Mobile Programming

Storage & DB

I/O

- App state information
- App preference information
- App data

App (UI) State (1/5)

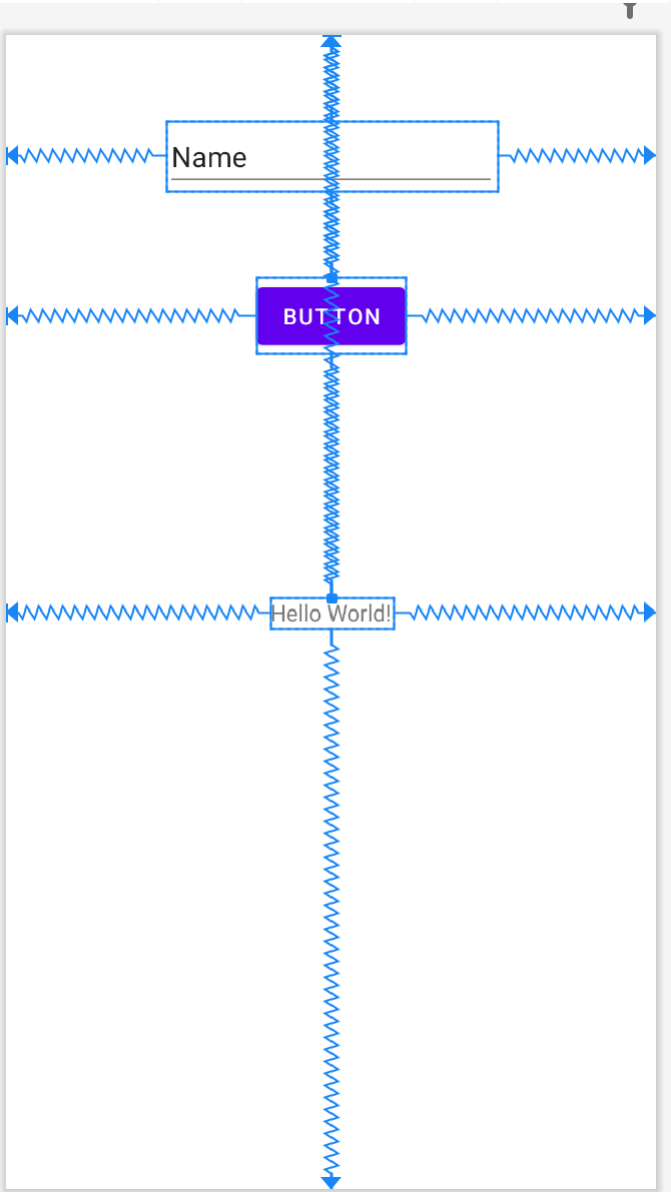
- A user expects an activity's UI state to remain the same throughout configuration changes
  - E.g., rotation or switching into multi-window mode, etc.

- The system destroys the activity by default when such a configuration change occurs, wiping away any UI state stored in the activity instance!
  - The original activity instance will have the following callbacks triggered:
    - onPause()
    - onStop()
    - onDestroy()
  - A new instance of the activity will be created and have the following callbacks triggered:
    - onCreate()
    - onStart()
    - onResume()

App (UI) State (2/5)

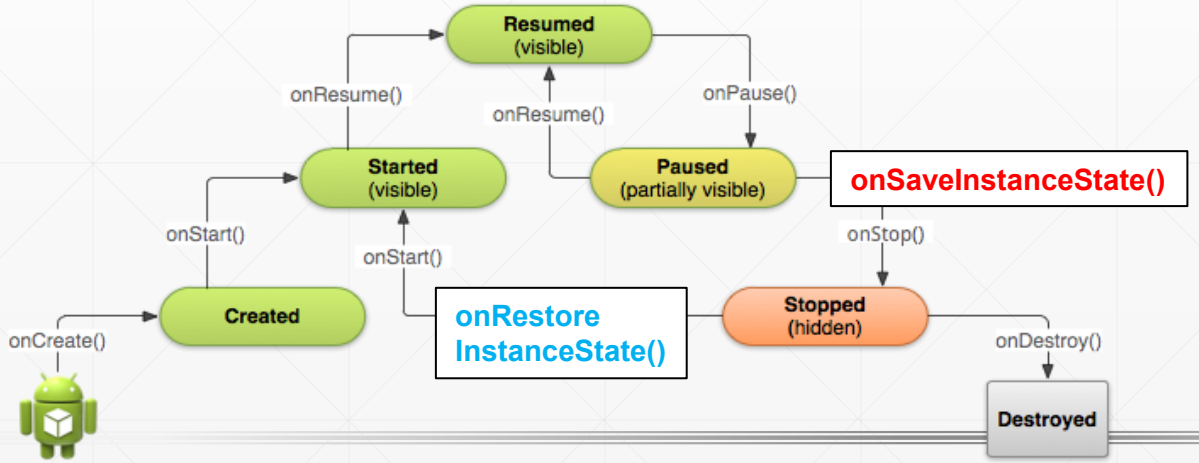
- Example) update your application to …
  - 1) display the address when a user touches a button!
  - 2) have an editText field

```
lateinit var location:String
…

binding.button.setOnClickListener(){
    when {
        ContextCompat.checkSelfPermission  ( this, "android.permission.ACCESS_FINE_LOCATION"  ) ==
PERMISSION_GRANTED -> {
            val locationManager = getSystemService(LOCATION_SERVICE) as LocationManager
            location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER).toString()
            Toast.makeText(this, location,  Toast.LENGTH_LONG).show()
            binding.textView.text=location
        }
…
```



App (UI) State (3/5)

- onSaveInstanceState(outState: Bundle)
  - When the activity is destroyed due to system constraints, we can preserve the user's transient UI state using this callback
- onRestoreInstanceState (savedInstanceState: Bundle)
  - When your activity is recreated after it was previously destroyed, you can recover your saved instance state from the Bundle that the system passes to your activity
  - Both the onCreate() and onRestoreInstanceState() callback methods receive the same Bundle that contains the instance state information



App (UI) State (4/5)

- onSaveInstanceState(outState: Bundle)
  - When the activity is destroyed due to **system constraints**, we can preserve the user's transient UI state using this callback
    - This callback does not fire when a user explicitly closes an activity (e.g., back gesture)
  - Use this when your data is simple/lightweight, such as a primitive data type or a simple object
  - Add key-value pairs to the Bundle object that is saved
    - Use various setter methods for the Bundle instance (Reference)

```
override fun onSaveInstanceState(outState: Bundle) {
    outState.run {
        putString("location", location)
    }
    Log.d("ITM", "onSave called!")
    super.onSaveInstanceState(outState)
}
```

App (UI) State (5/5)

- onRestoreInstanceState(savedInstanceState: Bundle)
  - The system automatically saves/restores the state of the view hierarchy
  - You can retrieve UI state information from Bundle object
    - Use various getters for Bundle instance (Reference)

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    location = savedInstanceState.getString("location").toString()
    binding.textView.text = savedInstanceState.getString("location")
}
```

  - override fun onCreate(savedInstanceState: Bundle?)
    - savedInstanceState can be null in the onCreate() callback
    - Therefore, you must check whether the state Bundle is null before you attempt to read it

Storage Overview (1/4)

	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, getFilesDir() or getCacheDir() From external storage, getExternalFilesDir() or getExternalCacheDir()	Never needed for internal storage Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher	No	Yes
Media	Shareable media files (images, audio files, videos)	MediaStore API	READ_EXTERNAL_STORAGE when accessing other apps' files on Android 11 (API level 30) or higher READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE when accessing other apps' files on Android 10 (API level 29) Permissions are required for all files on Android 9 (API level 28) or lower	Yes, though the other app needs the READ_EXTERNAL_STORAGE permission	No
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes, through the system file picker	No
App preferences	Key-value pairs	SharedPreference API	None	No	Yes
Database	Structured data	Room persistence library	None	No	Yes

Storage Overview (2/4)

- App-specific storage
  - Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within an external storage
  - Use the directories within internal storage to save sensitive information
- Shared storage
  - Store files that your app intends to share with other apps, including media, documents, and other files
- Preferences
  - Store private, primitive data in key-value pairs
- Databases
  - Store structured data in a private database using the Room persistence library

Storage Overview (3/4)

- How much space does your data require?
  - *Internal storage has limited space* for app-specific data
- How reliable does data access need to be?
  - If your app's basic functionality requires certain data, *place the data within internal storage* directory or a database!
  - App-specific files that are stored in external storage are not always accessible because some devices allow users to remove a physical device that corresponds to external storage

Storage Overview (4/4)

- What kind of data do you need to store?
  - If you have data that's only meaningful for your app, use app-specific storage!
  - For shareable media content, use shared storage so that other apps can access the content!
  - For structured data, use either preferences (for key-value data) or a database (for data that contains more than 2 columns)!
- Should the data be private to your app?
  - When storing sensitive data, use internal storage, preferences, or a database!
  - Internal storage has the added benefit of the data being hidden from users

App Preference (1/4)

- If you have a relatively small collection of key-values to save, you should use the SharedPreferences APIs!
  - SharedPreference points to a file containing key-value pairs and provides simple methods to read and write them

- getPreferences(int mode)
  - Use this from an Activity if you need to use only one shared preference file for the activity
  - Mode: Context.MODE_PRIVATE is default
- getSharedPreferences(String name, int mode)
  - Use this if you need multiple shared preference files identified by ***name***
  - Mode: Context.MODE_PRIVATE is default

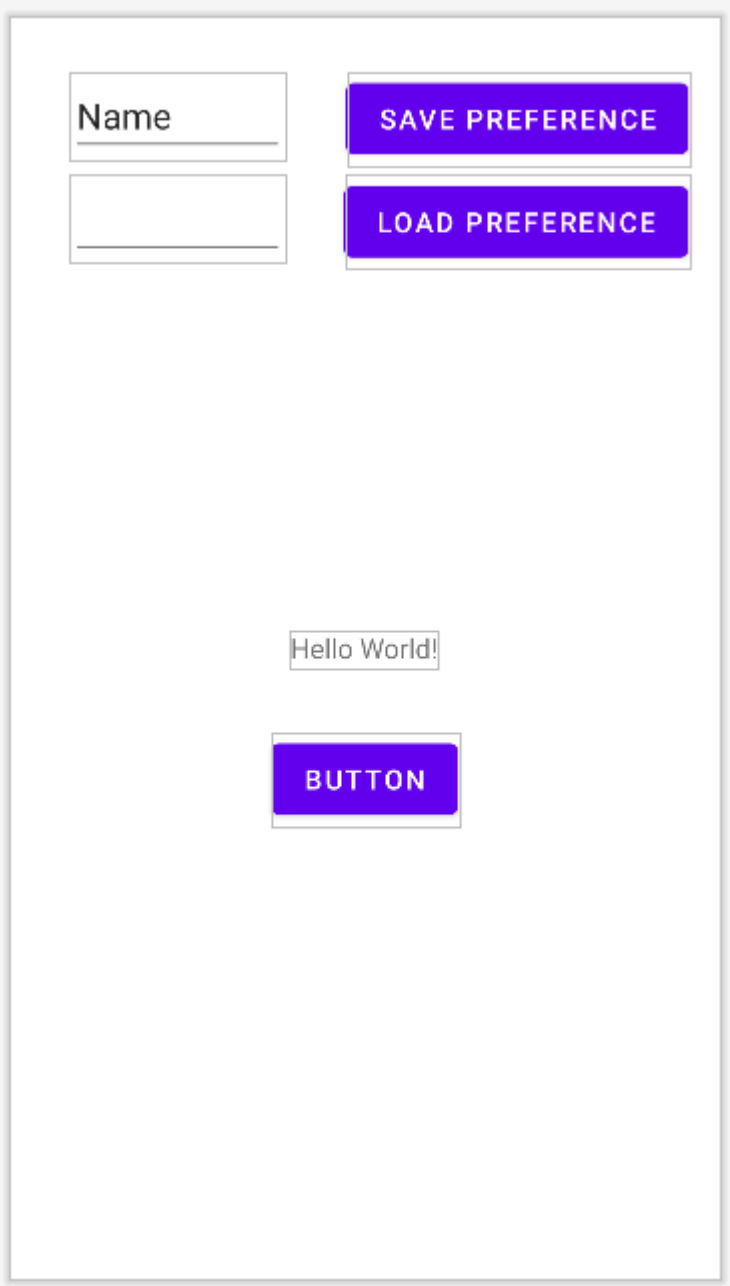
App Preference (2/4)

- Writing to a sharedPreference
  - Use SharedPreferences.Editor by calling edit() on your SharedPreferences
    - Setters: put[types]() methods (Takes key-value pairs)
      - putBoolean(), putFloat(), putInt(), etc.
      - remove()
    - Commit(): Commit your preferences changes
    - Apply(): Commits its changes to the in-memory SharedPreferences immediately, then starts an asynchronous commit to disk
  - More on the Reference

App Preference (3/4)

- Writing to a sharedPreference
  - Example)

```
fun savePreference(view: View){
   val sharedPref = getPreferences(Context.MODE_PRIVATE) ?: return
   with (sharedPref.edit()) {
      putInt("ITMCode", binding.editTextNumber.text.toString().toInt() )
      putString("ITMGrade",binding.editTextTextPersonName.text.toString())
      apply()
   }
   Toast.makeText(this,"preference saved",Toast.LENGTH_SHORT).show()
   binding.editTextNumber.text.clear()
   binding.editTextTextPersonName.text.clear()
}
```



  - Use UI editor to connect savePreference() to the button
    - onClick property

App Preference (4/4)

- Reading from the sharedPreference
  - Call methods such as getInt() and getString(), etc.
    - Key: the name of the preference to retrieve
    - You can set a default value to return if the preferent does not exist
  - More on the Reference
  - Example)

```
fun loadPreference(view: View) {
    val sharedPref = getPreferences(Context.MODE_PRIVATE) ?: return
    Snackbar.make(
        binding.root,
        "my Code was ${sharedPref.getInt("ITMCode", 0)} and my Grade was ${
            sharedPref.getString(
                "ITMGrade",
                "F"
            )
        }",
        Snackbar.LENGTH_LONG
    ).show()
}
```

App Preference

- Note
  - DataStore is a modern data storage solution that you should use instead of SharedPreferences
  - DataStore builds on Kotlin coroutines and Flow, and overcomes many of the drawbacks of SharedPreferences
  - Read the DataStore guide for more information

App-specific Data: Internal Storage (1/4)

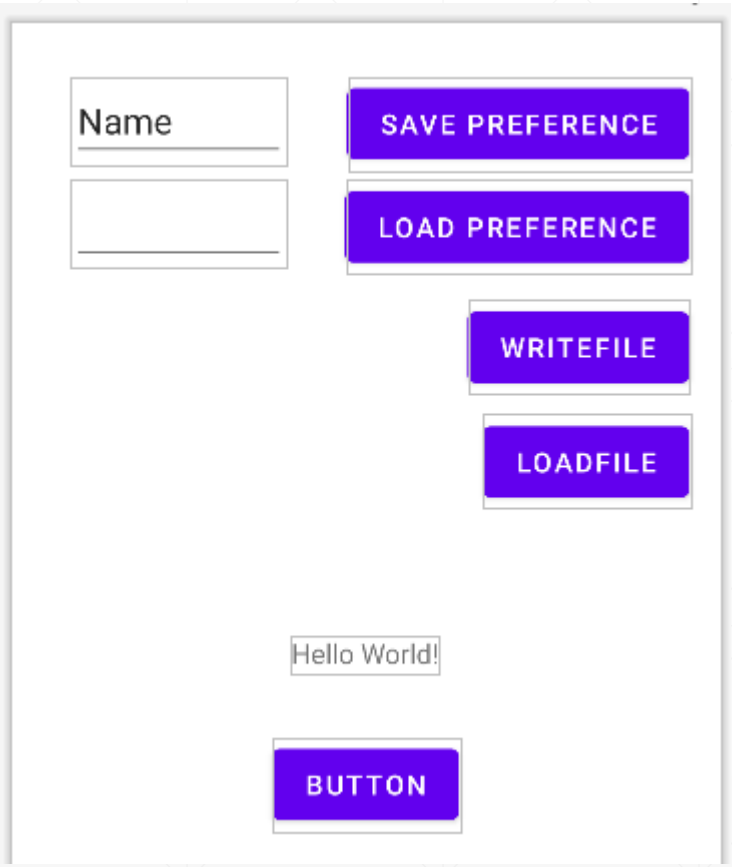
- For each app, the system provides directories within internal storage where an app can organize its files
  - Your app does not require any permissions to read and write to files in these directories
  - Other apps cannot access files stored within internal storage, which makes an internal storage a good place for app data that other apps shouldn't access

- Keep in mind, however, that these directories tend to be small!

App-specific Data: Internal Storage (2/4)

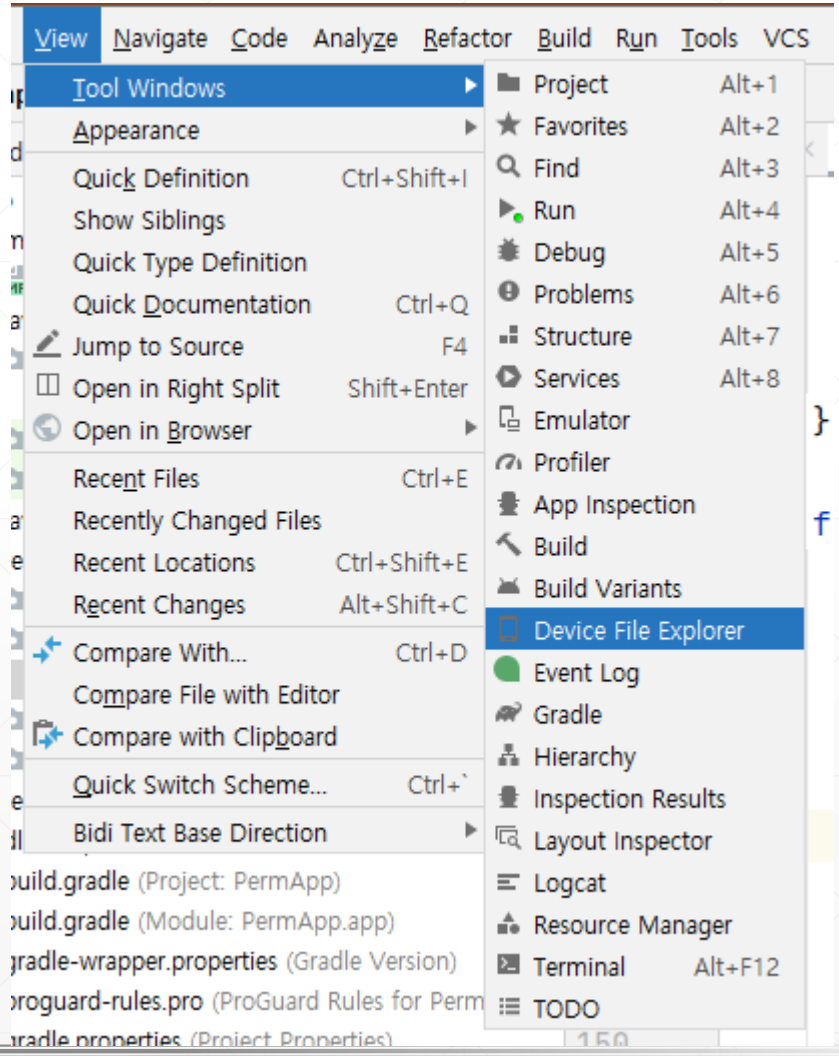
- Access persistent files
  - Your app's ordinary, persistent files reside in a directory that you can access using the *filesDir* property of a context object
  - You can use the File API to access and store files
  - Writing example)

```
fun saveFile(view: View){
    val filename = "myfile"
    val file = File(filesDir, filename)
    val writer = BufferedWriter(FileWriter(file))
    val sharedPref = getPreferences(Context.MODE_PRIVATE) ?: return

    Toast.makeText(this,"File saved!",Toast.LENGTH_SHORT).show()
    writer.append("my Code was ${sharedPref.getInt("ITMCode", 0)}")
    writer.append("my Grade was ${sharedPref.getString("ITMGrade", "A+")}")
    writer.close()
}
```

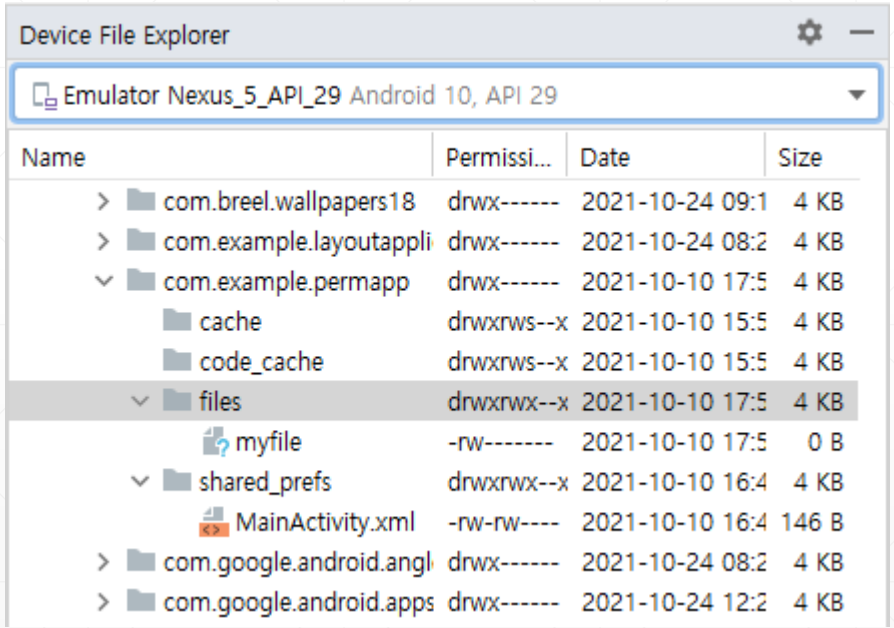


App-specific Data: Internal Storage (3/4)

- Device file explorer



/data/data/your_package/…



App-specific Data: Internal Storage (4/4)

- Access persistent files
  - Your app's ordinary, persistent files reside in a directory that you can access using the *filesDir* property of a context object
  - You can use the File API to access and store files
  - Reading example)

```
fun loadFile(view: View){
    val filename = "myfile"
    val file = File(filesDir, filename)
    val files: Array<String> = fileList()
    for (i in files)
        Log.d("ITM", i)
    val reader = BufferedReader(FileReader(file))
    reader.readLines().forEach{
        Log.d("ITM",it)
    }
}
```

App-specific Data: External Storage (1/6)

- If internal storage does not provide enough space to store app-specific files, consider using **external storage** instead!

- On Android 4.4 (API level 19) or higher, your app doesn't need to request any storage-related permissions to access app-specific directories within external storage

- Verify that storage is available first!
  - Because external storage resides on a physical volume that the user might be able to remove, verify that the volume is accessible before trying to access an external storage
  - Query the volume's state by calling Environment.getExternalStorageState()
    - MEDIA_MOUNTED, you can read and write app-specific files within external storage
    - MEDIA_MOUNTED_READ_ONLY, you can only read these files

App-specific Data: External Storage (2/6)

- Select a physical storage location
  - Sometimes, a device that allocates a partition of its internal memory as external storage also provides an SD card slot
  - This means that the device has multiple physical volumes (!) that could contain external storage, so you need to select which one to use for your app-specific storage

- To access the different locations, call ContextCompat.getExternalFilesDirs()
  - The first element in the returned array is considered the primary external storage volume
  - Use this volume unless it's full or unavailable

App-specific Data: External Storage (3/6)

- Example)

```
Log.d("ITM", "External Storage Mounted: " +
      "${Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED}")

val externalStorageVolumes: Array<out File> =    ContextCompat.getExternalFilesDirs(applicationContext, null)

for (ext in externalStorageVolumes) Log.d("ITM","ext device: $ext")

val primaryExternalStorage = externalStorageVolumes[0]
Log.d("ITM", "Primary ext storage: $primaryExternalStorage")
```

App-specific Data: External Storage (4/6)

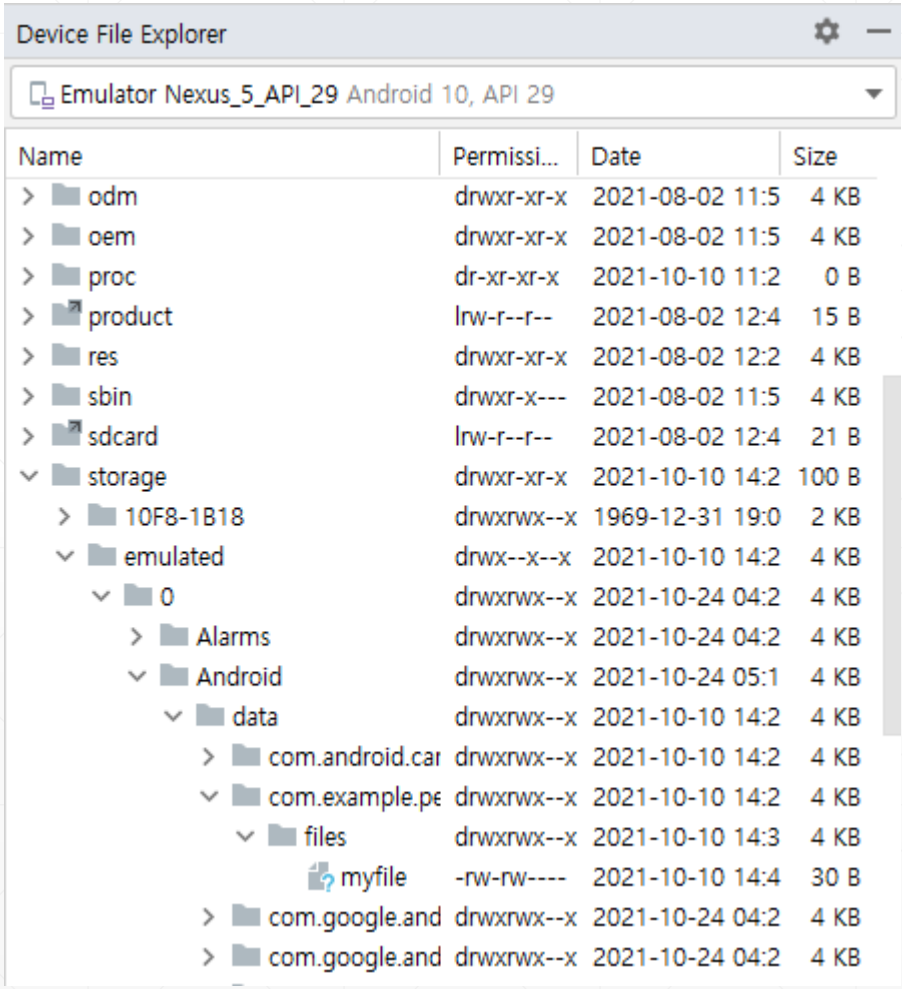
- Writing example)
  - To access app-specific files from external storage, call getExternalFilesDir(String type)
    - Returns the absolute path to the directory on the primary shared/external storage device
    - Type
      - May be null for the root of the files directory
      - or one of the following constants for a subdirectory: Environment.DIRECTORY_MUSIC, Environment.DIRECTORY_PODCASTS, Environment.DIRECTORY_RINGTONES, Environment.DIRECTORY_ALARMS, Environment.DIRECTORY_NOTIFICATIONS, Environment.DIRECTORY_PICTURES, or Environment.DIRECTORY_MOVIES

```
fun saveFile(view: View){
    val filename = "myfile"
//    val file = File(filesDir, filename)
    val file = File(getExternalFilesDir(null), filename)
    Log.d("ITM","$file")
    val writer = BufferedWriter(FileWriter(file))
    val sharedPref = getPreferences(Context.MODE_PRIVATE) ?: return

                ...
```

App-specific Data: External Storage (5/6)

- Reading example)

```
fun loadFile(view: View){
    val filename = "myfile"
    //val file = File(filesDir, filename)
    val file = File(getExternalFilesDir(null), filename)
    val files: Array<String> = fileList()

    …
```



App-specific Data: External Storage (6/6)

- Media contents
  - If your app works with media files that provide value to the user only within your app, it's best to store them in ***app-specific directories within external storage***

```
fun saveFile(view: View){
    val filename = "myfile"
//      val file = File(filesDir, filename)
    //val file = File(getExternalFilesDir(null), filename)

    val dir = getExternalFilesDir(Environment.DIRECTORY_PICTURES)
    val file = File(dir,filename)

    Log.d("ITM","$file")
    val writer = BufferedWriter(FileWriter(file))
    val sharedPref = getPreferences(Context.MODE_PRIVATE) ?: return

            ...
```

  - It's important that you use directory names provided by API constants like DIRECTORY_PICTURES
  - These directory names ensure that the files are treated properly by the system

Room: Introduction (1/2)

- Saving data to a database is ideal for repeating or structured data, such as contact information!

- SQLite
  - A small, fast, self-contained, high-reliability, full-featured, SQL database engine
  - Mobile, embedded database!
  - Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:
    - There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone
    - You need to use lots of boilerplate code to convert between SQL queries and data objects

- Room persistence library provides an abstraction layer over SQLite
  - Allows fluent database access while harnessing the full power of SQLite!

Room: Introduction (2/2)

Save data using SQLite

Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.

Caution: Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

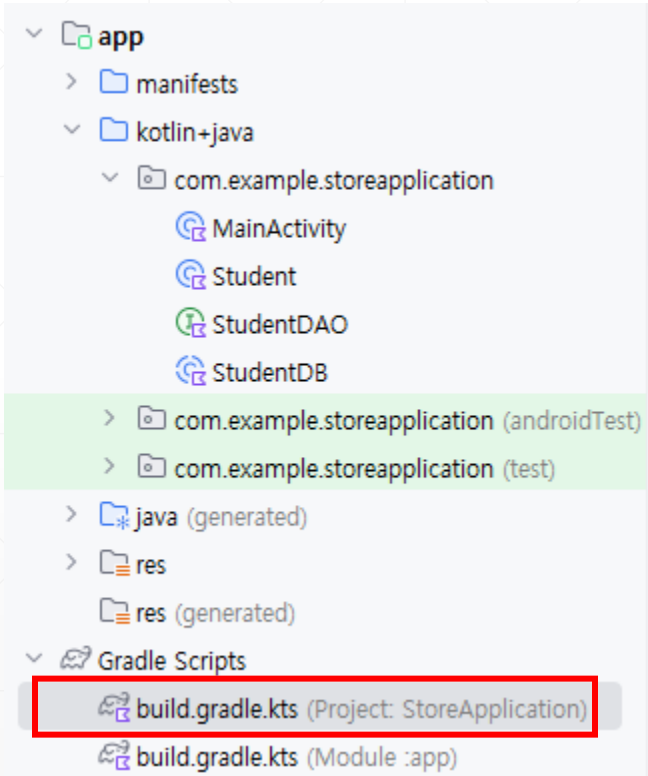
- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the Room Persistence Library as an abstraction layer for accessing information in your app's SQLite databases.

- This class also assumes that you are familiar with SQL databases in general!

Room: Setup

- KSP setup
  - Declare the KSP plugin in your top level build.gradle.kts file




```
1  // Top-level build file where you can add configuration options common t
2  plugins {
3      alias(libs.plugins.android.application) apply false
4      alias(libs.plugins.kotlin.android) apply false
5      id("com.google.devtools.ksp") version "2.0.21-1.0.28" apply false
6  }
```

★ **Note:** The first part of the KSP version must match the version of Kotlin being used in your build. For example, if you're using Kotlin `2.0.21`, the KSP version must be one of the `2.0.21-x.y.z` releases.

- Migrate from kapt to KSP | Android Studio | Android Developers

Room: Setup

- KSP setup
  - Then, enable KSP in your module-level build.gradle.kts file:


```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("com.google.devtools.ksp")
}

android {
    namespace = "com.example.storeapplication"
    compileSdk = 36

    defaultConfig {
        applicationId = "com.example.storeapplication"
        minSdk = 35
        targetSdk = 36
        versionCode = 1
```

- Migrate from kapt to KSP | Android Studio | Android Developers

Room: Setup

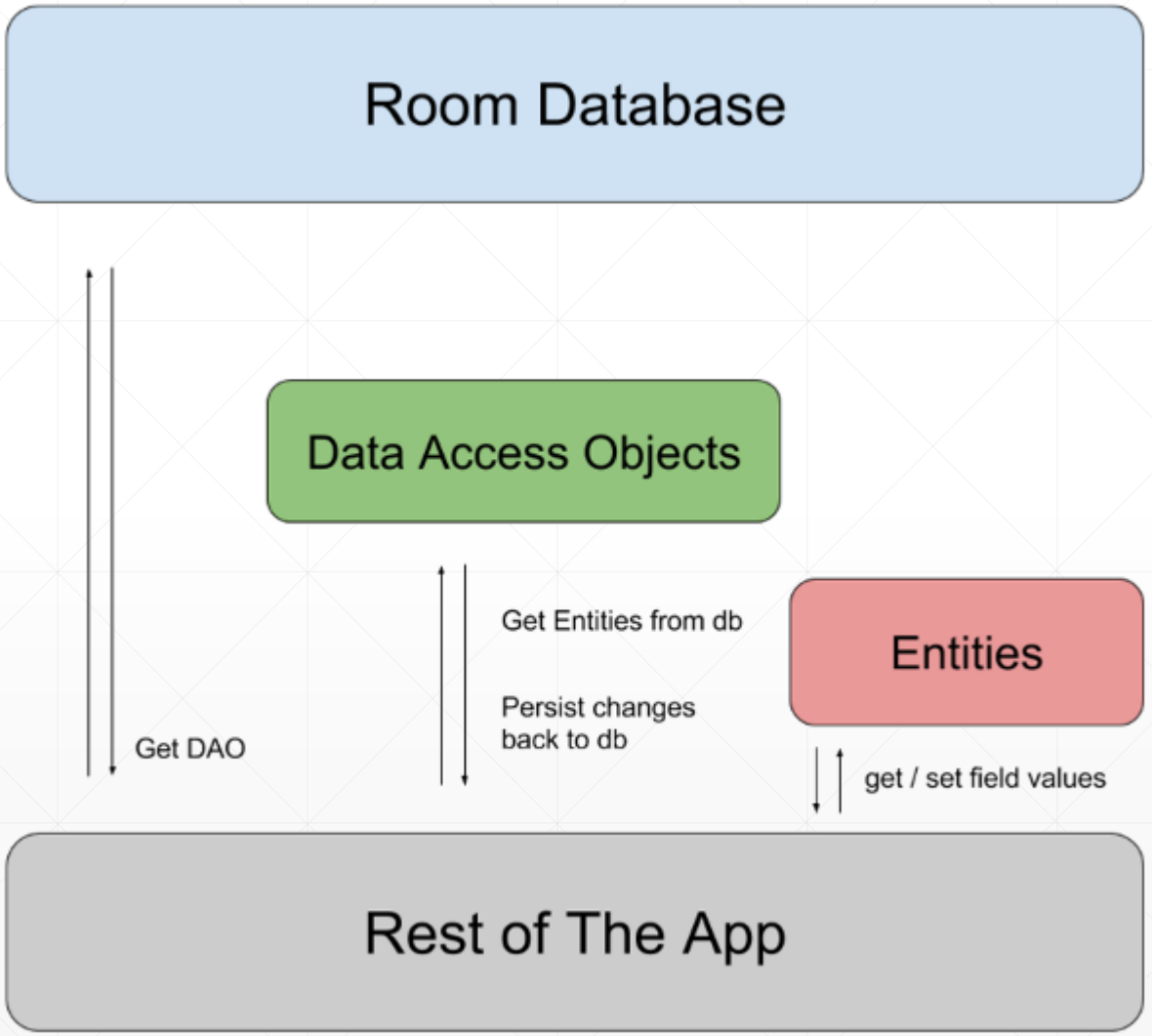
- Dependencies
  - Add the following dependencies to your module-level build.gradle.kts file

```
...

implementation("androidx.room:room-runtime:2.8.3")

ksp("androidx.room:room-compiler:2.8.3")

}
```

User the latest version here!

Room: Primary Components

- Database class
  - Holds the database and serves as the main access point for the underlying connection to your app's persisted data
- Data entities
  - Represent tables in your app's database
- Data access objects (DAOs)
  - Provide methods that your app can use to query, update, insert, and delete data in the database



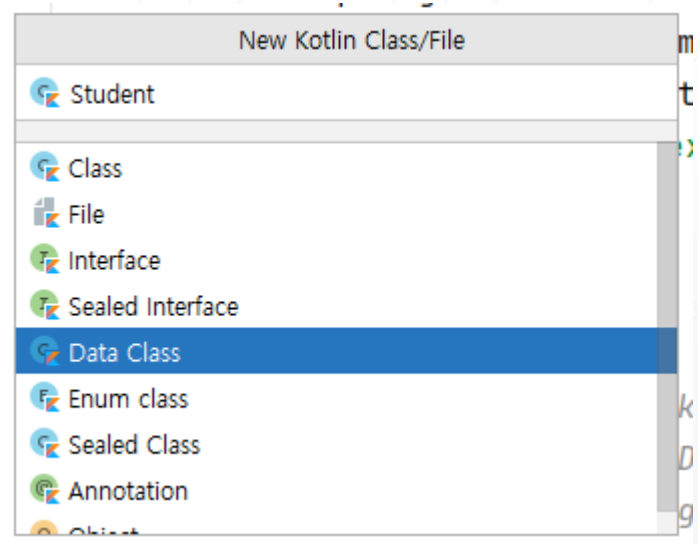
Room: Data Entities (1/3)

- You need to define entities to represent the objects that you want to store
  - Each entity corresponds to a table in the associated Room database
  - Each instance of an entity represents a row of data in the corresponding table

- You can use Room entities to define your database schema without writing any SQL code!

Room: Data Entities (2/3)

- Define each Room entity as a class that is annotated with @Entity
- A Room entity includes fields for each column in the corresponding table in the database, including one or more columns that comprise the primary key!
  - A composite primary key can be defined by listing the columns in the *primaryKeys* property of @Entity
  - "augoGenerate" property can be used to assign automatically generated IDs




```kotlin
@Entity
data class Student(
    @PrimaryKey(autoGenerate = true) val id:Int,

    val firstName: String,
    val lastName: String
)
```

```kotlin
@Entity(primaryKeys = ["firstName", "secondName"])
data class Student(

    val firstName: String,
    val lastName: String
)
```

Room: Data Entities (3/3)

- By default, Room uses the class name as the database table name
  - If you want the table to have a different name, set the tableName property of the @Entity annotation
- Similarly, Room uses the field names as column names in the database by default
  - If you want a column to have a different name, add the @ColumnInfo annotation to the field and set the name property

```
@Entity(tableName="students")
data class Student(
   @PrimaryKey(autoGenerate = true) val id:Int,

   @ColumnInfo(name="first_name") val firstName: String,
   @ColumnInfo(name="last_name") val lastName: String,
   val score: Int,
   val grade: String
)
```

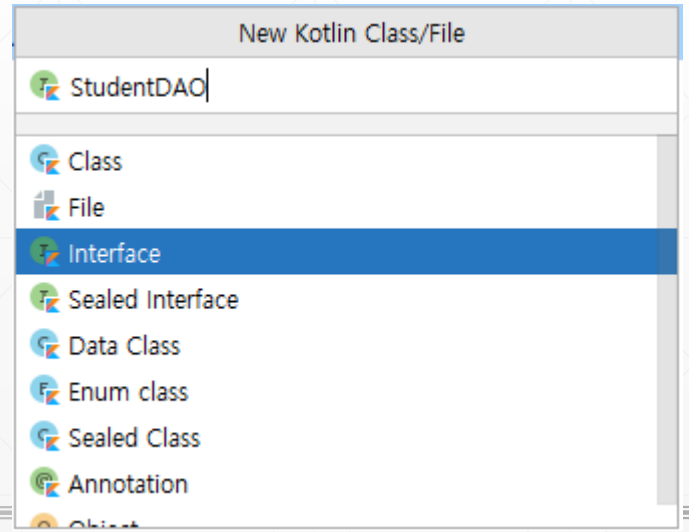
Room: DAO (1/5)

Data Access Object (DAO)

- Each DAO includes methods that offer abstract access to your app's database
  - You can specify SQL queries and associate them with method calls
  - The compiler checks the SQL and generates queries from convenience annotations for common queries, such as @Insert

DAO definition

- You can define each DAO as either an interface or an abstract class
- For basic use cases, you should usually use an interface



Room: DAO (2/5)

- DAOs must be annotated with @Dao
- Two types of DAO methods that define database interactions:
  - Convenience methods that let you insert, update, and delete rows in your database **without writing any SQL code!**
  - Query methods that let you **write your own SQL** query to interact with the database

```
@Dao
interface StudentDAO {
    @Insert
    fun insert(student: Student)

    @Delete
    fun delete(student: Student)

    @Query("SELECT * FROM students")
    fun getAll(): List<Student>
}
```

Room: DAO (3/5)

- Convenient methods
  - @Insert: allows you to define methods that insert their parameters into the appropriate table in the database
    - Each parameter for an @Insert method must be either an instance of a Room data entity class annotated with @Entity or a collection of data entity class instances
    - onConflict: strategy on what to do if a conflict happens!
      - OnConflictStrategy.ABORT (default) to roll back the transaction on conflict
      - OnConflictStrategy.REPLACE to replace the existing rows with the new rows
      - OnConflictStrategy.IGNORE to keep the existing rows

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun insert(student: Student)

@Insert
fun insertTwoStudents(student1: Student, student2: Student)

@Insert
fun insertStudents(studentList:List<Student>)
```

Room: DAO (4/5)

- Convenient methods
  - @Update/Delete: allows you to define methods that update/delete specific rows in a database table
  - Room uses the primary key to match passed entity instances to rows in the database
    - If there is no row with the same primary key, Room makes no changes!

```
@Update
fun updateStudent(student: Student)

@Delete
fun delete(student: Student)
```

Room: DAO (5/5)

- Query methods
  - @Query: allows you to write SQL statements and expose them as DAO methods
  - Use these query methods to query data from your app's database, or when you need to perform more complex inserts, updates, and deletes
  - Room validates SQL queries at compile time
    - This means that if there's a problem with a query, a compilation error occurs instead of a runtime failure!

```
@Query("SELECT * FROM students")
fun getAll(): List<Student>

@Query("SELECT firstName from students where score > :minScore")
fun searchStudent(minScore: Int): List<String>
```

```
@Entity(tableName="students")
data class Student(
   @PrimaryKey(autoGenerate = true) val id:Int,

   val firstName: String,
   val lastName: String,
   val score: Int,
   val grade: String
)
```

Room: Database (1/3)

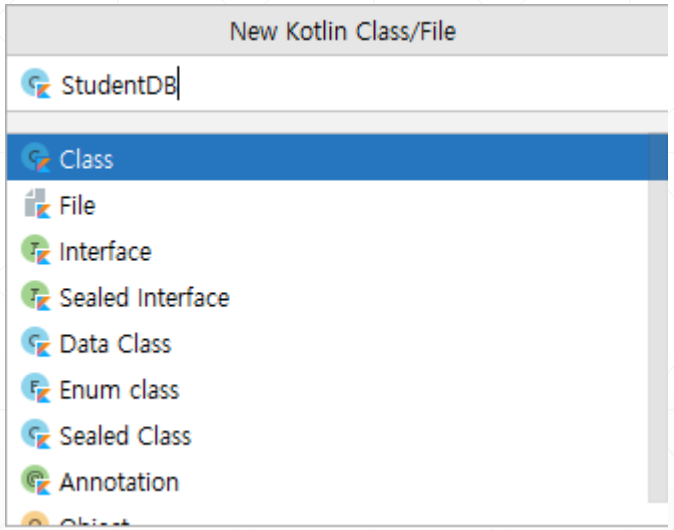
- Class to hold the database and define the database configuration
  - Serves as the app's main access point to the persisted data

- The database class must satisfy the following conditions:
  - The class must be annotated with a @Database annotation that includes an entities array that lists all of the data entities associated with the database.
  - The class must be an abstract class that extends RoomDatabase
  - For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class

Room: Database (2/3)

Database definition

- Room database class must be abstract and extend RoomDatabase




```
@Database(entities=[Student::class], version=1)
abstract class StudentDB : RoomDatabase() {

    abstract fun studentDAO(): StudentDAO

    ...
}
```

- entities: entities that belong in the database (each entity corresponds to a table that will be created in the database)
- version: database version

Room: Database (3/3)

- Database definition
  - Singleton pattern to return RoomDatabase instance
    - It prevents having multiple instances of the database opened at the same time
  - Creates the database the first time it's accessed, using Room's database builder to create a RoomDatabase object in the application context from the StudentDB class and names it “student_database"

```
@Database(entities=[Student::class], version=1)
abstract class StudentDB : RoomDatabase() {

    abstract fun studentDAO(): StudentDAO

    companion object {
        // Singleton prevents multiple instances of database opening at the same time.
        @Volatile
        private var INSTANCE: StudentDB? = null

        fun getInstance(context: Context): StudentDB {
            // if the INSTANCE is not null, then return it,
            // if it is, then create the database
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    StudentDB::class.java,
                    "student_database"
                ).build()
                INSTANCE = instance
                // return instance
                instance
            }
        }
    }
}
```

Room: Initialization and Usage (1/4)

- Get Room database instance

```
class MainActivity : AppCompatActivity() {
    val binding by lazy { ActivityMainBinding.inflate(layoutInflater) }
    val stuDB: StudentDB by lazy { StudentDB.getInstance(this) }
...
```

- Call methods defined in DAO for the database
  - To prevent queries from blocking the UI, Room does not allow database access on the main thread
    - This restriction means that you must make your DAO queries asynchronous!
  - Solution
    - Allow main-thread call (only for test purpose)
    - Perform as asynchronous tasks: coroutines/thread/…

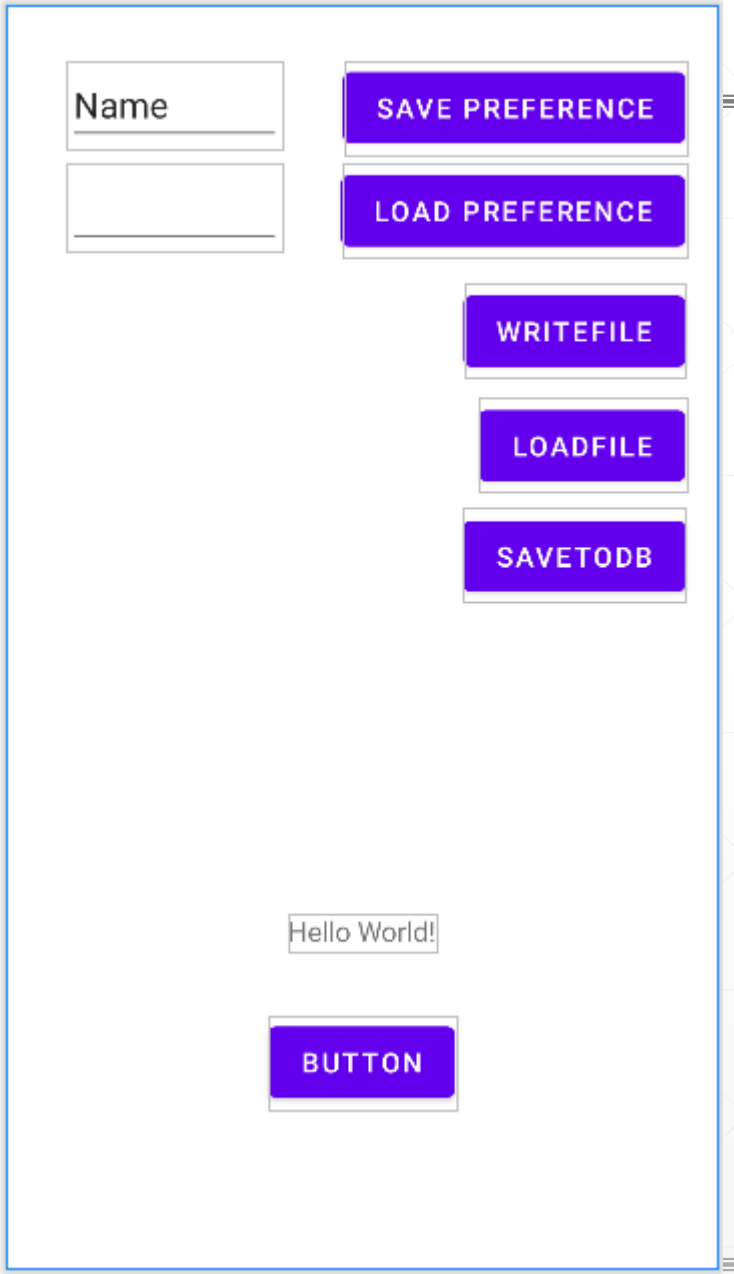
Room: Initialization and Usage (2/4)

- Solution
  - Allow main-thread call (only for test purpose)

```
return INSTANCE ?: synchronized(this) {
    val instance = Room.databaseBuilder(
        context.applicationContext,
        StudentDB::class.java,
        "student_database"
    ).allowMainThreadQueries()
    .build()
    INSTANCE = instance
    // return instance
    instance
}
```

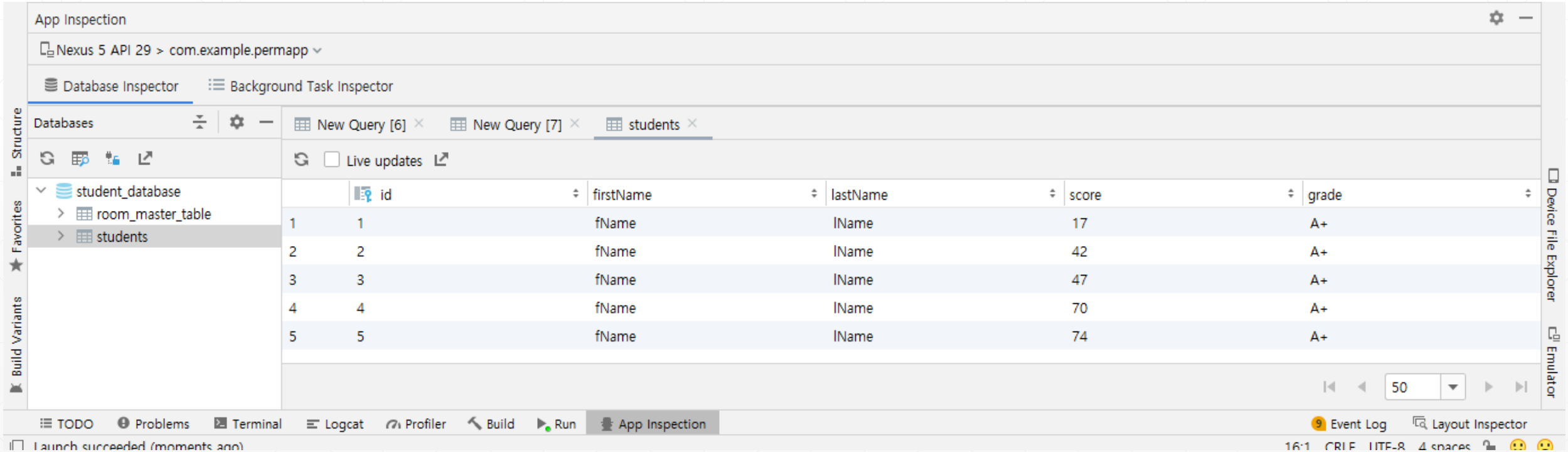
  - Insert feature

```
fun addStudents(view: View) {

    for (i in 0..4) {
        val stu = Student(0, "fName$i", "lName", (0..100).random(), "A+")
        stuDB.studentDAO().insert(stu)
    }

}
```



Room: Initialization and Usage (3/4)

- App inspection
  - You can check the current status of your database / tables / records
  - View → Tool Windows → App Inspection



Room: Initialization and Usage (4/4)

- Perform as asynchronous tasks: coroutines/thread/…
  - Simple coroutine example

```
fun addStudents(view: View) {
    GlobalScope.launch(Dispatchers.IO) {
        for (i in 0..4) {
            val stu = Student(0, "fName$i", "lName", (0..100).random(), "A+")
            stuDB.studentDAO().insert(stu)
        }
    }
}
```

- Example)

```
fun findStudent(view: View){
    GlobalScope.launch(Dispatchers.IO) {
        val stuList :List<String> = stuDB.studentDAO().searchStudent(90)
        Log.d("ITM",stuList.toString())
    }
}
```