

# **BCS 371**

# **Mobile Application Development I**

Arthur Hoskey, Ph.D.  
Farmingdale State College  
Computer Systems Department

- Architecture - Room Persistence Library and SQLite.
- Using Room DB with Kotlin Coroutines and Flows.

## Today's Lecture

## **SQLite**

- SQLite is a relational database
  - Open-source
  - Standards-compliant
  - Lightweight (little configuration required)
  - Single-tier (client, server, db all on the same machine)
- SQLite is implemented as a compact C library rather than a separate ongoing process
  - Note: Oracle and SQL Server both run in a separate process.
- Since SQLite is a library, it is integrated as part of the application that is using it. This type setup has the following effects:
  - Reduces external dependencies
  - Minimizes latency
  - Simplifies transaction locking and synchronization.

# SQLite

## **Room Persistence Library**

- Room Persistence Library - One of the new Android Architecture Components. Link:  
<https://developer.android.com/jetpack/docs/guide>
- Makes it easier to create/access an SQLite database.
- Both the Room Persistence Library and an SQLite database are used to store data locally.
- The Room library is better because it hides some messy code that is generally needed to setup an SQLite database.
- Uses of local storage:
  - Save user settings.
  - Save copies of remote data (data stored in the cloud).
- If data is coming from a web service (remote data) we might be able to store it locally in a database. We can check the local database first for the data BEFORE running a web service call. This is good because web service calls are slow.

## **Room Persistence Library**

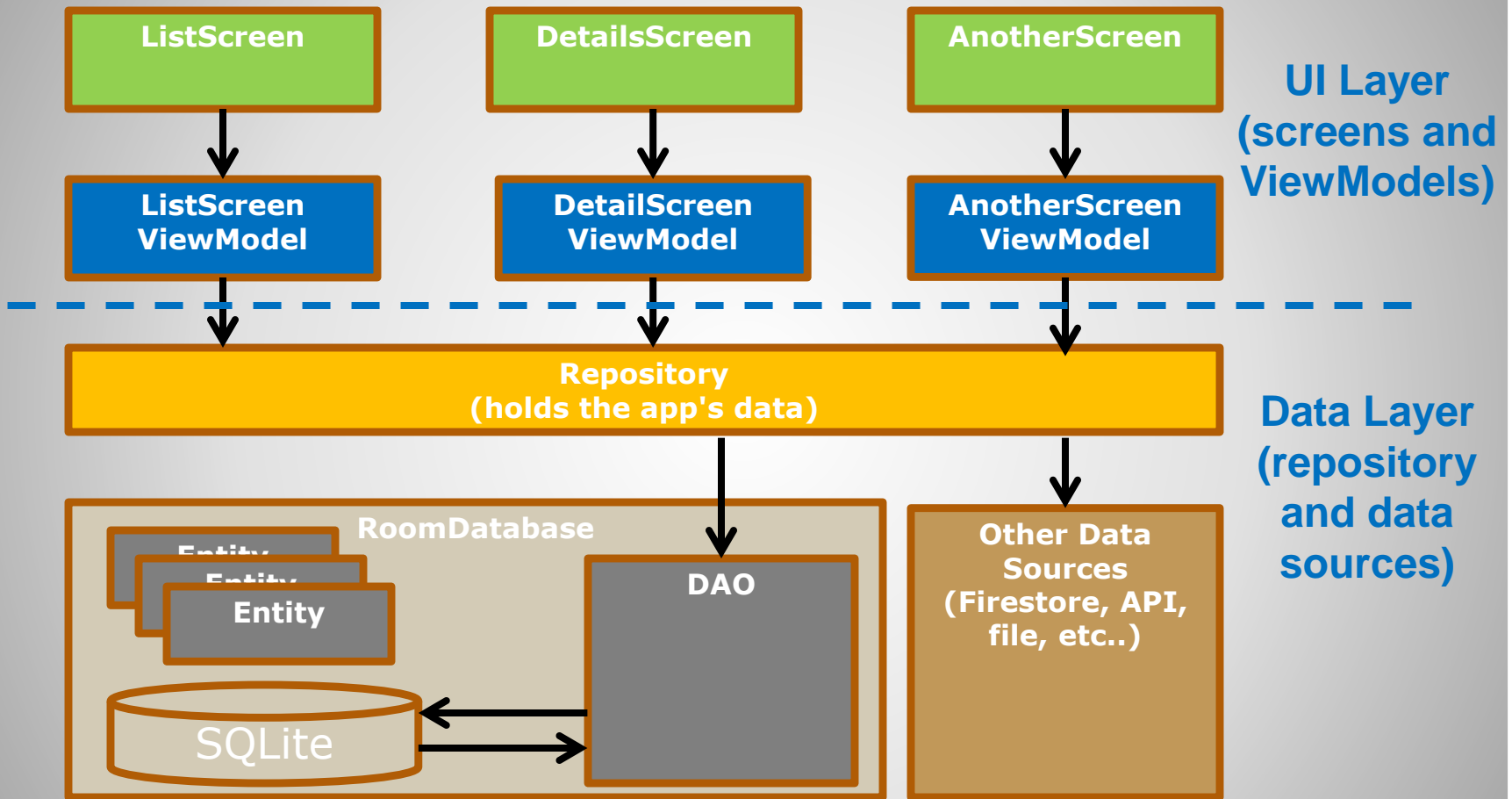
- **Object-Relational Mapping (ORM)** tools allow you to work with a relational database using objects inside of an object-oriented program.
- Room Persistence Library is an object relational mapping tool.



- There are ORM tools for other languages.
- For example: Entity Framework (.NET), Hibernate (Java), NHibernate(.NET), Sequelize (Node.js), SQLAlchemy (Python), Doctrine (PHP).


## Object Relational Mapping (ORM)

## High-Level Architecture and Room



## High-Level Architecture and Room

- Use following dependencies in build.gradle (lower-level file, app).

```
plugins {  
    //other plugins here...  
    id("kotlin-kapt")   
}
```

**IMPORTANT**  
Add the "Kotlin-kapt" plugin. It should be added to the end of the plugins block at the top of the file.

```
dependencies {  
    // Other dependencies here
```

```
var roomVersion = "2.6.1"
```

```
implementation("androidx.room:room-runtime:$roomVersion")  
implementation("androidx.room:room-ktx:$roomVersion")  
kapt( "androidx.room:room-compiler:$roomVersion")  
}
```

Versions change quickly in Android. Check the following link to get the latest dependency version:

<https://developer.android.com/jetpack/androidx/releases/room>

# Room – Gradle Dependencies

- Make sure to include the Room import statement as necessary in your Kotlin code.

```
import androidx.room.*
```

# Room – import Statement

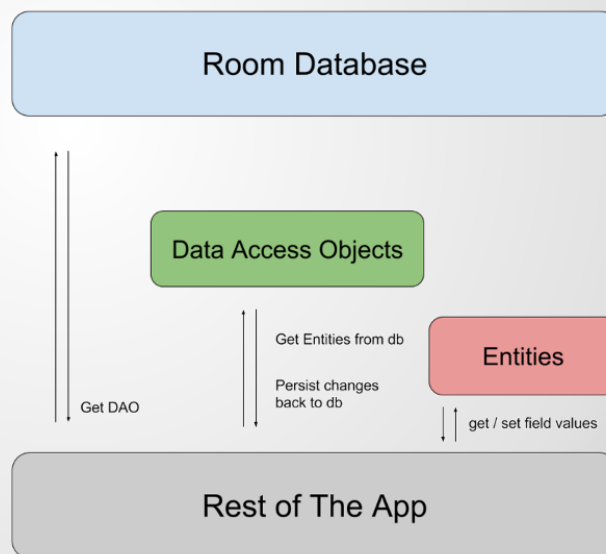


# Three Major Components in a Room Database

- **RoomDatabase**. Main access point to the database.
- **Entity Class**. Represents a table in the database.
- **Data Access Object (DAO)**. Contains methods used for accessing the database.
- Taken from: <https://developer.android.com/training/data-storage/room/>

Only need to define the  
above three  
components.

You do NOT need to  
make any changes to  
the application manifest



## Room Database Components

- A Room entity class maps to a table in a relational DB (a Room entity class is just a normal Kotlin class).
- The member variables in a Room entity class correspond to the columns in a relational database table.
- Here are the mappings:
  - uid (User class) corresponds to the Id column (User table)
  - firstName (User class) corresponds to the FirstName column (User table)
  - lastName (User class) corresponds to the LastName column (User table)

### **User Class (normal Kotlin class)**

```
uid: Int  
firstName: String  
lastName: String
```

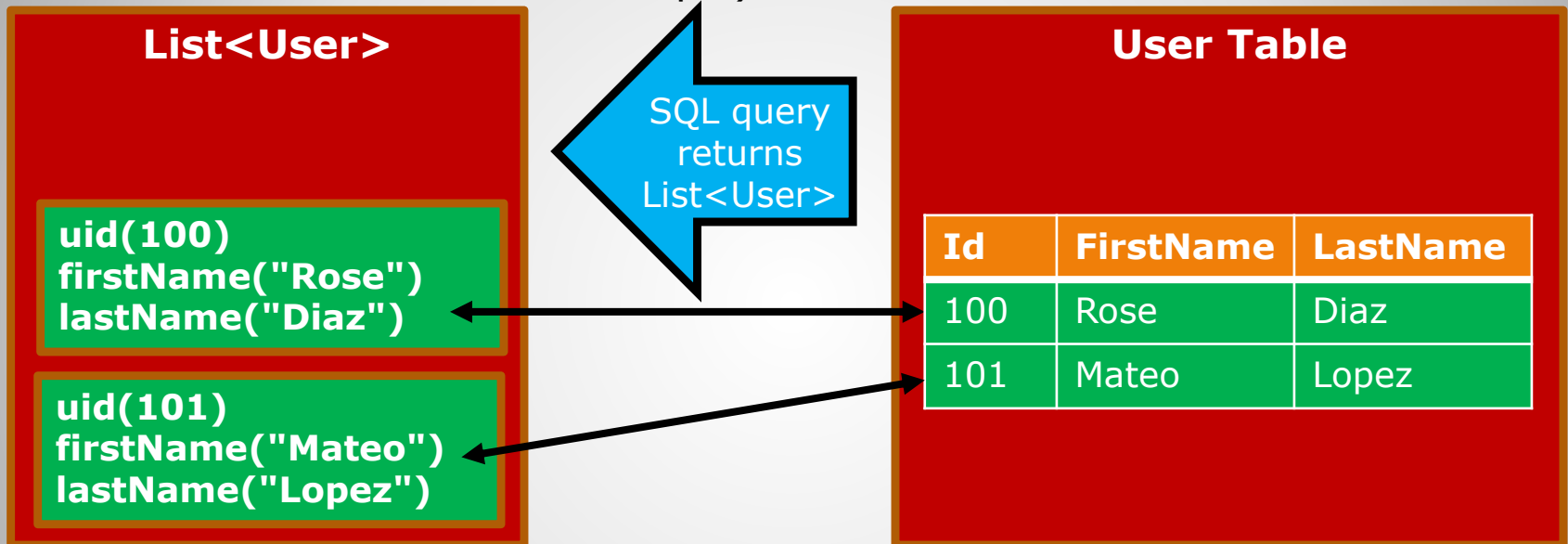
### **User Table in Relational DB**

<b>Id</b>	<b>FirstName</b>	<b>LastName</b>

**Note:** Class member variable names do not need to match table column names.

# **Entity Class Maps to Relational Table**

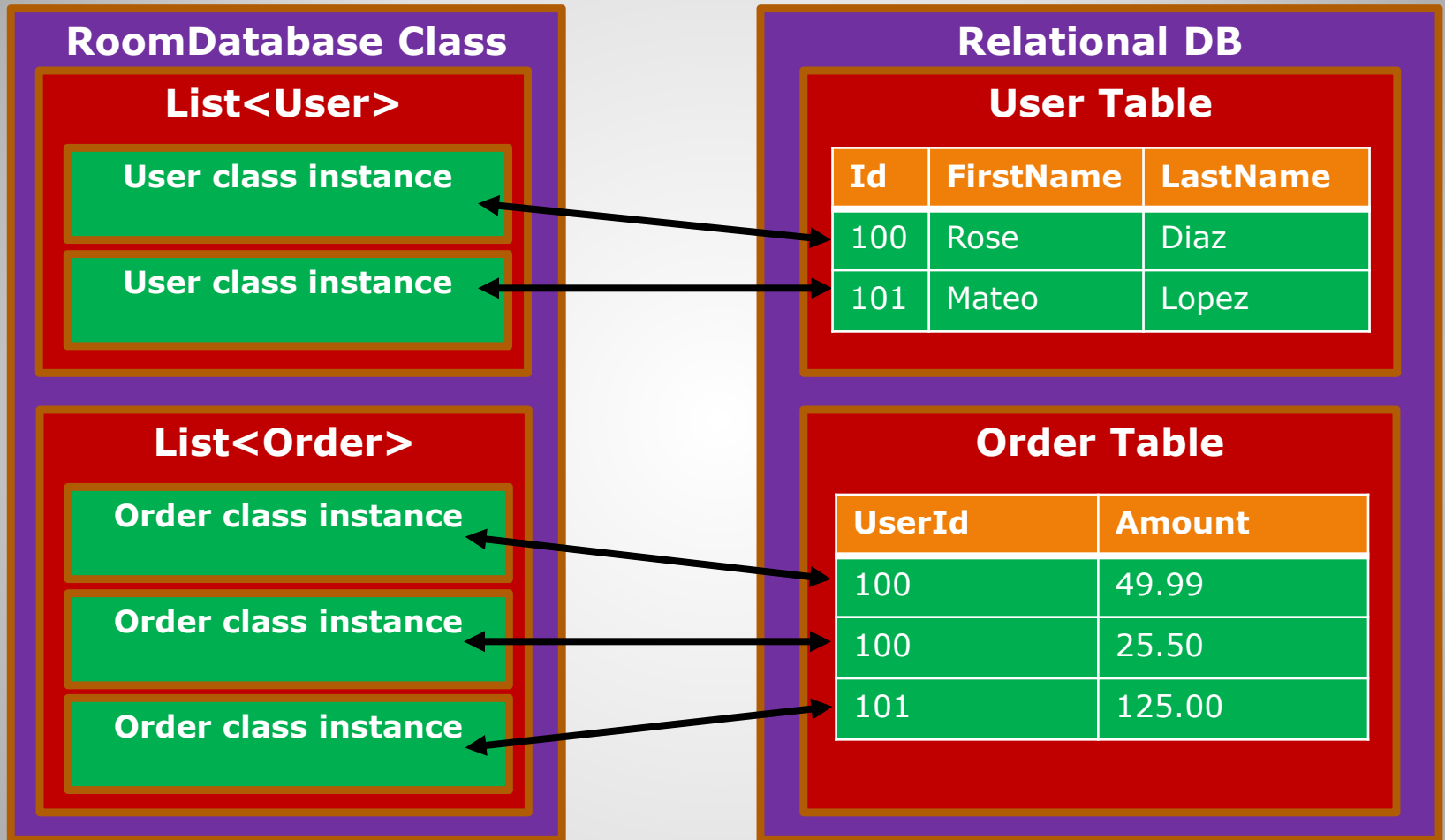
- One instance of a Room entity class corresponds to a row in a relational DB table.
- A one table SQL query returns a List of the entity class type (returns a list of the User class in this example).



- Check link for more complicated querying scenarios:  
<https://developer.android.com/training/data-storage/room/accessing-data>

## Entity Class Maps to Relational Table

- The example below has two entity classes.



## Room Mappings

## **Setup RoomDatabase**

- Need to do the following to setup the RoomDatabase:
  1. Create entity classes for each table (entity classes mirror the DB tables, one entity class per table).
  2. Define a Data Access Object interface.
  3. Define an abstract database class.
  4. Create the database.

## **RoomDatabase Setup Overview**

## 1. Create Entity class

This class will create a table named User

- Describes one table in the database.
- Create a normal Kotlin class decorated with annotations.
- Use annotations such as @Entity, @PrimaryKey, and @ColumnInfo.
- Taken From: <https://developer.android.com/training/data-storage/room/defining-data>

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

Annotations and their meanings:

- @Entity**: Indicates that class can be used as an entity. Use `@Entity(tableName="myTableName")` to link to a specific table name.
- @PrimaryKey**: Column is primary key (use `@PrimaryKey(autoGenerate = true)` to automatically generate ids)
- @ColumnInfo(name = "first\_name")**: Column in db should be named "first\_name" (not firstName).
  - `String?`: ? Allows null value (cannot use ? if the column is a primary key)
- @ColumnInfo(name = "last\_name")**: Column in db should be named "last\_name" (not lastName).

**Note:** If you leave out a ColumnInfo annotation specifying the name it will use the variable name for the column name (see uid above).

# 1. Create Entity Class

## 2. Data Access Object (DAO)

- An interface.
- Contains methods used to access the database.
- The implementations for the **methods of this interface are created automatically** for you behind the scenes.
- A **big advantage of writing SQL statements as part of DAO** annotations is that the compiler will make sure that the SQL is syntactically correct. If the SQL is not correct a compile error will be issued. Program will not even run. Not the case when using SQLite directly.
- Here is a sample Data Access Object...

## 2. Data Access Object

- DAO interface that uses flows and suspending functions.

`@Dao` ← Annotation indicating that this interface is a data access object

The compiler WILL check the syntax of SQL statements inside of the annotations!!!

```
interface UserDao {
```

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

```
    fun selectAll() : Flow<List<User>> ← Uses a Flow  
                                         (a Flow uses coroutines  
                                         internally so selectAll does not  
                                         need the suspend keyword)
```

```
    @Insert
```

```
    suspend fun insert(user: User)
```

```
    @Query("DELETE FROM user")
```

```
    suspend fun deleteAll()
```

Both functions are suspend so they should be run on a coroutine

```
}
```

## 2. Data Access Object



### 3. Define Abstract Database Class

- The database class is abstract (create a Kotlin class).
- The room library will create a concrete implementation of this abstract database class.
- Entities are passed into the Database annotation.
- The DAO can be retrieved using the abstract method on the Database class.

The Database annotation must be given the entities (tables)

User entity class

```
@Database(entities = [User::class], version = 1)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao?
}
```

Inherit from RoomDatabase

userDao is an abstract method that returns the DAO instance

### 3. Define Abstract Database Class

## 4. Create the Database

- Use the following code to get a database instance:

```
// Get the database
```

```
val db = Room.databaseBuilder(  
    applicationContext,  
    UserDatabase::class.java, "user"  
).build()
```

You can use whatever name you want for the database. You do not have to use "user".

**This code will NOT work if run on the main thread (UI thread)**

**See next slide for details...**

## **4. Create the DB - Getting the DB Instance**

**MainGUI cannot do anything until dbOperation finishes**

Run on Main Thread  
fun MainGUI()  
{

dbOperation()

**Wait for dbOperation to finish**

}

1. Main calls dbOperation synchronously

2. dbOperation runs (MainGUI is blocked)

Run on Main Thread  
fun dbOperation()  
{

// Code for some DB  
// operation (select,  
// insert, etc...)  
}

3. dbOperation returns and main continues execution

MainGUI function cannot update GUI while waiting for dbOperation (user cannot interact)



**The user will NOT be able to interact with the GUI while the dbOperation code is running! GUI HANGS, VERY BAD USER EXPERIENCE!**

# DB Operation Running on Main Thread (BAD)

- By default, if you try to run a database query/statement on the main (UI) thread the program will compile and execute but you will get the following runtime error:

**Caused by: java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long period of time.**

- This error is thrown because there is the potential to lock the UI screen (user cannot interact/UI hangs) if the operation is long.
- **Database access MUST be done on a background thread (not the UI thread).**
- **You can run code on a background thread** but unfortunately, we have not covered it yet.
- Here is a **dangerous workaround**:

```
val db = Room.databaseBuilder(  
    applicationContext,  
    UserDatabase::class.java, "user"  
).allowMainThreadQueries().build()
```

**This is a  
bad fix!!**

## 4. Create the DB Room and Threading

- Create the database inside a Kotlin coroutine on another thread.
- Room.databaseBuilder needs the application context as a parameter.
- Here are examples (run this code in init block for a view model):

```
runBlocking(Dispatchers.IO) {  
    val db = Room.databaseBuilder(  
        applicationContext,  
        UserDatabase::class.java, "user"  
    ).build()  
}
```

← Use runBlocking for DB creation if other code needs the database to be created before it can do anything.

← Run database creation code inside a coroutine. The application context must be passed as a parameter. If in a view model, pass the application in through a primary constructor and use that as the context.

```
val coroutineScope = CoroutineScope(Dispatchers.IO)  
coroutineScope.launch {  
    val db = Room.databaseBuilder(  
        applicationContext,  
        UserDatabase::class.java, "user"  
    ).build()  
}
```

← Could also create a coroutine scope. For a view model this can be run in the init block.

Note: If the db instance is going to be a member of a class remove "val" and declare db as a member variable elsewhere (like in a view model).

## 4. Create DB – Run DB Creation Code Inside a Kotlin Coroutine

**MainGUI continues immediately (does not wait for dbOperation to return)**

Run on Main Thread

```
fun MainGUI() {  
    Create coroutine,  
    call dbOperation()  
    on it
```

**Keep going!  
Do not wait for  
dbOperation to  
finish**

```
}
```

**1. Main calls  
dbOperation on  
a coroutine**



**2. dbOperation runs  
(MainGUI is NOT blocked)**

Run in a Coroutine

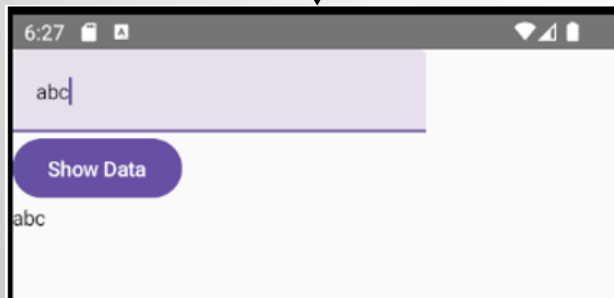
```
void dbOperation()  
{
```

```
{
```

```
    // Code for some DB  
    // operation (select,  
    // insert, etc...)
```

```
}
```

**MainGUI function does not  
wait and keeps updating GUI  
(user can interact)**



**The user can interact with the  
GUI while dbOperation runs  
(GUI not locked). Creates a  
good user experience.**

# DB Operation Running on a Coroutine (GOOD)

- Now on to executing queries and inserting data...

## Querying and Inserting Data

## Inserting Data Into the Database

- Create a normal instance of the entity class and populate it like you normally would (the User class in this example).
- Assumes the insert is being called in a view model.
- Use a different coroutine scope if not calling in a view model.

```
@Dao
interface UserDao {
    @Insert
    suspend fun insert(user: User)
}
```

There must be an insert method that takes one User as a parameter on the UserDao interface (code below will not work if it is missing)

```
viewModelScope.launch {
    val u = User(100, "Rose", "Diaz")
    db.userDao().insert(u)
}
```

Must call insert in a coroutine scope (change scope if not in view model)

Create a normal instance of User (the entity class)

Call insert on the userDao passing in the User instance

## Inserting Data Into the Database



## Getting Data From the Database

- Call methods on the data access object (DAO).
- This code assumes you are running queries from a view model.
- The code below assumes that you added a selectAll method prototype to the DAO.

```
var userList = mutableStateOf<List<User>>(emptyList())  
    private set
```

← **Member variable  
of view model**

```
init {  
    // other code here...  
    viewModelScope.launch {  
        db.userDao()!!.selectAll()!!.flowOn(Dispatchers.IO).collect {  
            currentUserList ->  
            userList.value = currentUserList  
        }  
    }  
}
```

← **Must call selectAll in a coroutine scope  
(change scope if not in view model)**

← **Put the current List<User> that comes  
through the flow into the userList state  
member. This will trigger the screen  
composable to do a recomposition  
(value changing in state variable)**

## Getting Data From the Database

- ViewModel that uses flows and suspending functions.

```
@Composable
fun MainScreen(modifier: Modifier) {

    val context = LocalContext.current

    val viewModel = viewModel { MainScreenViewModel(context.applicationContext as Application) }

    val userList = viewModel.userList.value

    // Code to display the users goes here

}
```

Get the context

Assumes MainScreenViewModel has been defined

Pass in the default Application instance (get it from the context)

Get the user list from the view model

## MainScreen that Uses Flow

- Now on to setting up a repository to access a Room DB...

## Setup Room DB in a Repository Class

- Repository that uses flows and suspending functions.

```
class UserRepository(  
    var userDatabase: UserDatabase  
) {  
    var userDao: UserDao  
  
    init {  
        userDao = userDatabase.userDao()!!  
    }  
  
    suspend fun getUsers() = userDao.selectAll()  
  
    suspend fun addUser(user: User) {  
        userDao.insert(user)  
    }  
}
```

Need to make both  
functions suspend  
because they are both  
calling suspend  
functions on the DAO

## Repository with Flow and Coroutines

- ViewModel that uses flows and suspending functions.

```
class MainScreenViewModel(  
    var userRepository: UserRepository  
) : ViewModel()  
{  
    var userListStateFlow = MutableStateFlow<List<User>>(emptyList())  
    private set  
  
    init {  
        viewModelScope.launch(Dispatchers.IO) {  
            userRepository.getUsers().flowOn(Dispatchers.IO).collect  
            { currentUserList ->  
                userListStateFlow.value = currentUserList  
            }  
        }  
    }  
}
```

Gets data immediately when  
the view model starts

Calls getUsers in a coroutine scope  
(coroutine scope can call suspend functions)

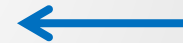
Put the current List<User> that comes  
through the flow into userListStateFlow  
member. This will trigger the screen  
composable to do a recomposition  
(value changing in state variable)

## ViewModel with Flow and Coroutines

- Custom Application class to create the Repository and database instances.

```
class MyApp : Application() {  
    companion object {  
        lateinit var userRepository: UserRepository  
    }  
  
    override fun onCreate() {  
        super.onCreate()  
        runBlocking(Dispatchers.IO) {  
            val userDatabase = Room.databaseBuilder(  
                applicationContext,  
                UserDatabase::class.java, "user"  
            ).build()  
        }  
        userRepository = UserRepository(userDatabase)  
    }  
}
```

Call DB builder code  
in onCreate



## Custom Application Class

- ViewModel that uses flows and suspending functions.

@Composable

```
fun MainScreen(modifier: Modifier) {
```

```
    val viewModel = viewModel { MainScreenViewModel(MyApp.userRepository) }
```

Get the  
ViewModel



collectAsState converts the  
StateFlow into a State object



```
    val userList = viewModel.userListStateFlow.collectAsState().value
```



Get the user list from  
the view model

```
    // Code to display the users goes here
```

```
}
```

## MainScreen that Uses Flow

- Now on to executing queries without using a flow...

## Data Access Object Query without Flows



- Queries can return normal lists (not flows).
- Important! The code below is not the suggested approach moving forward (flows should be used).
- This code will NOT automatically notify the app when data changes in the database.

```
@Dao
interface UserDao {

    @Query("SELECT * FROM user")
    fun getAll(): List<User>

}
```

## Data Access Object Query without Flows

- End of Slides

**End of Slides**