COM4510/6510 Software Development for Mobile Devices

Lecture 6: Architecture Components: Android JetPack

Temitope (Temi) Adeosun
The University of Sheffield
t.adeosun@sheffield.ac.uk

The University Of Sheffield.

Lecture Overview

- Week 2: Intro to Kotlin
- Week 3: Lifecycle and Layouts
- Week 4: (Architectural) Design Patterns
- Week 5: Persisting Data
- **Week 6: Architectural Components**
- Week 7: Sensing in Android
- Week 8: Background and Foreground Services
- Week 10: Context
- Week 11: Releasing Apps
- Week 12: Guest lecture

Lecture Overview

- Keeping Your App Responsive
- Towards True Mobile Computing
- Connecting to a server

- Lab tutorial :
    - MVVM, Live Data and Persistence (a simple but complete example)

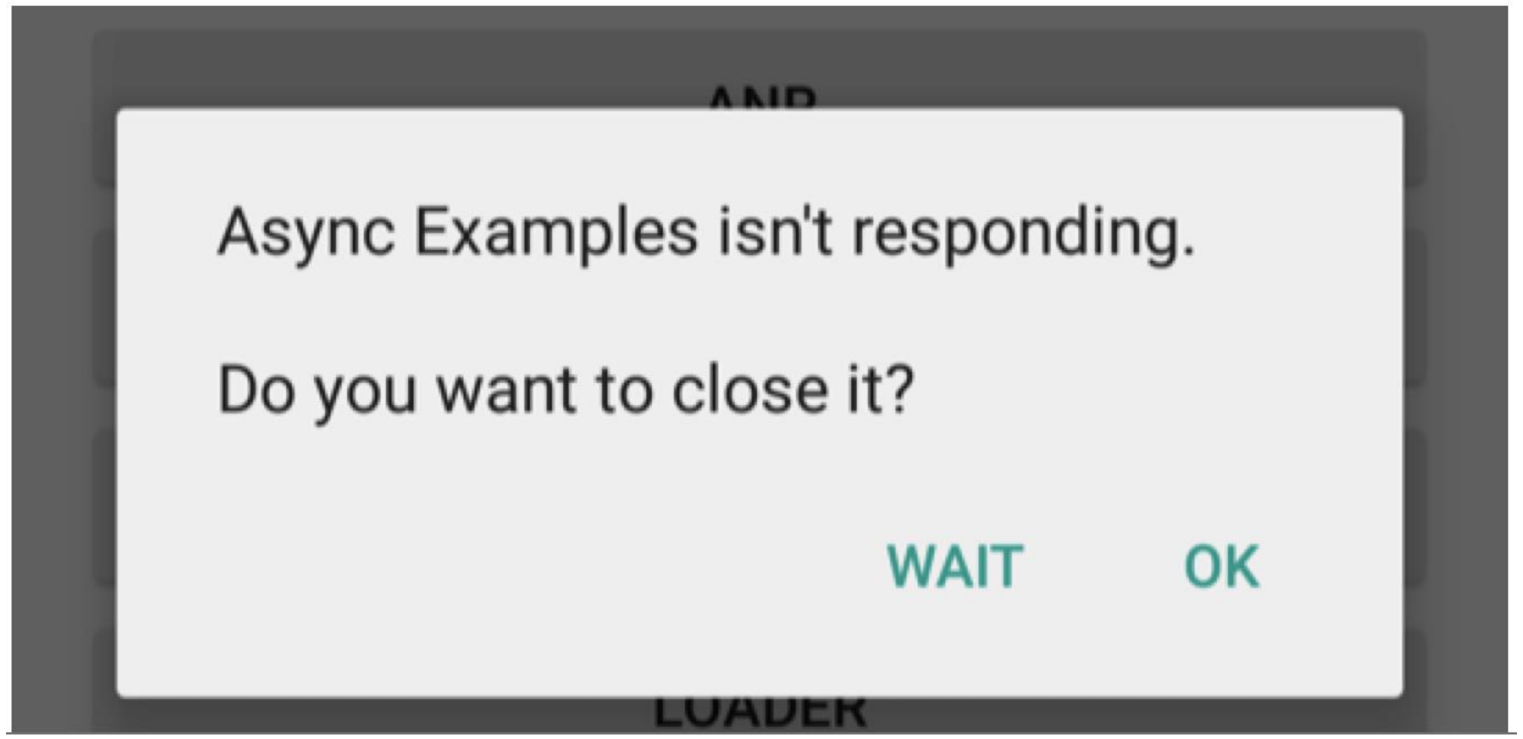
The University Of Sheffield.

Keeping Your App Responsive

https://developer.android.com/training/articles/perf-anr.html

Sluggish Response and ANRs

- The worst thing that can happen to your app's responsiveness is an "Application Not Responding" (ANR) dialog



Async Examples isn't responding.

Do you want to close it?

- It's critical to design responsiveness into your application
  - so the system never displays an ANR dialog to the user

The University Of Sheffield.

Sluggish Response and ANRs

- Android displays an ANR if an application cannot respond to user input
  - e.g. if an application blocks on some I/O operation (e.g. a network access) on the UI thread
  - so the system can't process incoming user input events
  - e.g. the app spends time building an elaborate in-memory structure on the UI thread
  - such computations may be efficient, but even the most efficient code still takes time to run!!

Do not overuse the UI Thread

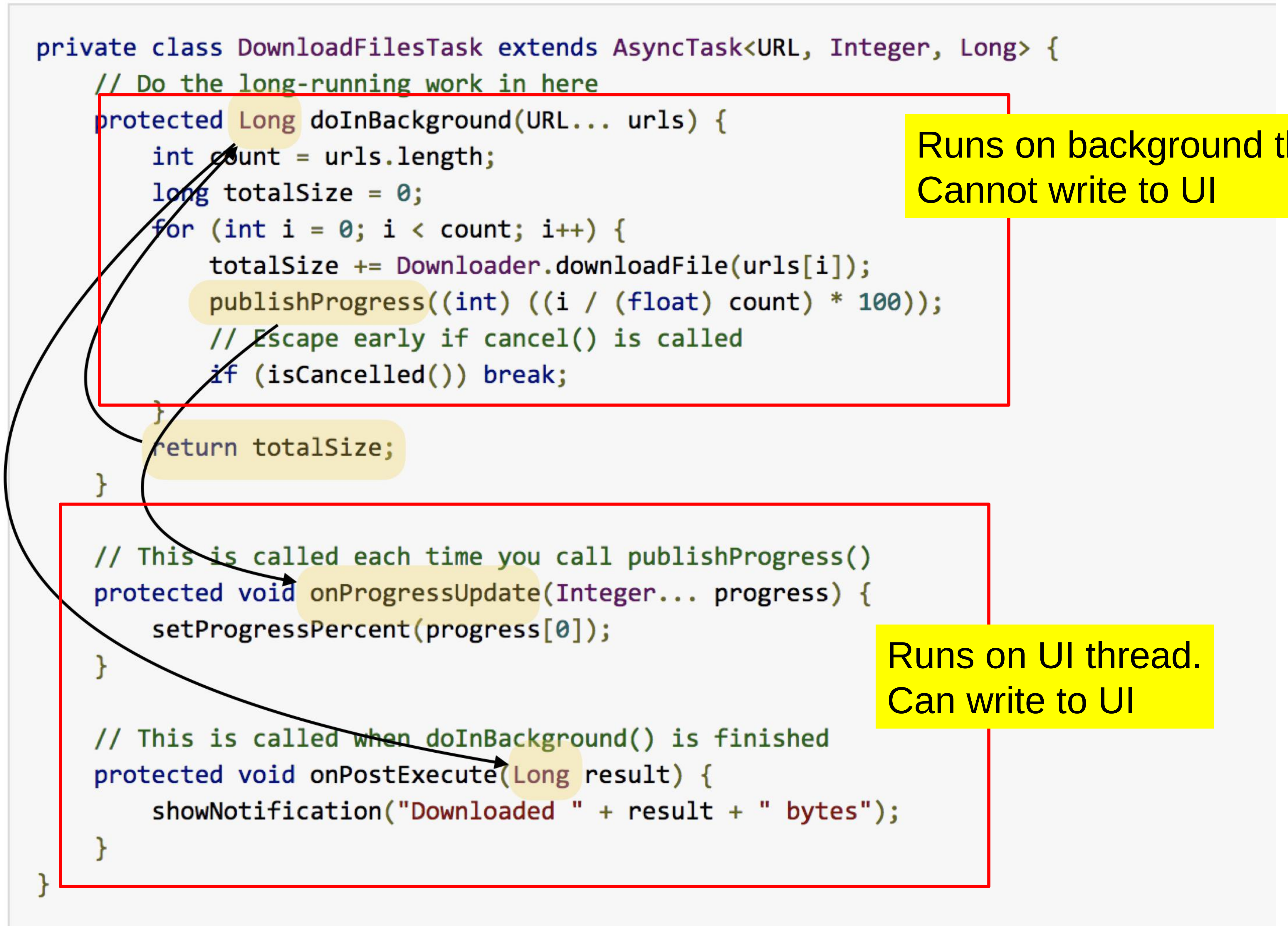
- Android applications normally run entirely on a single thread
  - by default the "UI thread" or "main thread"
- Anything your application is doing in the UI thread that takes a long time can trigger the ANR dialog
- Android will display the ANR dialog if:
  - No response to an input event (such as key press or screen touch events) **within 5 seconds**
  - A BroadcastReceiver hasn't finished executing **within 10 seconds**
- Any method running in the UI thread should do as little work as possible
  - Activities/Fragments should do **as little as possible** to set up in key life-cycle methods such as **onCreate() and onResume()**

The University Of Sheffield.

Process Asynchronously

- To avoid blocking the main thread, traditionally we would
  - start new worker threads to run long running processing in the background.
    - such as network or database operations,
    - computationally expensive calculations e.g. resizing bitmaps
  - Or use other asynchronous processing techniques such as Callbacks, Future promises etc.
- Next slide shows an example (in Java) starting a long running process using the AsyncTask class with callbacks (now Deprecated). Starting threads have some drawback:
  - Threads are expensive and requires expensive context switching
  - Threads are limited resources and can cause memory leaks
  - Threads are hard to debug and susceptible to race conditions.
  - Callbacks can become difficult to manage i.e. Callback hell

Using AsyncTask

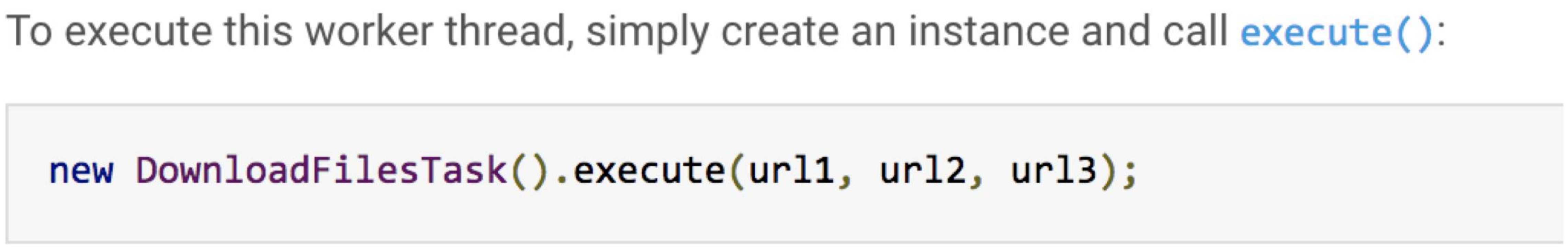


```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

Runs on background thread.
Cannot write to UI

Runs on UI thread.
Can write to UI

Using AsyncTask



To execute this worker thread, simply create an instance and call `execute()`:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

- Note, using Kotlin callback greatly simplifies but still ladened with problems. See here for some details

```
longRunningTask(1, 2){
   // Called after longRunningTask is completed

}
```

```
fun longRunningTask(param1: Int, param2: Int,
taskCallback: (String) -> Unit){
    val t = thread{
        // Do some long running task

        val taskResult = "task result"
        taskCallback(taskResult)
    }

}
```

Still relevant notes

- What is a thread ?
  A thread defines a process running
- What is UI Thread
  - Main thread of execution for your Android application
  - Most of your application code will run here onCreate, onPause, onDestroy, onClick, etc.
  - So simply Anything that causes the UI to be updated or changed HAS to happen on the UI thread
- When you explicitly spawn a new thread or an async task to do work in the background
- the code is NOT run on the UIThread, so YOU cannot modify the UI. Need to access UI thread from a background thread?
  - Don't! Means you are doing something wrong

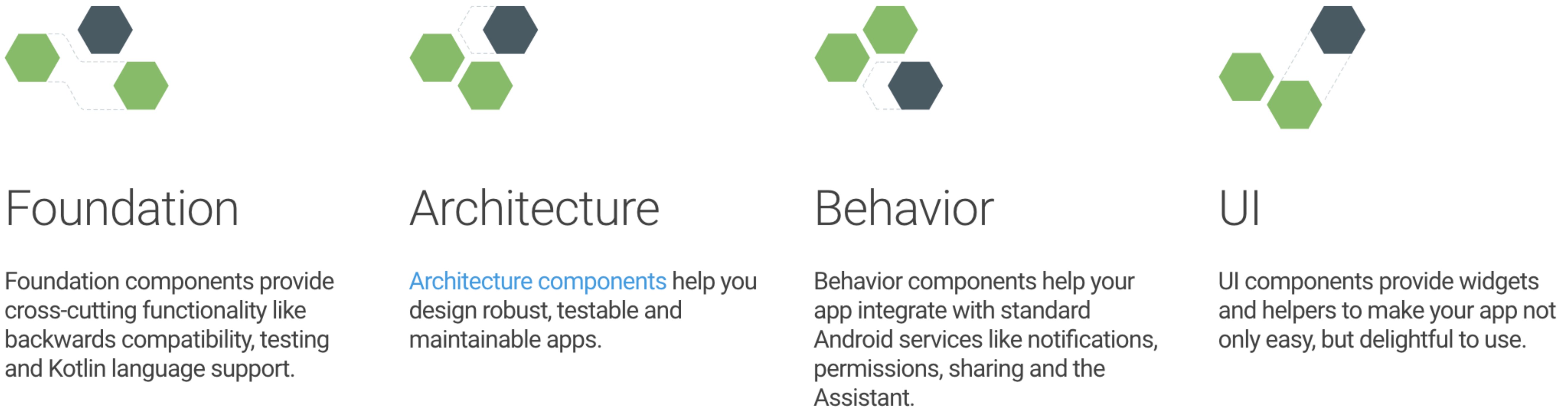
Towards True Mobile Computing

Changing

- Android has been changing rapidly to address issues that have emerged in some 10 years+ of development
    - From a Java/Linux environment to a fully fledged mobile environment
- Kotlin: as a language specifically designed to avoid common pitfalls
    - e.g. null pointer exceptions and Java verbosity
- Jetpack to create a mobile specific approach, to simplify application development and to avoid errors

Jetpack

- a collection of Android software components
- eliminating boilerplate code



Foundation

Foundation components provide cross-cutting functionality like backwards compatibility, testing and Kotlin language support.

Architecture

Architecture components help you design robust, testable and maintainable apps.

Behavior

Behavior components help your app integrate with standard Android services like notifications, permissions, sharing and the Assistant.

UI

UI components provide widgets and helpers to make your app not only easy, but delightful to use.

https://www.youtube.com/watch?v=7p22cSzniBM https://developer.android.com/jetpack/

ANR and Coroutines

- Using Coroutine is the recommended approach to concurrency and multithreading in Android.
  - Coroutines is essentially a library that allows programmers to run non-blocking background processes in an easier to manage, main safe fashion.
  - Starting a coroutine starts a new background job, running on a thread.
    - Coroutine are lightweight – doesn't not have to run on new thread, as threads/thread pools are reusable. So, does not result in out-of-memory error as you may get when starting many threads.

https://kotlinlang.org/docs/coroutines-guide.html

Important Coroutine ideas

- Coroutine scope and Coroutine context:
  - Every Coroutine runs within some context, which typically is a combination of the coroutine job and dispatcher.
  - Scope and context are essentially the same thing – just different words to for different usage – the scope is the context from which the thread is started. The context is the explicit context with which we start a coroutine
    - Starting a coroutine without specifying an explicit context will start the coroutine in the current scope, which is typically the main thread
  - A job offer controlled over the coroutine – start, wait, cancel. See here for more.

The University Of Sheffield.

Important Coroutine ideas

- Three key dispatchers determines the thread/threads on which a coroutine will run.
  - Dispatchers.Main: Runs coroutine on the main thread. Use only where Interaction with the UI is needed and processing is quick.
  - Dispatchers.IO: uses thread pool optimized for disk and network IO.
  - Dispatchers.Default: optimized for CPU intensive process outside the main thread.

The University Of Sheffield.

Important Coroutine ideas

- Coroutines are suspending functions:

  ```
  suspend fun longRunning(){…}
  ```

  - A suspending function can only be called from another suspending function
  - This could cause a chain of suspending function which go all the way to the main function!
  - Interrupt this by using one of the Coroutine primitives:
    - runBlocking – will block the calling thread
    - launch – starts a new coroutine without return result to a caller
    - async – starts a new coroutine and allows returning result to caller using .await()

https://developer.android.com/kotlin/coroutines/coroutines-adv

Important Coroutine ideas

- Jetpack lifecycle aware components provide native coroutine support and provide default scope:
  - ViewModelScope – defined for ViewModel. Needs:
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$ktx_ext_ver"
  - LifecycleScope – defined for LifeCycle objects such as Activity and Fragments. Needs:
    implementation "androidx.lifecycle:lifecycle-runtime-ktx:$ktx_ext_ver "
  - LiveData – used with LiveData objects. Needs:
    plementation "androidx.lifecycle:lifecycle-livedata-ktx:$ktx_ext_ver "

[illegible]

Important Coroutine ideas

- Avoid starting a coroutine from:
  - GlobalScope:
    - You can't control the job.
    - Its lifecycle might live longer than your activity or fragment
- Instead start coroutines from:
  - viewModelScope

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            // Coroutine that will be canceled when the ViewModel is cleared.
        }
    }
}
```

Important Coroutine ideas

- LifecycleScope

- class MyFragment: Fragment() {

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        …
            viewLifecycleOwner.lifecycleScope.launch {
            // Coroutine that will be canceled when the LifeCycle object is cleared.
            }
    }
}
```

- CoroutineScope: define scope property in class

```
val scope = CoroutineScope()
scope.launch{
// Coroutine that will be canceled when the Object is cleared
}
```

Architecture

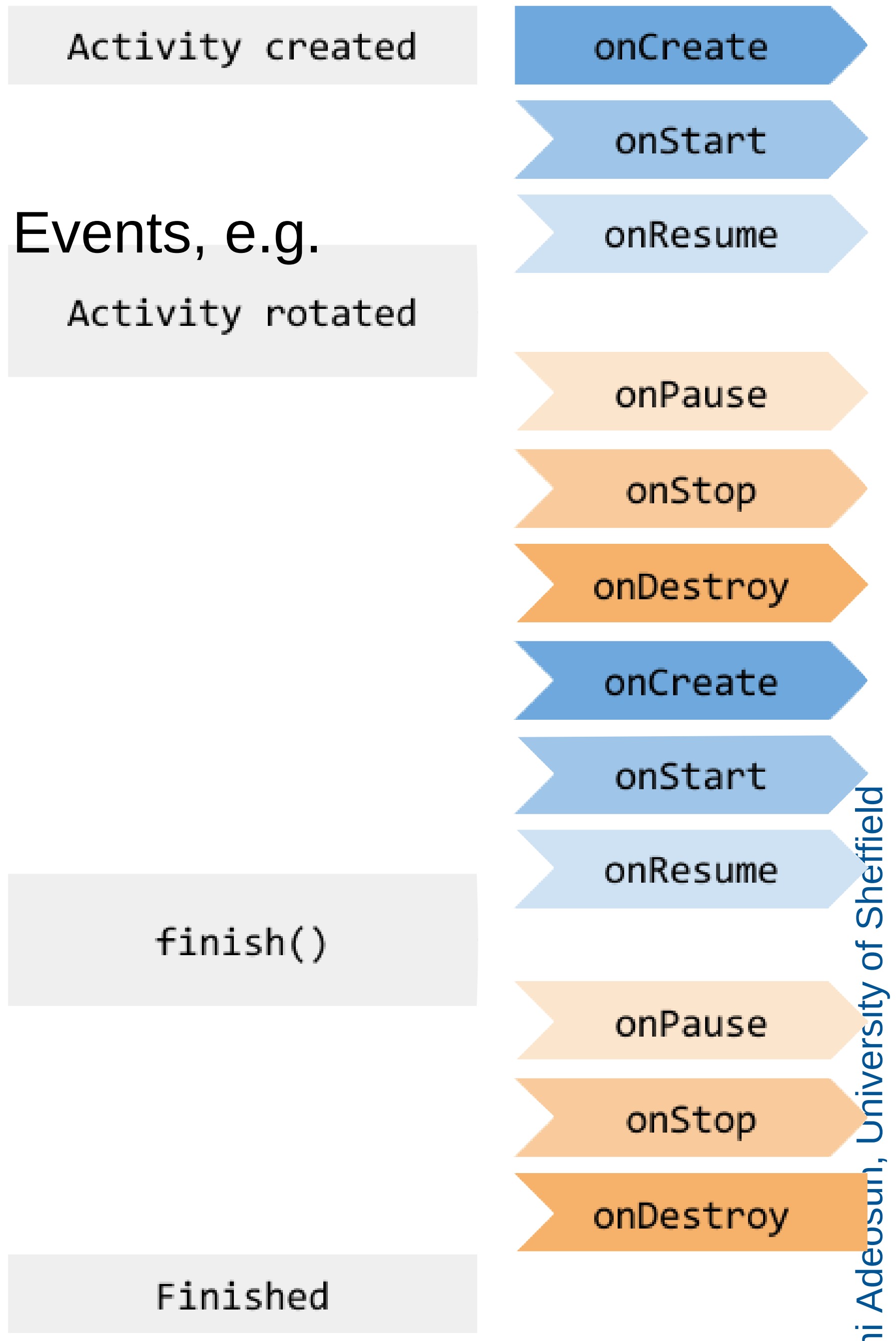
- Manage your app's lifecycle with ease
  - New lifecycle-aware components help you manage your activity and fragment lifecycles
  - Survive configuration changes, avoid memory leaks and easily load data into your UI.
- Use LiveData to build data objects that notify views when the underlying database changes (use Flow for stream data).
- ViewModel to store UI-related data that isn't destroyed throughout the app's lifecycle
- Room for databases

Patterns

- We have seen different patterns to simplify the relation between Model and UI (Week 4)
  - Jetpack extends this concept to new levels
- MVVM:
  - Model-View-ViewModel (MVVM) which is at the basis of current Android developments

Issues with the UI controller

- Activities have a lifecycle managed by Android
  - user actions and external actions let the view move through different statuses



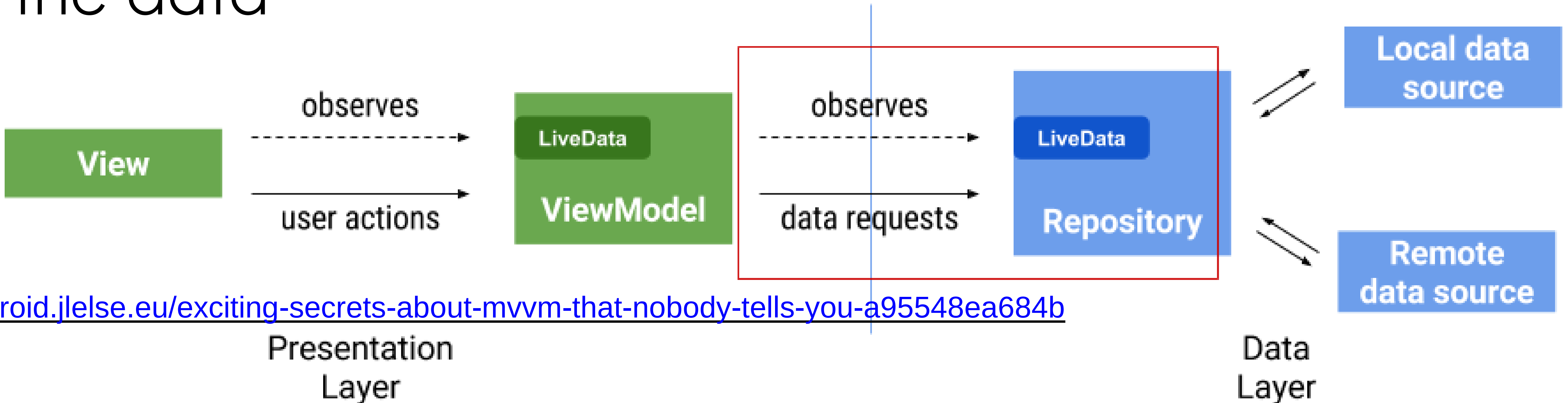
https://developer.android.com/topic/libraries/architecture/viewmodel

Issues

- Data is traditionally stored in the UI controller, i.e. it is bound to the interface
  - for example all the data displayed in a RecyclerView will be destroyed if the phone is rotated and will need to be re-fetched and recreated
- UI controllers frequently need to make asynchronous calls that may take some time to return
  - if the user navigates away, when the async call returns there is no interface to show it with
    - null pointer exception or memory leaks
- Assigning excessive responsibility to UI controllers
  - can result in a single class that tries to handle all of an app's work by itself, instead of delegating work to other classes.
- it also makes testing a lot harder

Model View ViewModel (AndroidX.Lifecycle.ViewModel)

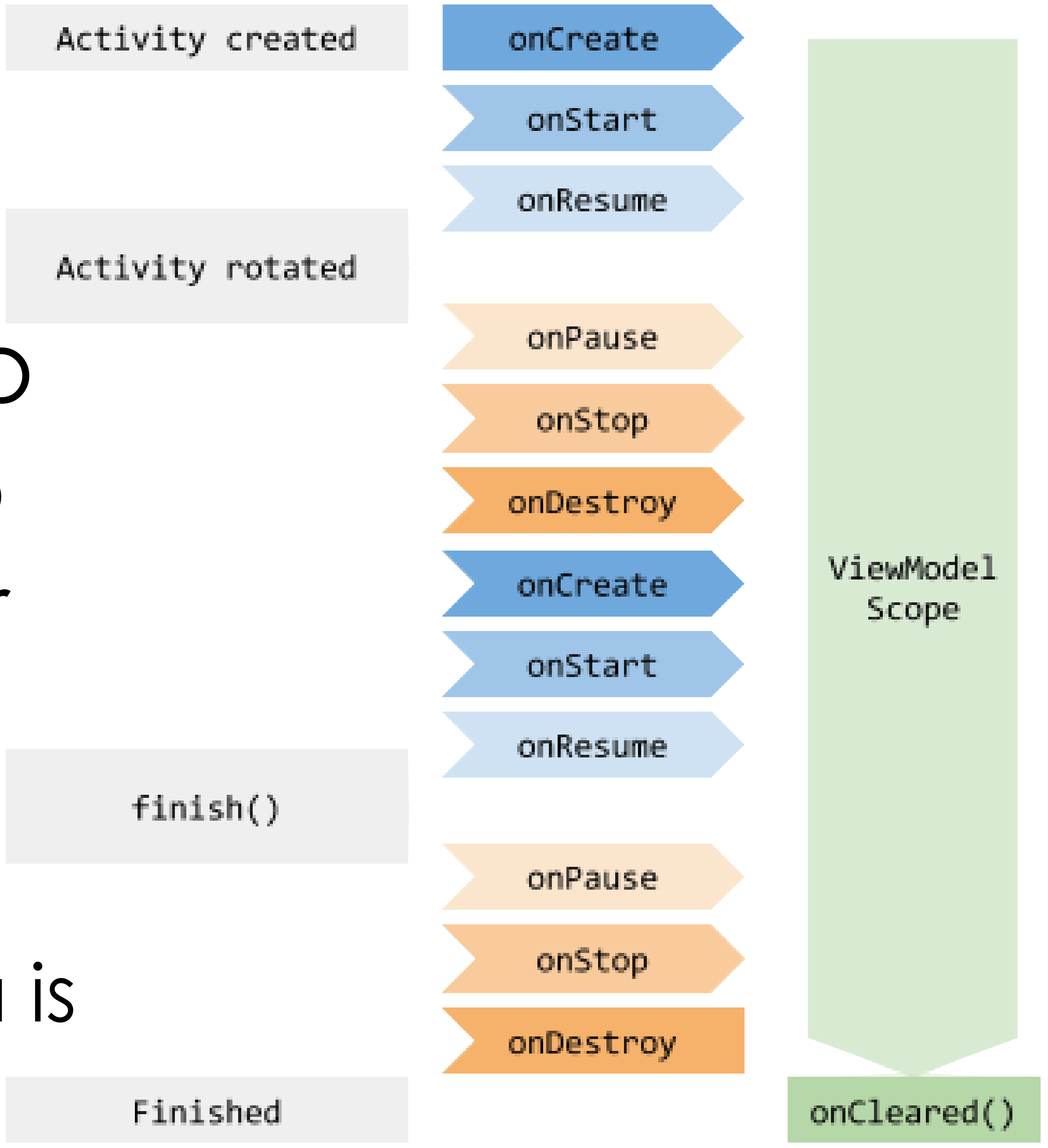
- The view does not call methods in the ViewModel
  - as it happened in MVP
  - there is no dependency on the View for the VM
- The communications
  - View > ViewModel > Model is direct
  - Model > ViewModel > View is done via "**Observing**" the data



tps://android.jlelse.eu/exciting-secrets-about-mvvm-that-nobody-tells-you-a95548ea684b

Lifecycle of ViewModel (androidX.lifecycle.ViewModel)

- It persists across changes of state of the UI controller
  - hence not requiring to reload/recreate the data
- As the UI is subscribing to the VM data, there is no way to have null pointer exceptions or memory leaks
  - if the UI disappears, the data is not observed



LiveData

- LiveData is an observable data holder class
  - It is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services.
  - This awareness ensures LiveData only updates app component observers that are in an active lifecycle state
- UI components subscribe to the live data and are immediately notified of any change
  - only if they are in a active state
    - e.g. activity is in foreground

https://developer.android.com/topic/libraries/architecture/livedata

Advantages of Live Data

- Ensures your UI matches your data state
- LiveData follows the observer pattern
- No memory leaks
  - Observers are bound to Lifecycle objects and clean up after themselves when their associated lifecycle is destroyed.
- No crashes due to stopped activities
  - If the observer's lifecycle is inactive, such as in the case of an activity in the back stack, then it doesn’t receive any LiveData events.

Advantages

- No more manual lifecycle handling
  - UI components just observe relevant data and don't stop or resume observation. LiveData automatically manages all of this since it's aware of the relevant lifecycle status changes while observing.
- Always up to date data
  - If a lifecycle becomes inactive, it receives the latest data upon becoming active again. For example, an activity that was in the background receives the latest data right after it returns to the foreground.
- Proper configuration changes
  - If an activity or fragment is recreated due to a configuration change, like device rotation, it immediately receives the latest available data.

How To

- Create an instance of LiveData
  - to hold a certain type of data.
  - in yourViewModel class.
- Create an Observer object that defines the onChanged() method
  - create an Observerobject in an activity or fragment
- Attach the Observer object to the LiveData object using the observe() method
  - which subscribes the Observer object to the LiveData object so that it is notified of changes.
  - in an activity or fragment

View: displaying a list of objects of type Strings

- In your Activity connect to the Model and set the live data change

```
override fun onCreate(savedInstanceState: Bundle?) {
        …

    // Get a new or existing ViewModel from the ViewModelProvider.
    this.myViewModel = ViewModelProvider(this)[MyViewModel::class.java]


    // Add an observer on the LiveData. The onChanged() method fires
    // when the observed data changes and the activity is
    // in the foreground.
    this.myViewModel!!.getNumberDataToDisplay()!!.observe(this,
        //  create observer, whenever the value is changed this func will be called
        {

            //  update the UI
        }
    )
}
```

ViewModel

- it must extend ViewModel/AndroidViewModel

```
class MyViewModel(application: Application) : AndroidViewModel(application) {
    // Create a variable of type LiveData<Type>
    private var strList: LiveData<List<String>>?
    // Create a connection to the Repository (Model) and initialize it
    private lateinit var mRepository: MyRepository
    // Create a getter to
    fun getStringList(): LiveData<List<String>>? {
        if (this.numberDataToDisplay == null) {
            // launch operation in the Model that fetches the data the first time (usually an asyc
operation)
        }
        return this.strList
    }
}
```

- Initialize the Model (Repository)

```
init {
    this.myRepository = MyRepository(application)
    this.strList = this.mRepository.valueGetter()
}
```

Connecting to a server

Presented for awareness, not fully covered

Sending data to a server

- Sending data to a server is needed to synchronise two processes
  - the client (the mobile)
  - the server (e.g. a node server)
- Sending data is a very heavy operation
  - it should be minimised
  - it should run at appropriate times
  - it must check for connectivity
  - it must not block the UI thread
  - infrequent sending of large blocks of data is more efficient than frequent small data sending

Sending data

- You can implement your own data sending if you really must
- But easier if you use an existing library
  - Retrofit – preferred because of Architecture component support - LiveData. Other good option – OKHttp (but Retrofit uses OKHttp and makes your work easier)
- use WorkManager from the JetPack to
  - allows sync between a local and remote database
  - very efficient, very effective, battery saving

Check out this Colab for a simple Retrofit + MVVM + LiveData implementation.

Summary

- Keeping Your App Responsive – using Coroutines
- Towards True Mobile Computing – Android Jetpack
- Connecting to a server – Quick overview (Retrofit + Workmanager)

- Lab tutorial :
  - MVVM, Live Data and Persistence (a simple but complete example)