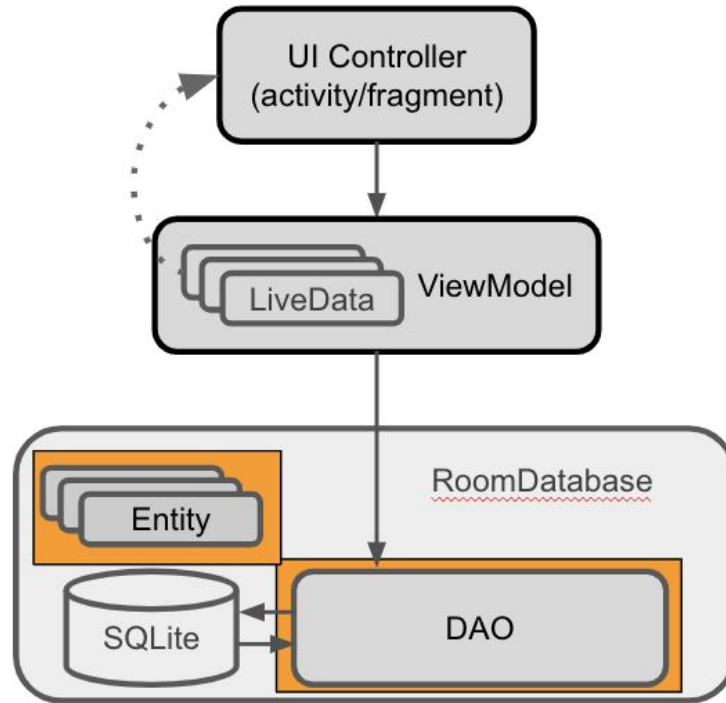


Room

Gestiremo tre principali componenti:

- Entity (rappresenta una tabella in un DB) `Color`
- DAO (metodi per accedere al DB) `ColorDao`
- Database (il gestore del DB) `ColorDatabase`

Room



Color class

```
data class Color {  
    val hex: String,  
    val name: String  
}
```

Annotazioni

- Forniscono ulteriori informazioni (metadati) al compilatore

`@Entity` marca le class Entity, `@Dao` per DAO, `@Database` per il database

- Possono avere parametri

```
@Entity(tableName = "colors")
```

- Possono autogenerare codice utile

Entity

Classe che viene mappata ad una tabella di un database SQLite

- `@Entity`
- `@PrimaryKey` (per marcare una colonna come PK)
- `@ColumnInfo` (di default, Room usa il nome della proprietà come nome di colonna nel DB. Questa annotazione serve per cambiare questo comportamento).

Entity: esempio

```
@Entity(tableName = "colors")  
data class Color {  
    @PrimaryKey(autoGenerate = true) val _id: Int,  
    @ColumnInfo(name = "hex_color") val hex: String,  
    val name: String  
}
```

colors
_id
hex_color
name

Entity: esempio

Altre annotazioni:

- `@Entity(primaryKeys = arrayOf("firstName", "lastName"))`
- `@Ignore val picture: Bitmap? // l'attributo non viene memorizzato su DB`
- `@Entity(indices = arrayOf(Index(value = ["last_name", "address"])))`
- ...

Data access object (DAO)

In Room si lavora con le classi DAO invece che accedere direttamente al database

- Le interazioni con il DB vengono specificate qui
- Un DAO va dichiarato come interfaccia o classe astratta
- Room crea le implementazioni DAO a compile-time
- Room verifica tutte le query DAO a compile-time

DAO: esempio

@Dao

interface ColorDao {

```
@Query("SELECT * FROM colors")  
fun getAll(): Array<Color>
```

@Insert

```
fun insert(vararg color: Color)
```

@Update

```
fun update(color: Color)
```

@Delete

```
fun delete(color: Color)
```

L'annotazione `@Query` serve per indicare che quel metodo eseguirà quella particolare query

Query

@Dao

```
interface ColorDao {
```

```
    @Query("SELECT * FROM colors")
```

```
    fun getAll(): Array<Color>
```

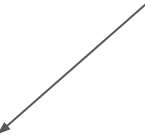
```
    @Query("SELECT * FROM colors WHERE name = :name")
```

```
    fun getColorByName(name: String): LiveData<Color>
```

```
    @Query("SELECT * FROM colors WHERE hex_color = :hex")
```

```
    fun getColorByHex(hex: String): LiveData<Color>
```

I parametri del metodo
possono essere usati per
parametrizzare la query



Insert

```
@Dao  
interface ColorDao {  
    ...
```

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
```

```
fun insert(vararg color: Color)
```

```
    ...
```

```
}
```

Si realizza un conflitto quando ad esempio si cerca di inserire un record già esistente. Esistono varie strategie di gestione dei conflitti (https://sqlite.org/lang_conflict.html)

Con vararg posso indicare un numero variabile di parametri passati, per effettuare inserimenti multipli

Update

```
@Dao
interface ColorDao {
    ...

    @Update
    fun update(color: Color)

    ...
}
```

Delete

```
@Dao
interface ColorDao {
    ...

    @Delete
    fun delete(color: Color)

    ...
}
```

Altre annotazioni

`@Transaction`

```
fun insertAndDeleteInTransaction(newColor: Color, oldColor: Color) {  
    // Anything inside this method runs in a single transaction.  
    insert(newColor)  
    delete(oldColor)  
}  
}
```

Creare un database Room

- Annotiamo una classe con `@Database` e includiamo una lista di entity (tabelle):

```
@Database(entities = [Color::class], version = 1)
```

- Dichiariamo una classe astratta che estende `RoomDatabase`:

```
abstract class ColorDatabase : RoomDatabase() {
```

- Dichiariamo un metodo astratto senza argomenti che ritorna il DAO:

```
abstract fun colorDao(): ColorDao
```

Room database: esempio

```
@Database(entities = [Color::class], version = 1)
```

```
abstract class ColorDatabase : RoomDatabase() {
```

```
    abstract fun colorDao(): ColorDao
```

```
    companion object {
```

```
        @Volatile
```

```
        private var INSTANCE: ColorDatabase? = null
```

```
        fun getInstance(context: Context): ColorDatabase {
```

```
            ...
```

```
        }
```

```
    }
```

```
    ...
```

Il companion object serve per dichiarare che l'oggetto Database è singleton (cioè ne esisterà un'unica istanza)

L'annotazione @Volatile indica che ogni modifica all'attributo INSTANCE è immediatamente visibile a tutti gli altri thread dell'applicazione (<<the value is never cached>>)

Creare un'istanza di database

```
fun getInstance(context: Context): ColorDatabase {  
    return INSTANCE ?: synchronized(this) {  
        INSTANCE ?: Room.databaseBuilder(  
            context.applicationContext,  
            ColorDatabase::class.java, "color_database"  
        )  
        .fallbackToDestructiveMigration()  
        .build()  
        .also { INSTANCE = it }  
    }  
}
```

Significa
che se più
thread
chiamano
questo
metodo
insieme,
vengono
comunque
eseguiti in
serie

Ottenere ed usare un DAO

Prendiamo un riferimento al DAO dal database:

```
val colorDao =  
ColorDatabase.getInstance(application).colorDao()
```

Creiamo un nuovo Color e usiamo il DAO per inserirlo nel DB:

```
val newColor = Color(hex = "#6200EE", name = "purple")  
colorDao.insert(newColor)
```

Laboratorio

Costruire un database ed un'app per scrivere e leggere informazioni su degli studenti.

- L'app può avere un'unica Activity senza viewmodel (per semplicità)
- L'app deve inserire uno studente nel DB
- Poi deve leggerlo e visualizzare su Log o a video le informazioni

Programmazione asincrona

Long-running tasks

- Download di informazioni
- Sincronizzare dati con un server
- Scrivere su file
- Calcolo pesante
- Leggere o scrivere da/su un database

Necessità della programmazione asincrona

- Il tempo per eseguire un task è limitato:
 - le componenti grafiche devono essere aggiornate in un tempo $< 16\text{ms}$ per garantire un refresh rate di 60 frame per secondo. Se ci impiega di più il sistema dovrà scartare alcuni frame (si parla di “jank”)
- Alcuni task richiedono necessariamente del tempo, malgrado tutte le ottimizzazioni che possiamo fare
- Dobbiamo controllare il “come” ed il “dove” un task viene eseguito
 - ad esempio in quali casi un task può interrompersi

Necessità della programmazione asincrona

```
button.setOnClickListener() {  
    killSomeTime()  
}  
  
private fun killSomeTime() {  
    for (i in 1..20) {  
        textView.text = i.toString()  
        println("i: $i")  
        Thread.sleep(2000)  
    }  
}
```

Malgrado possa sembrare che la UI venga aggiornata ogni 2 secondi, in realtà il Runtime sta scartando frames per via dell'istruzione bloccante `Thread.sleep(2000)`.

Dunque occorre spostare la computazione su un Thread separato.

Programmazione asincrona su Android

- Threading
- Callbacks
- Molte altre opzioni sono possibili

Qual è l'approccio consigliato?

Thread e Runnable

Classi che provengono da librerie Java. Questo approccio prevede di creare un thread separato per eseguire task onerosi.

- Si deve creare una classe che implementa l'interfaccia `Runnable()`, con la computazione dentro il metodo `run()`
- Passare un'istanza della classe alla classe `Thread`
- Chiamare `start()` dell'oggetto thread

Problema: nessun thread può modificare l'UI se non il main thread

Thread e Runnable

```
button.setOnClickListener() {  
    val runnable = Worker()  
    val thread = Thread(runnable)  
    thread.start()  
}  
  
inner class Worker: Runnable {  
    override fun run() {  
        killSomeTime()  
    }  
}
```

```
private fun killSomeTime() {  
    for (i in 1..20) {  
        println("i: $i")  
        Thread.sleep(2000)  
    }  
}
```

Altro approccio: libreria Anko

La libreria Anko di JetBrains include **doAsync**, una funzione di gestione di task asincroni:

- Aggiungere `implementation 'org.jetbrains.anko:anko-common:$version'` alle dipendenze su Gradle
- Inserire il codice da eseguire in modo asincrono in `doAsync { ... }`
- L'approccio di Anko per modificare l'UI è quello di prevedere una sezione `activityUiThread { ... }` dentro `doAsync` che viene eseguita sul thread principale

Altro approccio: libreria Anko

```
button.setOnClickListener {
```

```
    doAsync{
```

```
        for(i in 1..20){
```

```
            Thread.sleep(2000)
```

```
            activityUiThread{
```

```
                textView.text = i.toString()
```


```
            }
```

```
        }
```

```
    }
```

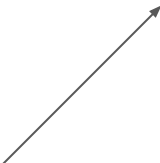
```
}
```

Questa porzione viene eseguita sul thread principale ed in questo modo è possibile manipolare l'interfaccia grafica



Approccio con le callback

```
class ViewModel: ViewModel() {  
    fun fetchDocs() {  
        get("developer.android.com") { result -> show(result) }  
    }  
}
```



Secondo questo approccio lancio la chiamata a una funzione (get nell'esempio) dal thread principale, ma di fatto l'esecuzione della funzione viene affidata ad un altro thread. Quando questo termina, la funzione di callback viene chiamata di nuovo sul thread principale (il metodo show)

Coroutine

Coroutine

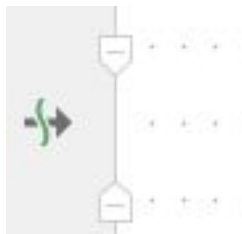
- Introdotte in Kotlin 1.3, mantengono l'app reattiva mentre vengono eseguiti dei long-running tasks.
- Semplificano il codice per gestire i task asincroni
 - Il codice può essere scritto in modo **sequenziale**
 - Le eccezioni possono essere gestite con blocchi `try/catch`

Benefici delle coroutine

- Approccio leggero computazionalmente
- Un numero minore di memory leaks
- Supporto automatico alla cancellazione dei task
- Integrazione con Jetpack

Funzioni “suspend”

- Aggiungere la keyword `suspend` affinché possa essere sospesa. Così la funzione può essere chiamata solo da una coroutine o da altre funzioni `suspend`



The diagram shows a vertical line representing a coroutine context. On the left, there is a green arrow pointing right, and two small white boxes with minus signs. Dotted lines connect these elements to the code snippet on the right.

```
suspend fun insert(word: Word) {  
    wordDao.insert(word)  
}
```

Suspend: esempio

```
suspend fun fetchDocs() {  
    val docs = get("url")  
    show(docs)  
}
```

```
suspend fun get(url: String) =  
    withContext(Dispatchers.IO) { ... }
```

Le coroutine forniscono un meccanismo per rendere non bloccante una funzione. Quando `get` ha terminato, `fetchDocs` riprende (resume) da dove era rimasta.

Suspend e resume

Due funzioni sono usate con le coroutine per rimpiazzare le callback:

- **Suspend**: quando una coroutine chiama una funzione suspend, invece di attendere il suo termine (e bloccare l'app) si sospende finché la funzione (che girerà su un altro thread) non ha terminato l'esecuzione (salvando le variabili locali)-> così non blocca l'esecuzione dell'applicazione
- **Resume**: al termine dell'esecuzione della funzione suspend, la coroutine si riattiva, carica automaticamente lo stato salvato e continua l'esecuzione dal punto in cui il codice era stato sospeso

Esempio

Quando una coroutine viene sospesa, lo stack viene copiato e messo da parte finché la funzione non viene riesumata.

Quando le coroutine del thread principale sono sospese, il thread può andare avanti a gestire la UI (ad esempio ad aggiornare delle view, a rilevare eventi, etc...)



Aggiungere suspend ai metodi DAO

```
@Dao
interface ColorDao {

    @Query("SELECT * FROM colors")
    suspend fun getAll(): Array<Color>

    @Insert
    suspend fun insert(vararg color: Color)

    @Update
    suspend fun update(color: Color)

    @Delete
    suspend fun delete(color: Color)
```

Controllare dove gireranno le coroutine

Le coroutine possono essere eseguite sul thread principale (se non sono dispendiose) o su un altro. Kotlin ha diversi “dispatcher” per indicare dove la coroutine va eseguita.

Dispatcher	Description of work	Examples of work
<code>Dispatchers.Main</code>	UI and nonblocking (short) tasks	Updating LiveData, calling suspend functions
<code>Dispatchers.IO</code>	Network and disk tasks	Database, file IO
<code>Dispatchers.Default</code>	CPU intensive	Parsing JSON

Uso di withContext per scegliere il Dispatcher

```
suspend fun get(url: String) {  
  
    // Start on Dispatchers.Main by default  
  
    withContext(Dispatchers.IO) {  
        // Switches to Dispatchers.IO  
        // Perform blocking network IO here  
    }  
  
    // Returns to Dispatchers.Main  
}
```

Lancio di una coroutine: CoroutineScope

Le coroutine devono essere lanciate all'interno di uno speciale contenitore, il `CoroutineScope`:

- Tiene traccia delle coroutine lanciate al suo interno (anche quelle in stato di sospensione)
- Consente di cancellare le coroutine al suo interno
- Fornisce un bridge tra le normali funzioni e le coroutine

Esempi: `GlobalScope` (default)

`ViewModel` ha `viewModelScope`

`Lifecycle` ha `lifecycleScope`

Come lanciare una coroutine

- `launch` - se non ho bisogno di risultati

```
fun loadUI() {  
    launch {    //qui uso lo scope di default  
        fetchDocs()  
    }  
}
```

- `async` - può ritornare dei risultati tramite una funzione `await` (che però non può essere chiamata dalle normali funzioni, quindi è quasi sempre meglio eseguire `launch`)


Come lanciare una coroutine

- `launch` - in altri termini, `launch` è un bridge tra il mondo delle funzioni normali e quello delle coroutine

Il posto migliore dove lanciarla è il `viewModel`. Per farlo però occorre aggiungere:

ViewModelScope

```
class MyViewModel: ViewModel() {  
  
    init {  
        viewModelScope.launch {  
            // Coroutine that will be canceled  
            // when the ViewModel is cleared  
        }  
    }  
    ...  
}
```



Nota: un `viewModelScope` viene creato automaticamente per ciascun `viewModel`. Utile legare la vita della coroutine al `viewModel`, perché così se questo termina lo faranno anche le `suspend function` lanciate dalla coroutine

Esempio viewModelScope

```
class ColorViewModel(val dao: ColorDao, application: Application)
    : AndroidViewModel(application) {

    fun save(color: Color) {
        viewModelScope.launch {
            colorDao.insert(color)
        }
    }

    ...
}
```

Testing databases

Add Gradle dependencies

```
android {  
    defaultConfig {  
        ...  
        testInstrumentationRunner "androidx.test.runner  
            .AndroidJUnitRunner"  
        testInstrumentationRunnerArguments clearPackageData: 'true'  
    }  
}  
  
dependencies {  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'  
    androidTestImplementation  
'androidx.test.espresso:espresso-core:3.1.1'  
}
```

Testing Android code

- `@RunWith (AndroidJUnit4::class)`
- `@Before`
- `@After`
- `@Test`

Create test class

```
@RunWith(AndroidJUnit4::class)
```

```
class DatabaseTest {
```

```
    private lateinit val colorDao: ColorDao
```

```
    private lateinit val db: ColorDatabase
```

```
    private val red = Color(hex = "#FF0000", name = "red")
```

```
    private val green = Color(hex = "#00FF00", name = "green")
```

```
    private val blue = Color(hex = "#0000FF", name = "blue")
```

```
    ...
```


Create and close database for each test

In DatabaseTest.kt:

`@Before`

```
fun createDb() {  
    val context: Context = ApplicationProvider.getApplicationContext()  
    db = Room.inMemoryDatabaseBuilder(context, ColorDatabase::class.java)  
        .allowMainThreadQueries()  
        .build()  
    colorDao = db.colorDao()  
}
```

`@After`

```
@Throws(IOException::class)  
fun closeDb() = db.close()
```

Test insert and retrieve from a database

In DatabaseTest.kt:

```
@Test
@Throws(Exception::class)
fun insertAndRetrieve() {
    colorDao.insert(red, green, blue)
    val colors = colorDao.getAll()
    assert(colors.size == 3)
}
```

Summary

Riassunto

In questa lezione abbiamo imparato:

- A configurare un database usando la libreria Room
- Ad usare le coroutine per la programmazione asincrona
- Ad usare le coroutine con Room
- A testare un database

Links

- [7 Pro-tips for Room](#)
- [Room Persistence Library](#)
- [SQLite Home Page](#)
- [Save data using SQLite](#)
- [Coroutines Guide](#)
- [Dispatchers - kotlinx-coroutines-core](#)
- [Coroutines on Android \(part I\): Getting the background](#)
- [Coroutines on Android \(part II\): Getting started](#)
- [Easy Coroutines in Android: viewModelScope](#)
- [Kotlin Coroutines 101](#)