SENG440

Week 6

Asynchronous tasks, Coroutines

Services and broadcast receivers

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

Overview

- Asynchronous tasks in Android
- Coroutines
- Flow
- Services, Broadcast Receivers and Notifications

Asynchronous tasks in Android

- **Main/UI thread** dispatches events to UI elements (widgets)
- Main/UI thread code should **always be non-blocking**
  - > 5 sec will result in **“Application not responding” dialog**
- Android UI toolkit is **not thread-safe** (never access from outside UI thread)
- In Kotlin **can use `Thread` and `Runnable`**, similar to Java, to create worker threads
- **`AsyncTask`** class simplifies the creation of workers that interact with the UI
- For projects written in Kotlin, **coroutines** are replacing AsyncTask

AsyncTask

- Create your worker by subclassing `AsyncTask`
- Implement `doInBackground()`
  - Callback function that runs in pool of background tasks
- Implement `onPostExecute()`
  - Delivers the result from `doInBackground` to the UI
  - Runs in the UI thread
- Task is run using `execute()` from the UI thread
- `onProgressUpdate` method can also be implemented to provide progress update in UI (e.g. download progress bar)

Kotlin coroutines

- `AsyncTask` is hold over from Java Android API
- Kotlin language version 1.3 provides **coroutines**
  - `AsyncTask` will become deprecated in future versions of Android API
- Coroutines **manage long-running tasks** and safely call network or disk operations
- **Dispatchers** are used to determine which thread runs a coroutine
- Coroutines are much **lighter weight than Threads** (i.e., use less memory)

Dispatchers

- **Dispatchers.Main** – used only for interacting with the UI and quick work.
  - Calling suspend functions
  - Calling UI functions
  - Updating LiveData
- **Dispatchers.IO** – optimized to perform disk or network I/O outside of the main thread
  - Database
  - Reading/writing files
  - Networking
- **Dispatchers.Default** – CPU-intensive work outside of the main thread (e.g. parsing data, sorting a large list, scientific / AI model running, etc.)

suspend functions

- suspend functions are functions that can do long running computation without blocking

```
suspend fun fetchDocs() {                          // Dispatchers.Main
    val result = get("developer.android.com")      // Dispatchers.Main
    show(result)                                   // Dispatchers.Main
}

suspend fun get(url: String) =                     // Dispatchers.Main
    withContext(Dispatchers.IO) {                  // Dispatchers.IO (main-safety block)
        /* perform network IO here */              // Dispatchers.IO (main-safety block)
    }                                              // Dispatchers.Main
}
```

Starting a new coroutine

- Can only run a suspend function from another suspend function or using `launch` or `async` to start a new coroutine
  - `launch` doesn't return a result to the caller
  - `async` allows you to return a result with a suspend function called `await`

```
fun onDocsNeeded() {
    viewModelScope.launch {    // Dispatchers.Main
        fetchDocs()            // Dispatchers.Main (suspend function call)
    }
}

suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

Built-in coroutine scopes

- `ViewModelScope`
  - Defined for each `ViewModel`
  - Automatically cancelled when `ViewModel` is cleared

  `androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0`

- `LifecycleScope`
  - Lifecycle is a class that holds the information about the lifecycle state of a component (like an activity or a fragment)
  - Coroutine launched in this scope is cancelled when the Lifecycle is destroyed.

  `androidx.lifecycle:lifecycle-runtime-ktx:2.2.0`

- `LiveData`

  Use Kotlin coroutines with lifecycle-aware components | Android Developers

  `androidx.lifecycle:lifecycle-livedata-ktx:2.2.0`

Flows

- **Suspend** functions return only one value
- A **flow** emits multiple values sequentially
- **Producer** is usually **data source** or **repository** (e.g. database, socket)
- **Consumer** is normally the `View`

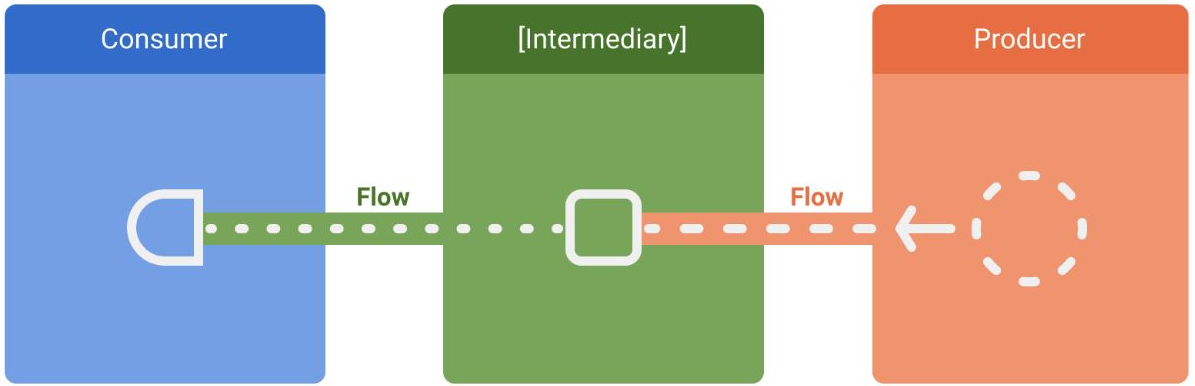


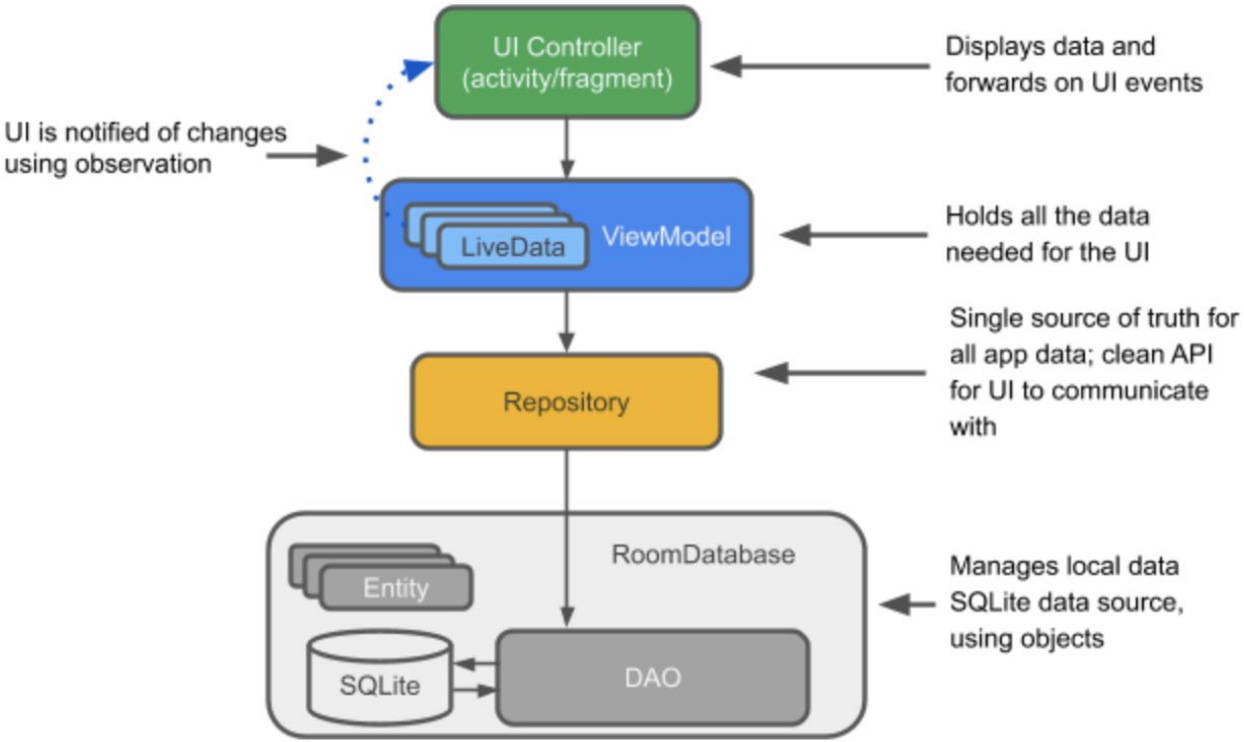
Figure 1. Entities involved in streams of data: consumer, optional intermediaries, and producer.

Kotlin Flow builder notation

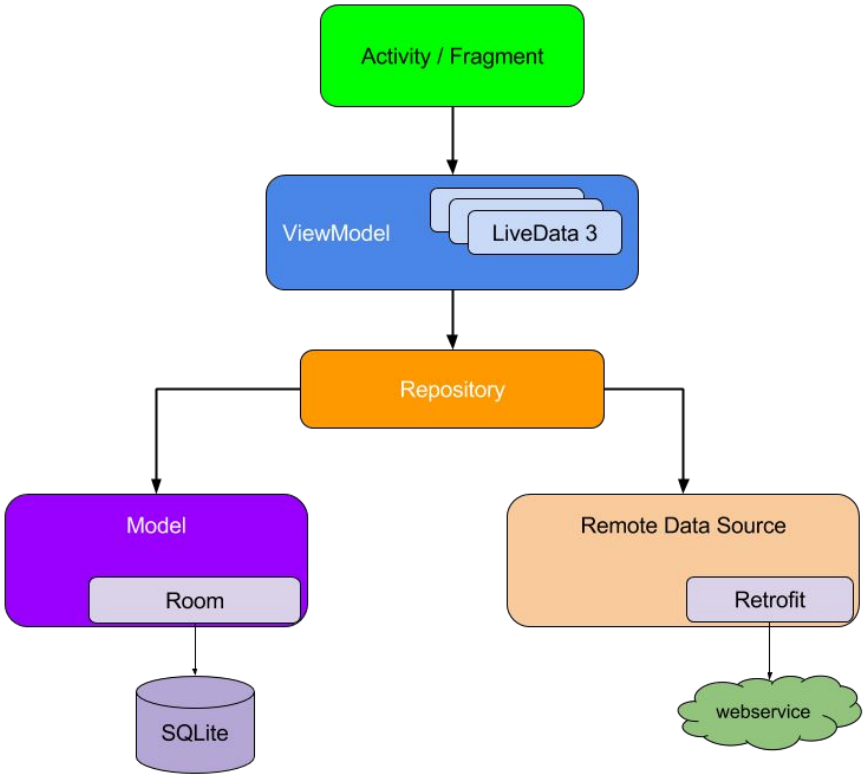
```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}
...
simple().collect { value -> println(value) }
```

- Many useful Flow methods: `collect`, `map`, `onEach`, `filter`, `count`, `drop`, etc.

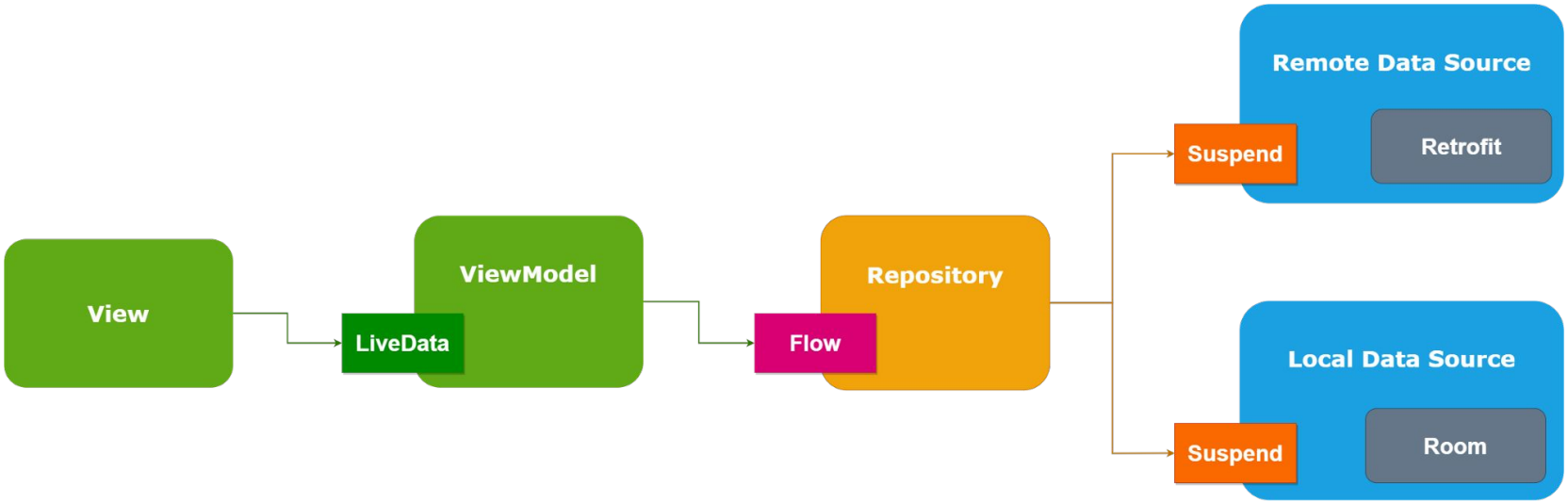
Room + Flow + LiveData + ViewModel



Room + Flow + LiveData + ViewModel



Room + Flow + LiveData + ViewModel



More reading/exercises

- Read Kotlin coroutines in Android
- Read Creating a flow and skim following sections
- Exercises:
  - For **Connect 440 Live Room** implement **Delete Friend** functionality
  - For **Connect 440 Live Room** implement **Edit Friend** functionality
  - For **Lately** implement **NewsDataSource flow** that emits a list of Guardian headlines once per minute.

Services

- Application component that performs **long-running operations in background** with no UI
- Application starts service and service **continues to run even if original application ended** or user moves to another application
- A way to run code when one of app's activities is **not the forefront activity**

Service Examples

- Download app from store
  - Leave store app, but download continues
  - Any kind of download or upload via network
- Play music even when music player dismissed (if user desires)
- Maintain connection in chat app when phone call received
- Periodically poll website for updates / changes
  - May lead to notification

Service Basics

- **No user interface** components
- Belongs to an app: **must be declared** in the app manifest
- May be **running** even if app is **not at forefront**, interacting with the user
- May be **private** (usable only by the app they belong to) or **public** (usable by apps other than yours)

Starting Services

Two ways to start services:

1. Manually by an app using a call to the API: `startService()`
   - Recall the `startActivity` method for Activities

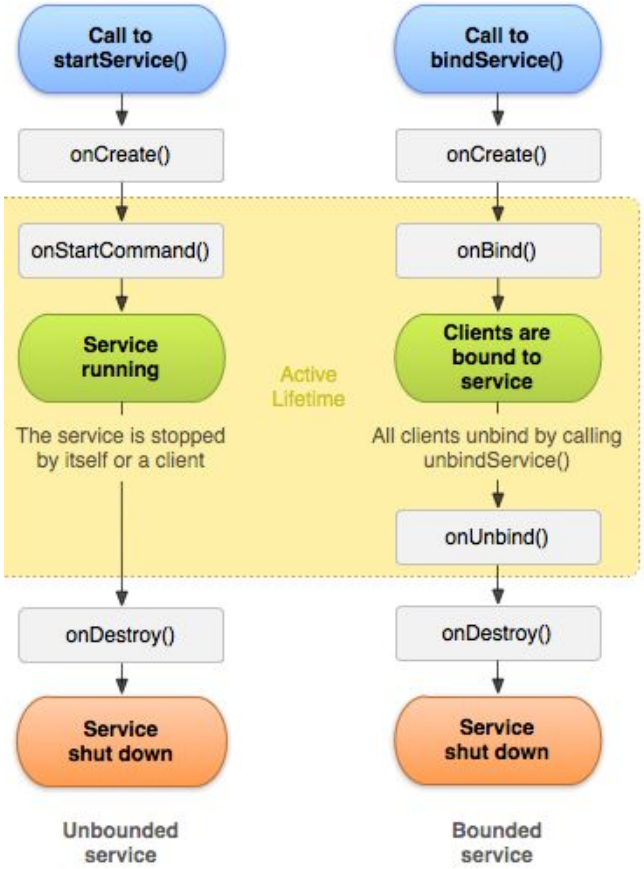
2. Another activity tries to connect (bind) to a service via inter process communication: `bindService()`
   - Could be an app other than yours if you make your service public

Stopping Services

Services will run until shut down:

1. By themselves when task completed or possibly by owning app `stopSelf`
2. By the owning app via a call to `stopService`
3. If Android needs the RAM the service is using just like it does for activities deep in the activity stack

Service Lifecycle



Service Responsiveness

- Services run on the main thread of hosting process
- By default a Service does **not** create a separate thread of execution
- For CPU-intensive or blocking code within a Service, it must still create a separate thread
- **IntentService (subclass of Service) uses a worker thread to handle start requests**

Service Types

- **Foreground**
  - Performs some operation that is noticeable to the user (e.g. play audio)
  - WorkManager API can be used to schedule deferrable, asynchronous tasks
- **Background**
  - Performs an operation that isn't directly noticed by the user
  - System imposes restrictions
- **Bound**
  - Application component binds itself to existing service via the `bindService()` method
  - Bound service provides client-server interface that allows application component to interact with service
  - Interact with service, send requests, get result via IPC (inter process communication)
  - Service runs as long as one or more applications bound to it
  - Destroyed when no applications bound

Broadcast Receivers

- The third of four application components
  - activities, services, **broadcast receivers**, content providers / receivers
- "A *broadcast receiver* is a component that **responds to system-wide broadcast announcements**."
- Android system sends **multiple kinds of broadcasts:**
  - Screen turned off, battery low, picture captured, SMS received, SMS sent, and more…

Broadcasts listed in `Intent` class

https://developer.android.com/reference/android/content/Intent#standard-broadcast-actions

String	ACTION_CAMERA_BUTTON	Broadcast Action: The "Camera Button" was pressed.
String	ACTION_CHOOSER	Activity Action: Display an activity chooser, allowing the user to pick what they want to before proceeding.
String	ACTION_CLOSE_SYSTEM_DIALOGS	Broadcast Action: This is broadcast when a user action should request a temporary system dialog to dismiss.
String	ACTION_CONFIGURATION_CHANGED	Broadcast Action: The current device `Configuration` (orientation, locale, etc) has changed.
String	ACTION_CREATE_SHORTCUT	Activity Action: Creates a shortcut.
String	ACTION_DATE_CHANGED	Broadcast Action: The date has changed.
String	ACTION_DEFAULT	A synonym for ACTION_VIEW, the "standard" action that is performed on a piece of data.
String	ACTION_DELETE	Activity Action: Delete the given data from its container.
String	ACTION_DEVICE_STORAGE_LOW	Broadcast Action: A sticky broadcast that indicates low memory condition on the device This is a protected intent that can only be sent by the system.

Permissions for broadcasts

- Android 6.0, Marshmallow, API level 23 introduced changes to permissions
- Dangerous vs. Normal permissions
- Necessary to request **Dangerous Permissions at runtime**, not install time
- E.g. listening for SMS send and receive Broadcasts is a Dangerous Permission

Receiving Broadcasts

Activities and Services can **listen for Broadcasts**

1. Subclass `BroadcastReceiver` (implement `onReceive` method)
2. Create `IntentFilter` object to specify the kinds of Broadcasts you want
3. Register receiver (`onResume()` of Activity)
4. Unregister receiver (`onPause()` of Activity)

Alternatively use `manifest.xml` to register receiver but for some broadcasts that is not allowed (e.g. `ACTION_TIME_TICK`)

Spoofing startup (broadcasts)

- Painful to shut down and start up device
- Possible to spoof broadcasts
- Recommend using emulator
- Go to terminal
- adb shell

```
C:\Users\scottm\AndroidStudioProjects\Service_Example_SMS_Responder>adb shell
root@android:/ # am broadcast -a android.intent.action.BOOT_COMPLETED
am broadcast -a android.intent.action.BOOT_COMPLETED
Broadcasting: Intent { act=android.intent.action.BOOT_COMPLETED }
Broadcast completed: result=0
root@android:/ #
```

Initiating broadcasts ourselves

- Applications can initiate broadcasts to **inform other applications of status or readiness**
- **Don't display UI**
  - Can create status bar **Notifications**
- Usually just a **gateway to other components** and does very minimal work
  - **Initiate service** based on some event
- Broadcasts are **delivered as Intents**

More on Broadcast Receivers

- Intents sent by `sendBroadcast()` method
- `LocalBroadcastManager` to send Broadcasts within your application only
- Can't initiate asynchronous actions in `onReceive`
  - Like running a `suspend fun`
  - Because when method done `BroadcastReceiver` no longer active and system can and will kill the process
- ***Might need to use Notification Manager or start a service***

Notifications

- Message shown outside of app's UI

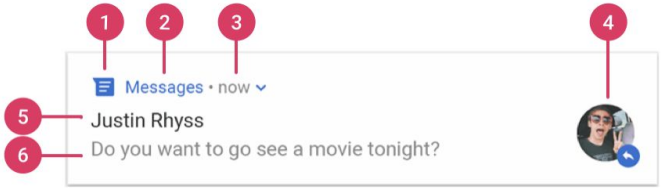


Figure 7. A notification with basic details

The most common parts of a notification are indicated in figure 7 as follows:

1. Small icon: This is required and set with `setSmallIcon()`.
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with `setWhen()` or hide it with `setShowWhen(false)`.
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with `setLargeIcon()`.
5. Title: This is optional and set with `setContentTitle()`.
6. Text: This is optional and set with `setContentText()`.

Notification Builder

```
var builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle(textTitle)
        .setContentText(textContent)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

- Many options, see Create a Notification | Android Developers

Notification Channel for Activity

```
private fun createNotificationChannel() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        // Register the channel with the system
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```