# MAD code snippets

- By Falcon

Note – Codes are highlighted in this color

Disclaimer: This document provides code snippets for reference purposes only. Please exercise caution and use them at your own risk, as they may contain potential errors.

1. Get user inputs (Consider getting a Float value from the user): -

```
var num1: Float? = findViewById<EditText>(R.id.edtNumber1).text.toString().toFloat()
```

2. Bind Buttons to values: -

```
val btnAdd: Button = findViewById(R.id.btnAdd)
```

3. Using explicit Intents: -
   a. Declaration: -

   ```
   val addScheduleIntent = Intent(this, AddScheduleActivity::class.java)
   ```

   b. Usage (Starting another activity from “this” activity): -

   ```
   startActivity(addScheduleIntent)
   ```

   c. Passing data to the new activity using Intents (via extras): -

   ```
   addScheduleIntent.putExtra(“Key”, Value)
   ```

   // Note that “key” is the index or reference name of the data that we are passing.

   //Value is the actual data value. So, we can access the value by calling it’s “Key”.

   d. Retrieving above value in another activity: -

   ```
   val receivedText = intent.getStringExtra("key")
   ```

4. Using Toast msgs: -

```
Toast.makeText(this,"Your toast msg here", Toast.LENGTH_SHORT).show()
```

//In here ".show()" is used to display the Toast. You can assign the Toast to a "Val" and then later use ".show()" to show the Toast msg as well

5. Using RecyclerView : -

   Ref - [Create dynamic lists with RecyclerView | Android Developers]()

   a. First you need to have a layout for the items (item rows) (In here "rowTextView" is included in that layout)

   b. Adapter class implementation: -

```
class MyAdapter(private val dataList: List<String>) : RecyclerView.Adapter<MyViewHolder>() {


class MyViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
   // Define references to the views in the item layout
   val textView: TextView = itemView.findViewById(R.id.rowTextView)
}
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
       // Inflate the item layout and create a ViewHolder
       val itemView = LayoutInflater.from(parent.context).inflate(R.layout.item_layout, parent, false)
       return MyViewHolder(itemView)
   }
   override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
       // Bind data to the views in each item
       val data = dataList[position]
       holder.textView.text = data
   }
   override fun getItemCount(): Int {
       // Return the number of items in the data list
       return dataList.size
   }
}
```

c. Usage (in the activity where you are going to use the recyclerView)

```
val recyclerView: RecyclerView = findViewById(R.id.recyclerView)
val data: List<String> = getData() // Replace with your data source
val adapter = MyAdapter(data)
recyclerView.adapter = adapter
recyclerView.layoutManager = LinearLayoutManager(this)
```

6. Using the SQLite database using Rooms: -

Ref - [Save data in a local database using Room | Android Developers]()

a. First you need to add dependencies to the Gradle build file(Module:app)

(I don’t think they will ask this (dependency adding) but wouldn’t risk it either. This part tends to be faulty. So, proceed with caution)

```
implementation("androidx.room:room-runtime: 2.5.1")
annotationProcessor("androidx.room:room-compiler: 2.5.1")

// To use Kotlin annotation processing tool (kapt)
kapt("androidx.room:room-compiler: 2.5.1")
// To use Kotlin Symbol Processing (KSP)
ksp("androidx.room:room-compiler: 2.5.1 ")
```

b. To use Rooms, you need to have at least 2 classes and 1 interface (Entity class, Database class, & DAO interface). Note that annotations are a must. Do not neglect the annotations.

i. Entity class: - This class act as the table schema for the database

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

ii. DAO Interface: - This interface provides methods to perform CRUD operations.

```
@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
}
```

iii. Database class: -

This class defines the database configuration and act as the main access point to the stored data. DAO interface is implemented by this class. (Use singleton design pattern to ensure only one instance of this class is created per process.)

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

iv. Usage: - (To use the Rooms, first you need to build the database. Then you can access the DAO interface methods through an object of Database class and perform Database operations)

1. Building the database: -

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

2. Accessing DAO interface methods through Database object:

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

//Please refer to the DAO interface to see the “getAll()” function

7. Fragments usage: - (refer Lab 5)

(Imagine you have created a fragment called 'userFragment.' To initiate that fragment with a button (or any element) click, you need to add the following code inside the setOnClickListener of that element.)

Like this: -

```
    val btnUser:Button = findViewById(R.id.imgUser)

    btnUser.setOnClickListener() {

        supportFragmentManager.beginTransaction().apply {
            replace(R.id.fragmentContainerView, UserFragment())
            commit()
        }
    }
```

8. Coroutines usage: - (refer tutorial 5)

```
CoroutineScope(Dispatchers.Main).launch{

            //Whatever you want to run in a coroutine can be invoke or implement within this

        }
```

Note: -

i. Use coroutines to perform database CRUD operations.

ii. Dispatchers: -

1. **Dispatchers.Default:** Use this for CPU intensive tasks (Such as complex calculations or media processing)

2. **Dispatchers.IO:** Use this for IO-bound tasks, such as reading/ writing files.

3. **Dispatchers.Main:** Use this for UI-related tasks, such as updating the value of UI elements.

**4.** **Dispatchers.Unconfined:** Use this when there's no specific task (or when you have no idea which dispatcher is the most suitable one)