SENG440

Week 5

Persisting data

Ben Adams (benjamin.adams@canterbury.ac.nz)

Data persistence

- Internal storage
  - Small and private to the app.
  - When app is uninstalled, internal files are removed.
- `SharedPreferences` and `DataStore`
  - Stores lightweight, cookie-like information
  - Key, value pair data.
- External storage and shared storage
  - Large and possibly shared with other apps.
  - When an app is uninstalled, shared files are not removed.
- SQLite database
  - Supports storage and querying of complex information
  - **Room** for object-relational mapping

Bundle

- Bundles are used to transfer information from one Activity to another
- For dealing with rotation, etc. can be used to maintain session state

```
// This callback is called only when there is a saved instance that is previously saved by usi
// onSaveInstanceState(). We restore some state in onCreate(), while we can optionally restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    textView.text = savedInstanceState?.getString(TEXT_VIEW_KEY)
}

// invoked when the activity may be temporarily destroyed, save the instance state here
override fun onSaveInstanceState(outState: Bundle?) {
    outState?.run {
        putString(GAME_STATE_KEY, gameState)
        putString(TEXT_VIEW_KEY, textView.text.toString())
    }
    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState)
}
```

Internal storage directories

- Write file using `Context.MODE_PRIVATE`
- Only accessible to the app, not other apps
- Android 10 and higher, directory location is encrypted
- Security API can be used to encrypt files in older versions

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

Static files

- If you need or have a file with a lot of data at compile time:
  - Create and save file in project `res/raw/` directory
  - Open file using the `openRawResource(id: Int)` method and pass the `R.raw.`*`id`* of file
  - Returns an `InputStream` to read from file
  - Cannot write to the file. It is part of the APK

Cache files

- If you need to cache data for application instead of storing persistently:
  - Call `getCacheDir()` method to obtain a `File!` object that is a directory where you can create and save temporary cache files
  - Files may be deleted by Android later if space needed but you should clean them up on your own
  - Recommended to keep under 1 MB

Other useful methods for internal files

- All inherited from Context
- **`getFilesDir`**`(): File!`
  - get absolute path to filesystem directory where app files are saved
- **`getDir`**`(name: String!, mode: Int): File!`
  - get and create if necessary a directory for files
- **`deleteFile`**`(name: String!): Boolean`
  - get rid of files, especially cache files
- **`fileList`**`(): Array<String!>!`
  - get an array of Strings with files associated with Context (application)

SharedPreferences

- Used to save key-value data for your app, e.g. a high-score for a game
- Can have one or multiple SharedPreferences files
- If you are are specifically persisting a small set of key-value pairs, SharedPreferences are a better choice than files

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE)
```

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

Jetpack DataStore

- Improvement on SharedPreferences that uses co-routines to store data asynchronously *(more on co-routines later)*

- **Preferences DataStore** – **key, value storage** like SharedPreferences
- **Proto DataStore** - stores data as custom data types **using protobuf**

More info:
https://developer.android.com/topic/libraries/architecture/datastore

Accessing external storage

- **`getExternalFilesDir`**`(type: String?): File?`
  - Primary *shared/external* storage device where the application can place persistent files it owns
  - No security enforced, can be overwritten by any app with `android.Manifest.permission#WRITE_EXTERNAL_STORAGE`
- Type indicates sub-directory where it is stored:
  - `null`, root directory
  - Otherwise, string constant `Environment.DIRECTORY_{type}`, where type is:
    - `ALARMS, AUDIOBOOKS, DCIM, DOCUMENTS, DOWNLOADS, MOVIES, MUSIC, NOTIFICATIONS, PICTURES, PODCASTS, RINGTONES, SCREENSHOTS`

Shareable storage directories

- Media content not owned by your app is saved in public directory and accessed using the `MediaStore` API
  - Depending on version of Android might require giving the app external storage permissions
  - `<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>`

Access media files from shared storage

- Other documents (e.g. PDF files) stored in other public directories are accessed using a document provider
  - App invokes an `Intent` to create, open, etc. a file

Access documents and other files from shared storage

Object-Relational Mapping (ORM)

- Automated mapping of our models (object-oriented classes) in code to relational database tables
  - **Room in Android**
  - ActiveRecord in Ruby on Rails
  - Hibernate in Java
  - Core Data in iOS
- **Annotate our models** that we want as tables with @Entity
- Create an interface for a **data access object (DAO)**–bridges between database type and the host language type. DAO for each model.
- Create an interface for the **database**, which imposes getters for each model's DAO

Room dependencies

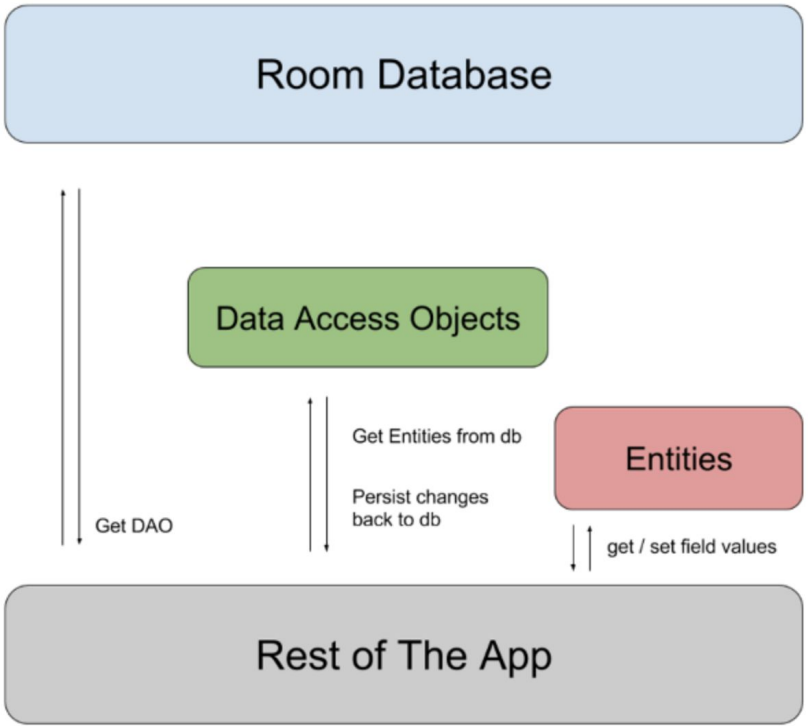
```
dependencies {
  def room_version = "2.2.6"

  implementation "androidx.room:room-runtime:$room_version"
  kapt "androidx.room:room-compiler:$room_version"

  // optional - Kotlin Extensions and Coroutines support for Room
  implementation "androidx.room:room-ktx:$room_version"

  // optional - Test helpers
  testImplementation "androidx.room:room-testing:$room_version"
}
```

Room architecure



Entity

- Each entity will be a row in the database
- Use annotations (@) to indicate mapping from model to DB

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Entity

- Each entity will be a row in the database
- Use annotations (@) to indicate mapping from model to DB
  - Can take parameters in parenthesis (e.g., tableName)

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Entity

- Each entity will be a row in the database
- Use annotations (@) to indicate mapping from model to DB
  - Can take parameters in parenthesis (e.g., tableName)
  - Will add fields by default, but can ignore columns, etc. More info here.

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    val lastName: String?,
    @Ignore val picture: Bitmap?
)
```

Data Access Object (DAO)

- An interface that defines how method names map to SQL commands

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Database

- Sub-class of `RoomDatabase`, that has a method that returns the DAO that you have defined

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Using Database

- Use Room database builder to create a database instance

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, "database-name"
        ).build()
```

- Access DAO methods to interact with database

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

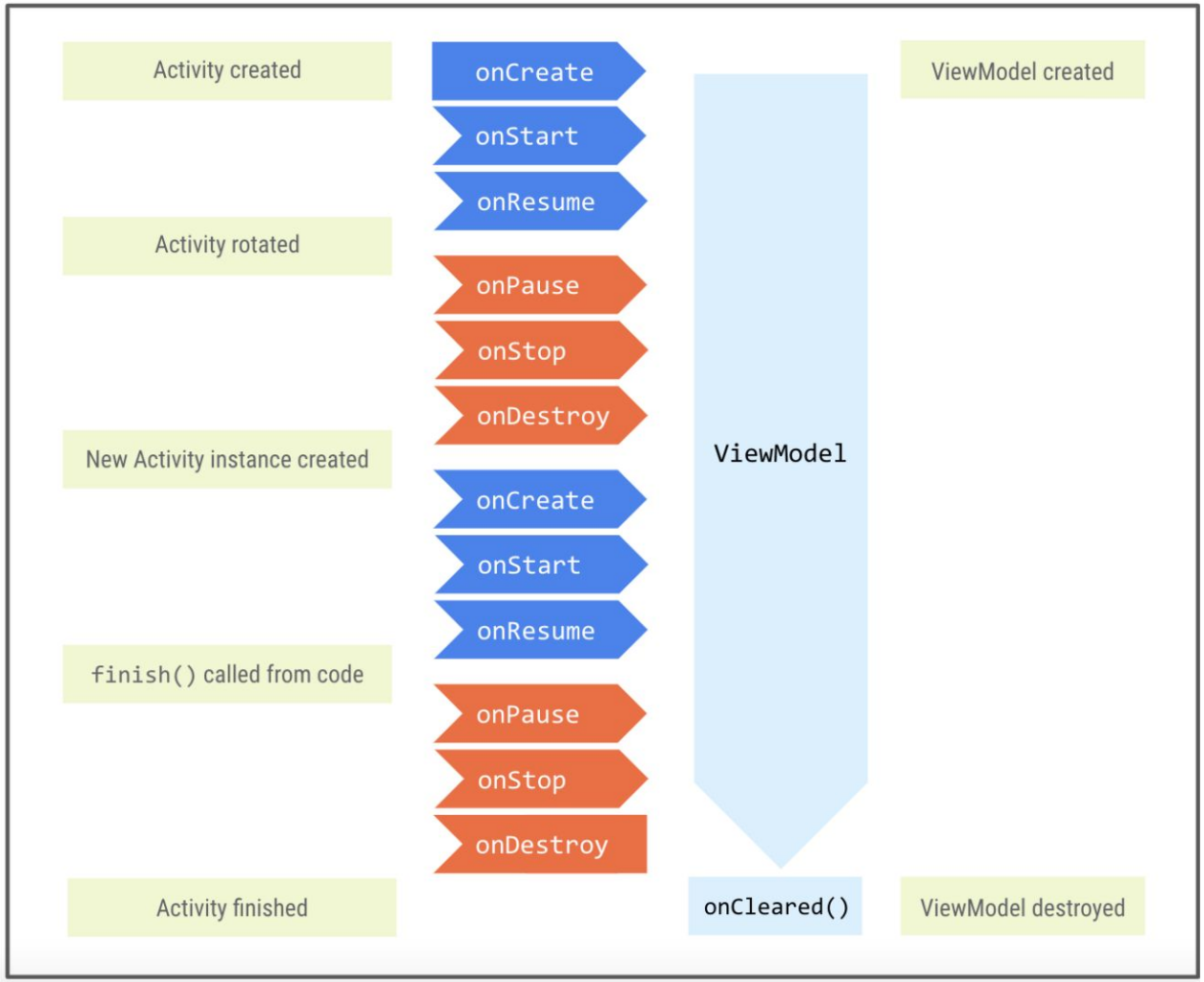
ViewModel

- Helper class to prepare data for the UI
- Lifecycle aware
- Retained during configuration changes
  - E.g. orientation changes

```
// ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0'
```

- Activity can have a `ViewModel` to share data between fragments. Attach view model to the fragment.

```
private val viewModel: GameViewModel by viewModels()
```


Activity created
onCreate
onStart
onResume
ViewModel created
Activity rotated
onPause
onStop
onDestroy
New Activity instance created
ViewModel
onCreate
onStart
onResume
finish() called from code
onPause
onStop
onDestroy
Activity finished
onCleared()
ViewModel destroyed

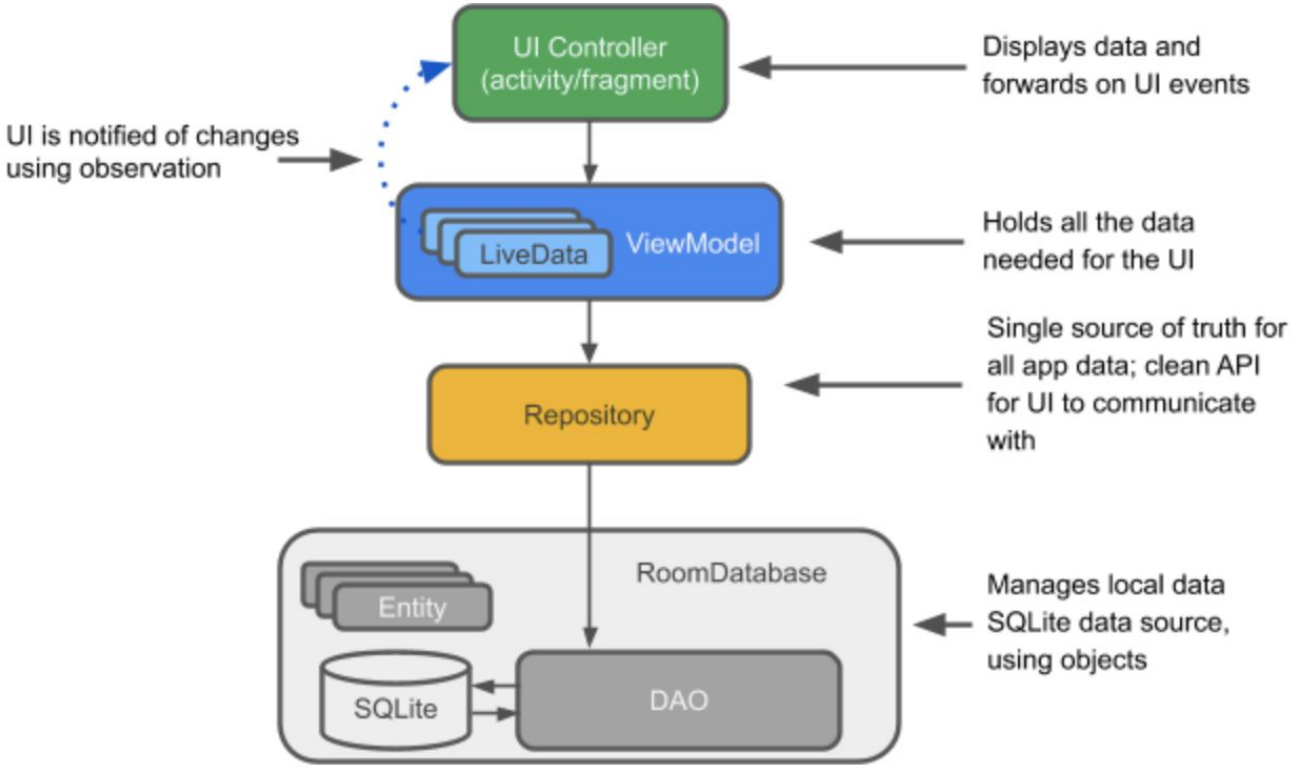
Backing property in View model

- Allows you to return something from a getter other than the exact object.
- Override the getter method to return a read-only version of your data

```
// Declare private mutable variable that can only be modified
// within the class it is declared.
private var _count = 0

// Declare another public immutable field and override its getter method.
// Return the private property's value in the getter method.
// When count is accessed, the get() function is called and
// the value of _count is returned.
val count: Int
   get() = _count
```

Integrating LiveData and Room



Observing live data in Compose

- As seen in Week 3 tutorial, `MutableState` types used to observe data changes and recompose the view
- Can also map `LiveData` observable to compose state:
  - Add new gradle dependency
  - `androidx.compose.runtime:runtime-livedata`

```
import androidx.compose.material.Text
import androidx.compose.runtime.livedata.observeAsState

val value: String? by liveData.observeAsState()
Text("Value is $value")
```

Reading

- Data and File Storage Overview
- Access app-specific files
- ViewModel Overview
- LiveData Overview