Android Applications Development Using Kotlin

Java™ Education and Technology Services

Invest In Yourself , Develop Your Career

Java™ Education and Technology Services

Course Agenda

Day1

- Jetpack Compose
- Basic UI(Text, Button, Image..)
- Lists
- Modifiers

Day2

- Permissions
- Locations

Day3

- Broadcast Receivers
- Services
  - Background Services
  - Foreground Services
  - Started Services
  - Bound Services
  - IntentService
  - JobIntentService
- Notifications

Day4

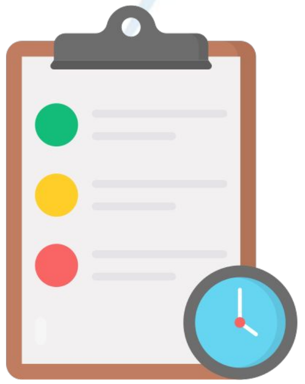
- WorkManager

Day5

- Using coroutines in Android
  - Retrofit
  - Room
  - WorkManager

Day6

- Navigation
- Layouts



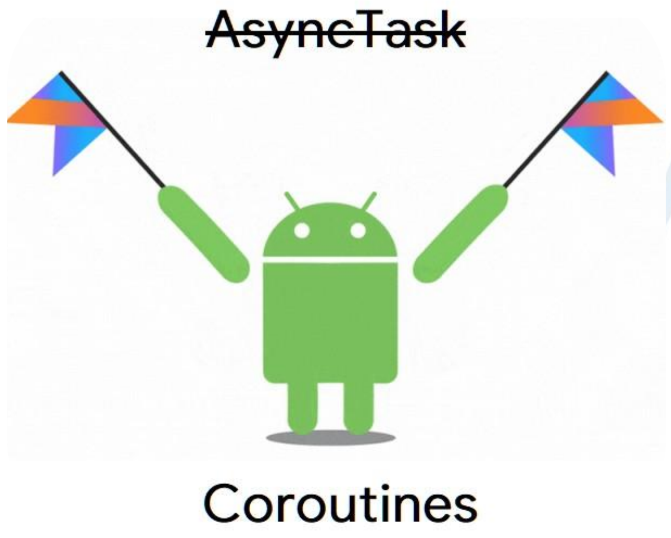
Coroutines in Android

Coroutine with Android

In Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive.

Over 50% of professional developers who use coroutines have reported seeing increased productivity.

Coroutines are used in many places that you are familiar with. For example: network calls, WorkManager ..etc.



Coroutines

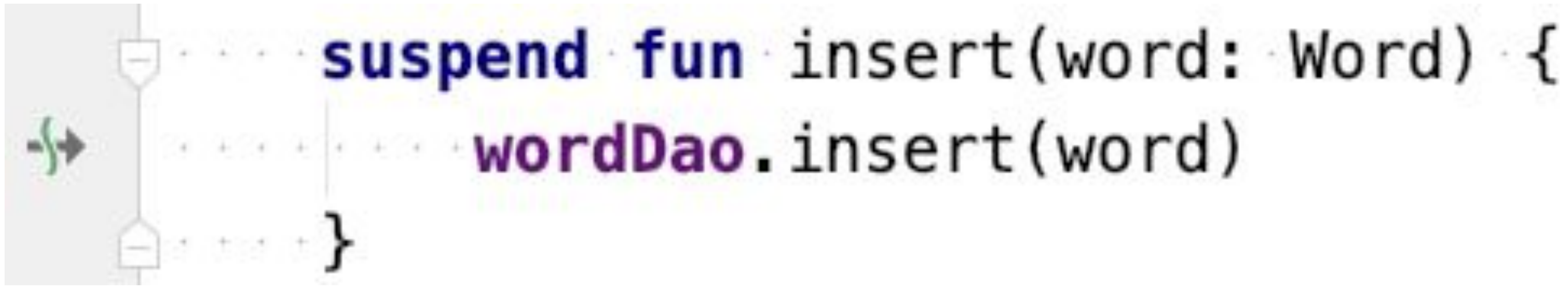
- Keep your app responsive while managing long-running tasks.
- Simplify asynchronous code in your Android app.
- Write code in sequential way
- Handle exceptions with `try/catch` block

Benefits of coroutines

- Lightweight
- Fewer memory leaks
- Built-in cancellation support
- Jetpack integration

Suspend functions

- Add `suspend` modifier
- Must be called by other `suspend` functions or coroutines



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

Coroutines – Getting Ready

Coroutine is not a part of Kotlin standard library which means that it should be added to the project.

For Adding it you can check https://github.com/Kotlin/kotlinx.coroutines

```
//Coroutines Dependencies
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.0'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.0'
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2"
```

Java™ Education and Technology Services

Control where coroutines run

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

JavaTM Education and Technology Services

withContext

```
suspend fun get(url: String) {

   // Start on Dispatchers.Main

     withContext(Dispatchers.IO) {
          // Switches to Dispatchers.IO
          // Perform blocking network IO here
     }

     // Returns to Dispatchers.Main
}
```

CoroutineScope

Coroutines must run in a `CoroutineScope`:

- Keeps track of all coroutines started in it (even suspended ones)
- Provides a way to cancel coroutines in a scope
- Provides a bridge between regular functions and coroutines

Examples: `GlobalScope`

`ViewModel` has `viewModelScope`

`Lifecycle` has `lifecycleScope`

Start new coroutines

- `launch` - no result needed

```
fun loadUI() {
    launch {
        fetchDocs()
    }
}
```

- `async` - can return a result

Coroutine - Retrofit

- As mentioned Coroutines make the heavy work easier. One of the most heavy work is network connections
- Using retrofit with Kotlin requires no extra dependencies than we used to use and makes the code more concise

```
//Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
//GSON
implementation 'com.google.code.gson:gson:2.10.1'
```

Define a Retrofit Service

```
interface SimpleService {

 @GET("posts")
 suspend fun listAll(): List<Post>

  @GET("posts/{userId}")
  suspend fun listByUser(@Path("userId") userId:String): List<Post>

  @GET("posts/search")  // becomes post/search?filter=query
  suspend fun search(@Query("filter") search: String): List<Post>

  @POST("posts/new")
  suspend fun create(@Body post : Post): Post
}
```

Set up Retrofit

```
object RetrofitHelper {

   private val retrofit = Retrofit.Builder()
         .addConverterFactory(GsonConverterFactory.create())
         .baseUrl(BASE_URL)
         .build()

   val retrofitService : SimpleService =
           retrofit.create(SimpleService::class.java)

}
```

Use Retrofit with Coroutine

```
lifeCycleScope.launch(Dispatchers.IO) {
    val postList = retrofitServie.listAll()

    withContext(Dispatchers.Main){ //To work on Main Thread

         //update the UI with your list
    }
}
```

Coroutine – Room

- Another heavy work that made easy using Coroutines is dealing with Database (Room for example)
- The following steps are needed to use Room with Kotlin:

1. In the dependencies section add the following:

```
//Room
implementation ("androidx.room:room-ktx:2.6.1“)
implementation ("androidx.room:room-runtime:2.6.1“)
ksp ("androidx.room:room-compiler:2.6.1")
```

2. Enable the Kotlin Symbol Processor (KSP) using the upcoming steps.

Java™ Education and Technology Services

Enabling Kotlin Symbol Processor (KSP)

- In your build.gradle (project) add:

```
id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
```

- In your build.gradle(module) add this to your plugins:

```
id("com.google.devtools.ksp")
```

Finally Sync your project!

Java™ Education and Technology Services

Add suspend modifier to DAO methods

```
@Dao
interface ColorDao {

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

    @Insert
    suspend fun insert(color: Color): Long

    @Update
    suspend fun update(color: Color)

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

Room DataBase

```
@Database(entities = arrayOf(Color::class), version = 1 )
abstract class ColorDataBase : RoomDatabase() {
    abstract fun getColorDao(): ColorDao
    companion object{
        @Volatile
        private var INSTANCE: ColorDataBase? = null

        fun getInstance (ctx: Context): ColorDataBase{
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    ctx.applicationContext, ColorDataBase::class.java, "color_database")
                    .build()
                INSTANCE = instance
                // return instance
                instance }
        }
    }
}
```

Using Room DataBase

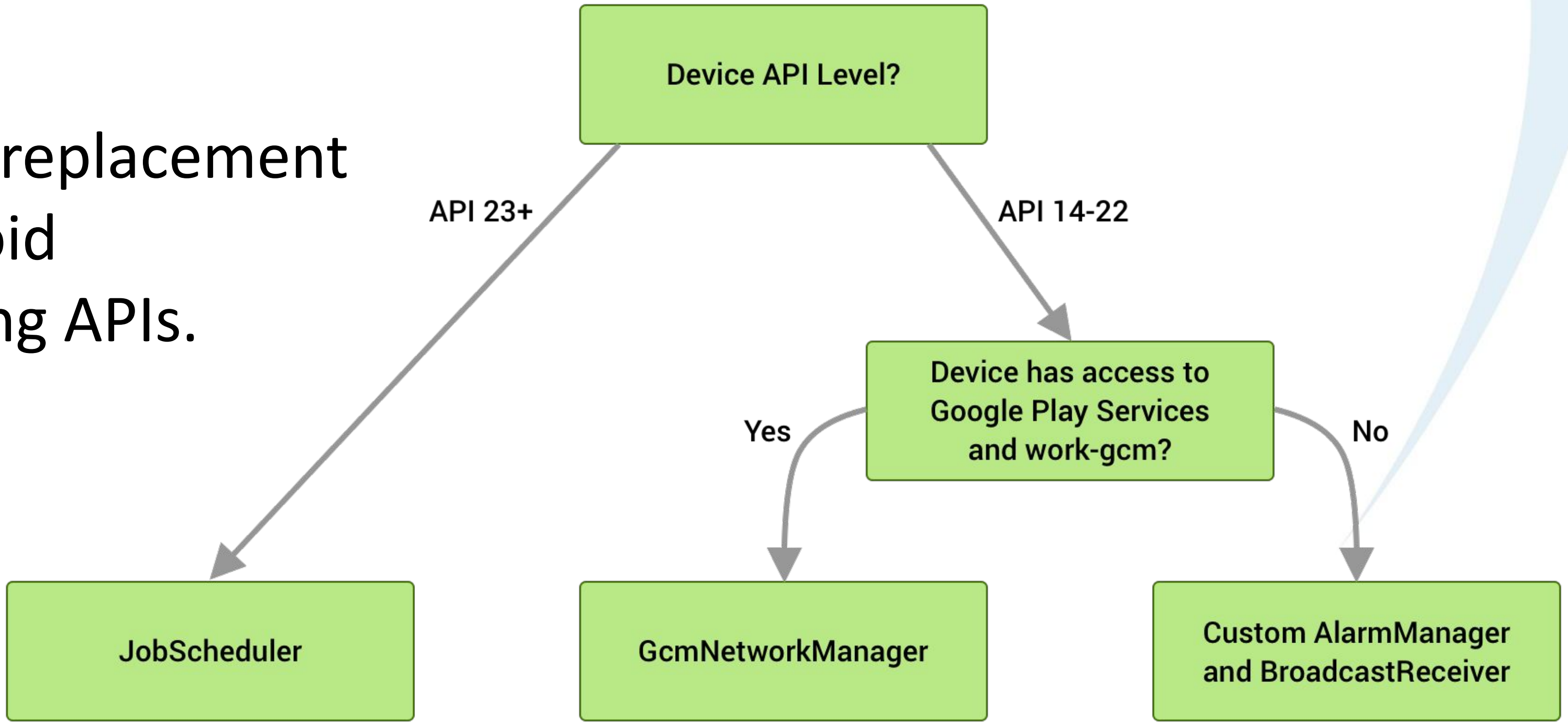
Now from your `Activity`|`Fragment` you can start a `lifeCycleScope` to launch your coroutines and interact with Room

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{ }

        val myColorDao : ColorDao =
            ColorDataBase.getInstance(this).getColorDao()

        var color = Color("Red", "0xFF0000")

        lifecycleScope.launch (Dispatchers.IO){
            val res = myColorDao.insertColor(color)
            withContext(Dispatchers.Main){
                //Update UI      }
        }

    }
}
```

WorkManager with Coroutine

WorkManager is an API that makes it easy to schedule reliable, asynchronous tasks that are expected to run even if the app exits or the device restarts.

It is a **recommended** replacement for all previous Android background scheduling APIs.


Device API Level?
API 23+
API 14-22
Device has access to Google Play Services and work-gcm?
Yes
No
JobScheduler
GcmNetworkManager
Custom AlarmManager and BroadcastReceiver

WorkManager with Coroutine

For Kotlin, WorkManager provides first-class support for coroutines.

Instead of extending Worker, you will extend **`CoroutineWorker`**, which has a **suspending** version of **`doWork()`**

CoroutineWorker run on **`Dispatchers.Default`** by default if you didn’t specify a dispatcher. (can be changed based on your logic)

Java™ Education and Technology Services

Extend CoroutineWorker instead of Worker

```
class UploadWorker(appContext: Context, workerParams: WorkerParameters) :
        CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {

        // Do the work here (in this case, upload the images)
        uploadImages()

        // Indicate whether work finished successfully with the Result
        return Result.success()
    }
}
```

Lab

Products App: is a mobile application that displays information about products. It can display fresh data fetched from the products in addition to stored data.
https://dummyjson.com/products

- The Application contains two screens:
  1. Main Screen:
     That displays a list of the products (portrait) or a list of products on the left and the details on the right (landscape)
     The row in the list composed of just icon and product name.
  2. Details Screen:
     That displays the details of the selected product.

Lab Cont'd

Check for the network status if it is:

1. **Connected**: Use Coroutine + Retrofit to fetch the data then store them to Room using Coroutines.

2. **Disconnected**: Use Room + Coroutine to show a list of the stored products that you saved to your device

For coroutine scope use `lifecycleScope`