MOBILE DEVELOPMENT

Process Management

Concurrency and Threading in Android

Quick Disclaimer

- There’s a lot of information to digest when it comes to process management in Android
    - Commonly used components being depreciated…
    - New components / architectures being introduced…
- So… discussion is going to be somewhat general
    - If you need to learn more, you’ll have an idea of concepts that should be kept in mind as you wade through

Outline

- Brief overview of concurrency
- Android app’s main thread
- Moving to worker threads
- General design considerations

What comes to mind when you hear **concurrency**?

Concurrency

- Process by which multiple processes run at the same time
- Concurrent vs serial execution
- Processes can contain one or more threads

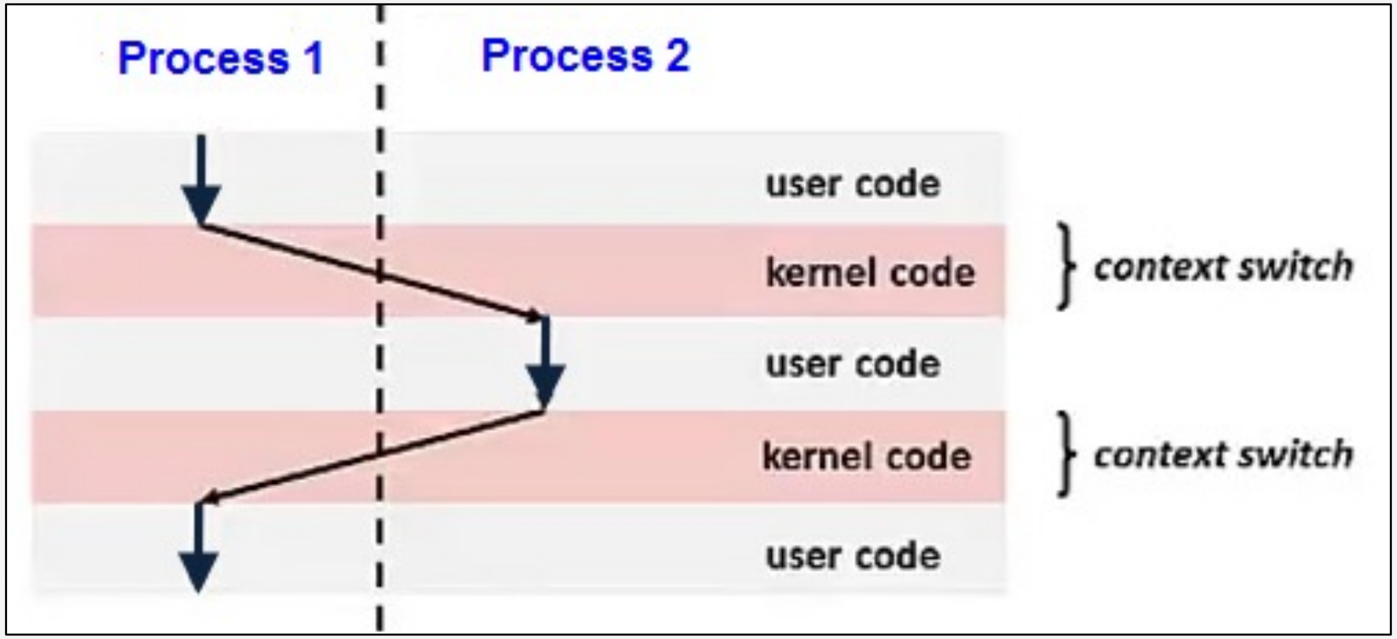


Image source: https://www.bogotobogo.com/cplusplus/multithreaded.php

What situations would the usage of **multiple threads** prove useful?

Common Uses

- Downloading or uploading data
- Sending a network request
- Saving data to a database
- Executing tasks in the background
- Any long / slow running tasks

Any kind of task that might block / temporarily stop / get in the way of the user interacting with the UI

Why Study Concurrency?

- By default, an Android app runs on a single thread called the main thread
  - Sometimes called the UI thread
  - Its main task is to oversee user interactions and UI output
- With a single thread, all methods are executed synchronous
  - In other words, execute one task at a time in queue like fashion

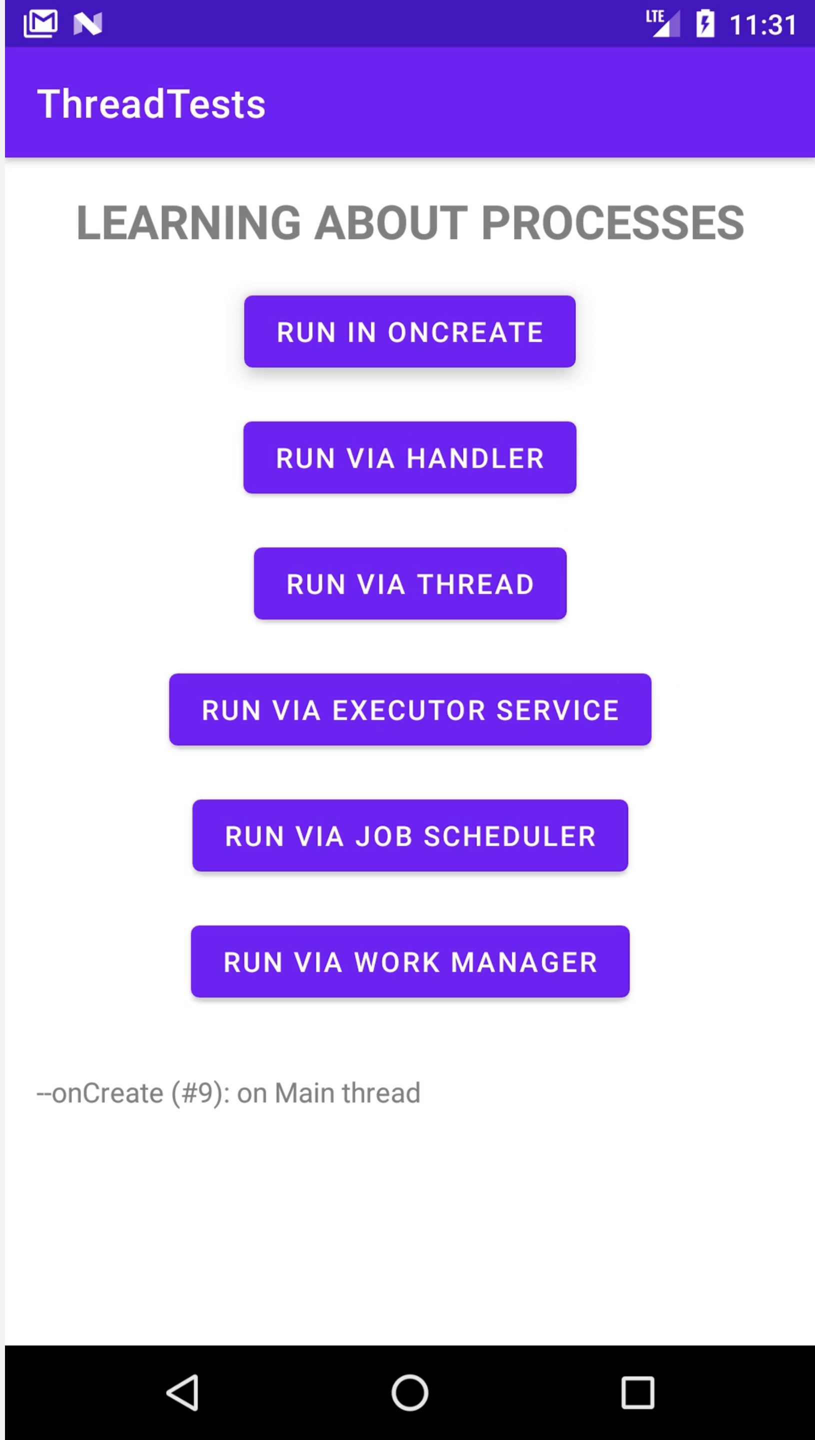
Why Study Concurrency?

- However, if the main thread is blocked for ~5 seconds, the user will be presented with a “Application not responding” dialog

- Downloading or uploading data
- Sending a network request
- Saving data to a database
- Executing tasks in the background
- Any long / slow running tasks

Not structuring solution to these situations might lead to your app freezing as shown

Blocking the main thread means stopping onDraw() or updating the displayed views from taking place



11:31

ThreadTests

LEARNING ABOUT PROCESSES

--onCreate (#9): on Main thread

Moving Off the Main Thread

- The easiest way to move off the main thread is just to utilize a Thread object
- Pass it a Runnable object with your task in the run() method and you're good to go!

```
this.someBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // your operation
            }
        }).start();
    }
});
```

For simplicity, we'll call any thread that isn't the main or UI thread as a <u>**worker thread**</u>

Moving Off the Main Thread…

- However, there are three things you have you keep in mind:
  1. Worker threads cannot access any Views
  2. You'll need to clean up thread operations when they're no longer meant to be running
  3. Threads are killed with the app's process is killed

1. Worker Threads can’t Access Views

- As mentioned, the main thread is responsible for handling the user interface
  - Does so by interacting with the Android UI toolkit
  - … which apparently isn’t thread-safe…
- Hence, all updates to our Views/Widgets need to run on the main thread

But if we can't update UI from a separate thread, how do we access the UI thread from a worker thread?

UI Thread Access

- We have a few options:
  - `Activity.runOnUiThread(Runnable)`
    - Or just `runOnUiThread()` in an Activity
  - `View.post(Runnable)`
  - `View.postDelayed(Runnable, long)`

UI Thread Access

```
 public void onClick(View v) {
        new Thread(new Runnable() {
            public void run() {
                // a potentially time-consuming task
                final Bitmap bitmap = processBitMap(”its_me_dio.png");
                imageView.post(new Runnable() {
                    public void run() {
                        imageView.setImageBitmap(bitmap);
                    }
                });
            }
        }).start();
    }
});
```

Long running operation

Updates to the user interface

If you try to update a view directly from a worker thread, Android will through an error

From the Android Documentation: https://developer.android.com/guide/components/processes-and-threads

UI Thread Access

- But what if we don’t have access to a View, Activity, or Context?

- Alternative: We can make use of a Handler object…

…BUT FIRST… we need to understand
what a Looper is

Looper

The associated thread of a looper doesn't need to the be Main Thread. You can create a new Thread with its own Looper.

- Think of a Looper as a continuously running thread that queues messages or tasks to execute

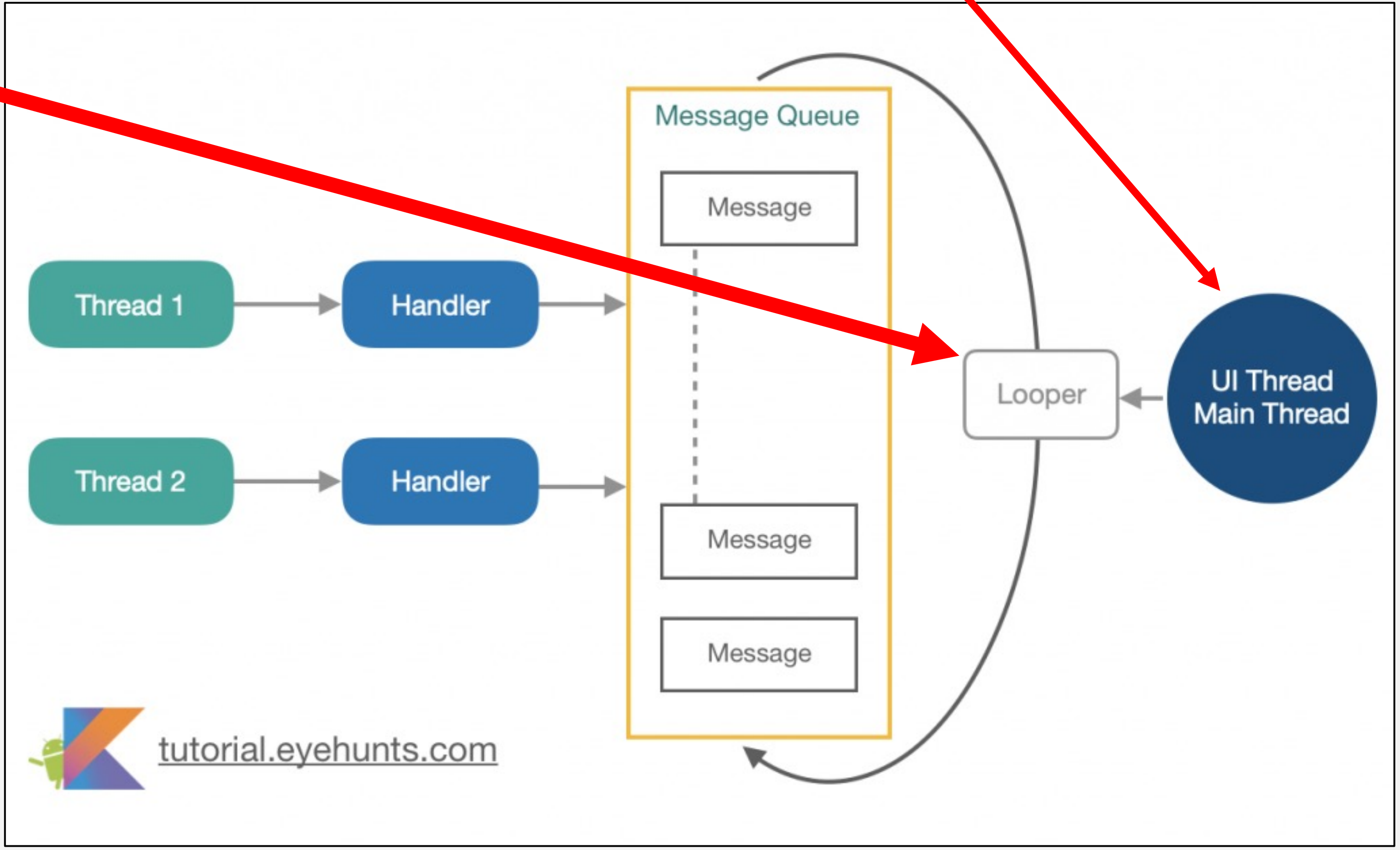
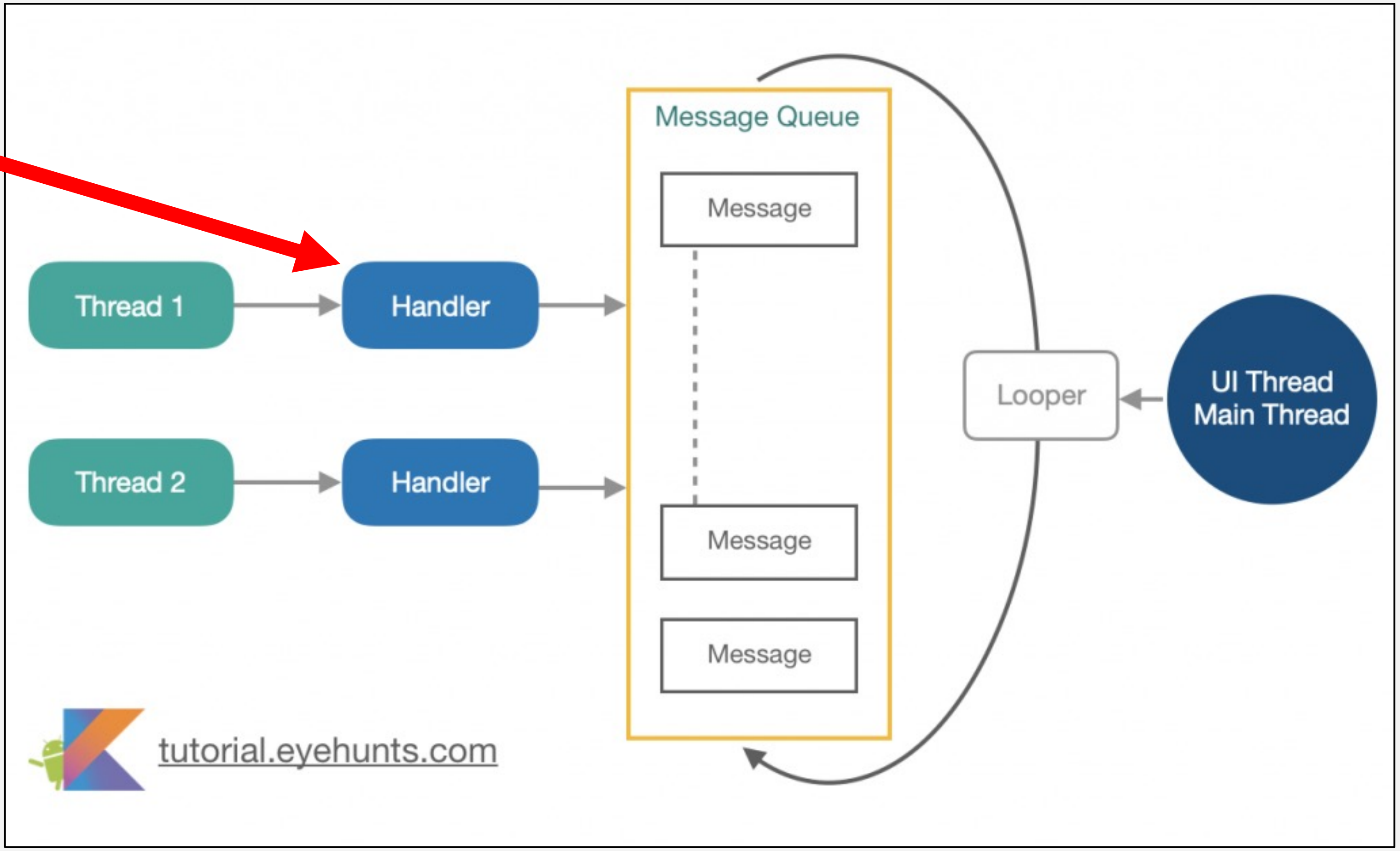


Image source:
https://tutorial.eyehunts.com/android/android-handler-background-thread-communicate-ui-thread/

Handler

- Handlers are associated to a looper
- Act as a bridge for threads to the main thread's looper



If you don't assign a Handler a particular Looper, it is assigned to the looper of the thread where it was created.

Image source:
https://tutorial.eyehunts.com/android/android-handler-background-thread-communicate-ui-thread/

UI Thread Access

- …we can make use of a Handler object tied to the main thread and pass that object to another thread in order to send updates to the UI thread

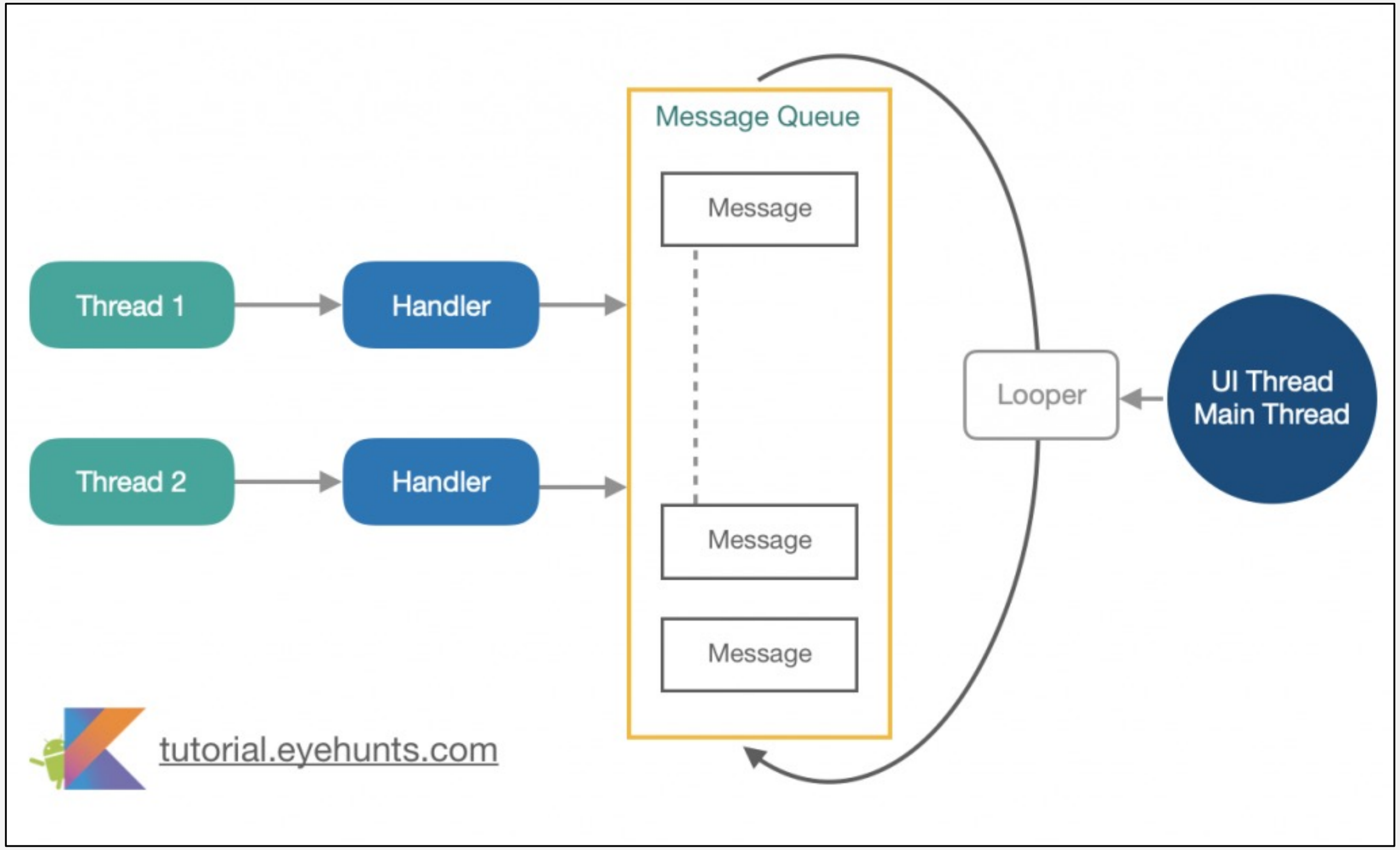


Image source:
https://tutorial.eyehunts.com/android/android-handler-background-thread-communicate-ui-thread/

```java
Handler mHandler = new Handler();
…
new Thread(new Runnable() {
    int start = 0, end = 100;
    @Override
    public void run() {
        for (int i = start ; i <= end; i++) {
            int progress = i;
            try {
                // This is where you'd want to performing the heavy lifting
                // For now, simulation of computing is done through a sleep() method
                Thread.sleep(progress);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            mHandler.post(new Runnable() {
                @Override
                public void run() {
                    progressBar.setProgress(progress);
                }
            });
        }
    }
}).start();
```

Context: The code here is responsible for updating a progress bar after some processing

Other methods for scheduling tasks:

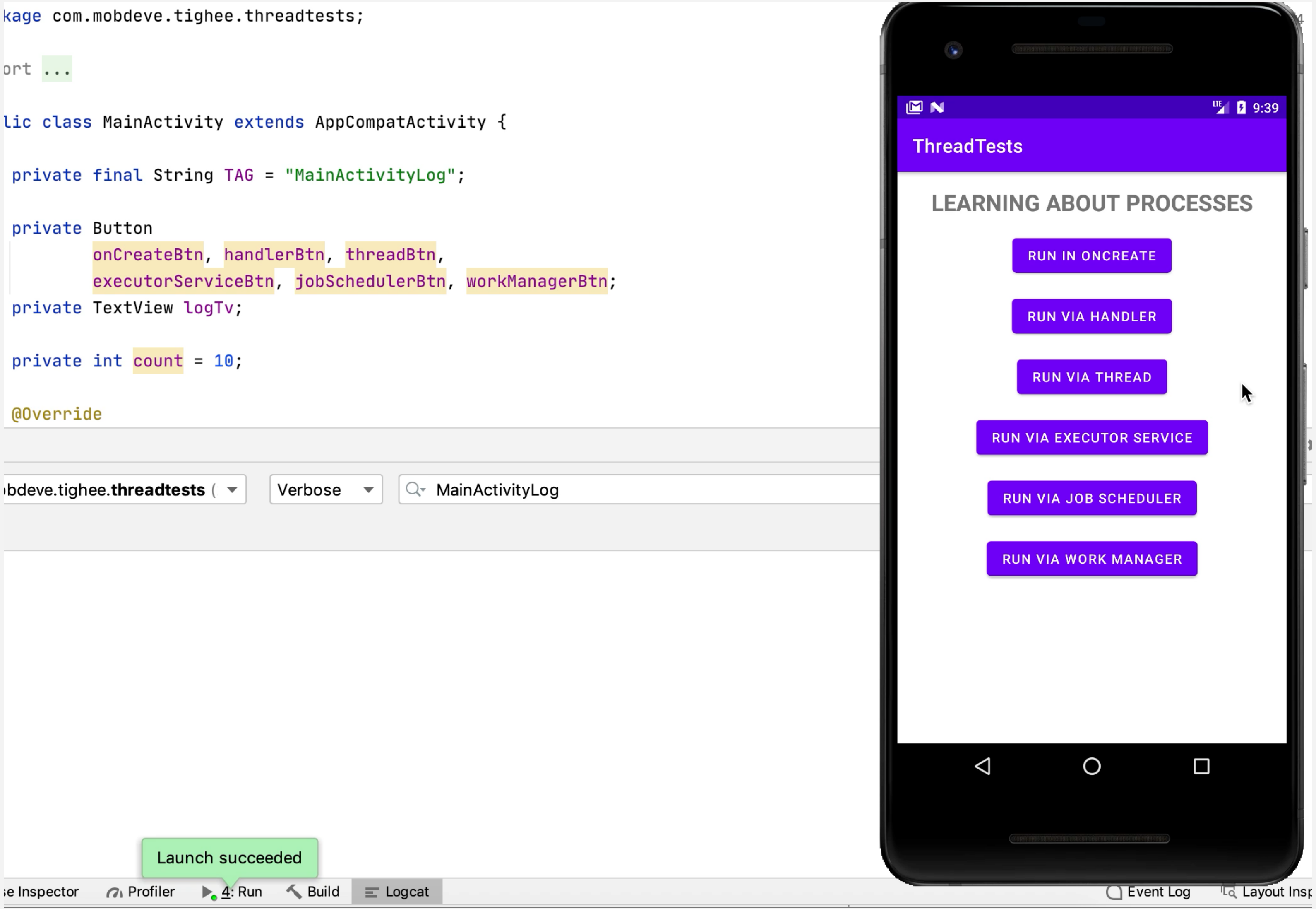
- Runnable object
  - post(Runnable)
  - postAtTime(java.lang.Runnable, long)
  - postDelayed(Runnable, Object, long)
- Message object
  - sendEmptyMessage(int)
  - sendMessage(Message)
  - sendMessageAtTime(Message, long)
  - sendMessageDelayed(Message, long)

UI Thread Access

- View.post() and Handler.post() methods are roughly the same if the Handler object is pointing to the main thread
    - A Handler object is just a bridge, so it can actually be pointing to another Looper that isn't on the main thread

2. Cleaning up thread operations

Cleaning up refers to situations where your thread is no longer needed but could continue executing



```
kage com.mobdeve.tighee.threadtests;

ort ...

lic class MainActivity extends AppCompatActivity {

 private final String TAG = "MainActivityLog";

 private Button
         onCreateBtn, handlerBtn, threadBtn,
         executorServiceBtn, jobSchedulerBtn, workManagerBtn;
 private TextView logTv;

 private int count = 10;

 @Override
```

In the example here, we see a thread continuing to execute after the activity / app closes

2. Cleaning up thread operations

- This point is more of a reminder to be aware of how worker threads execute
  - Threads will continue to execute once a task has been passed
  - However, the app should at least stop any new tasks from being executed
- There’s also no one way of trying to stop the operations and its highly dependent on what you’re trying to implement
  - Can make use of volatile flag variables or Thread.interrupt()

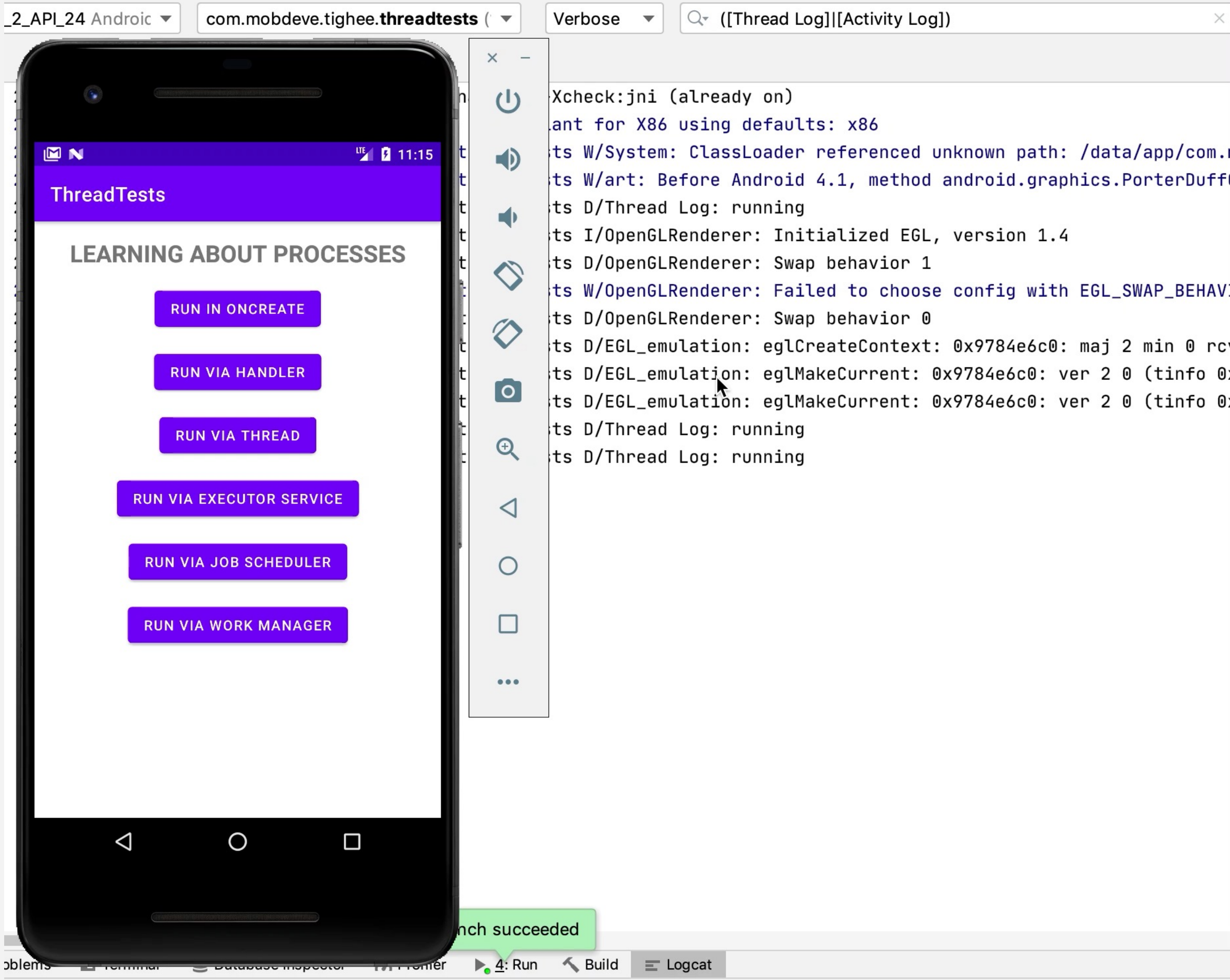
Example

Assumptions:

- Declared as a class variable
- stopThread() called in onStop()

```
class MyThread extends Thread {
  @Override
  public void run() {
    while(!Thread.currentThread().isInterrupted()) {
      Log.d(TAG, "running");
      try {
        // your operation
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
        Log.d(TAG, "thread interrupted");
        return;
      }
    }
  }

  public void stopThread() {
    interrupt();
  }
};
```



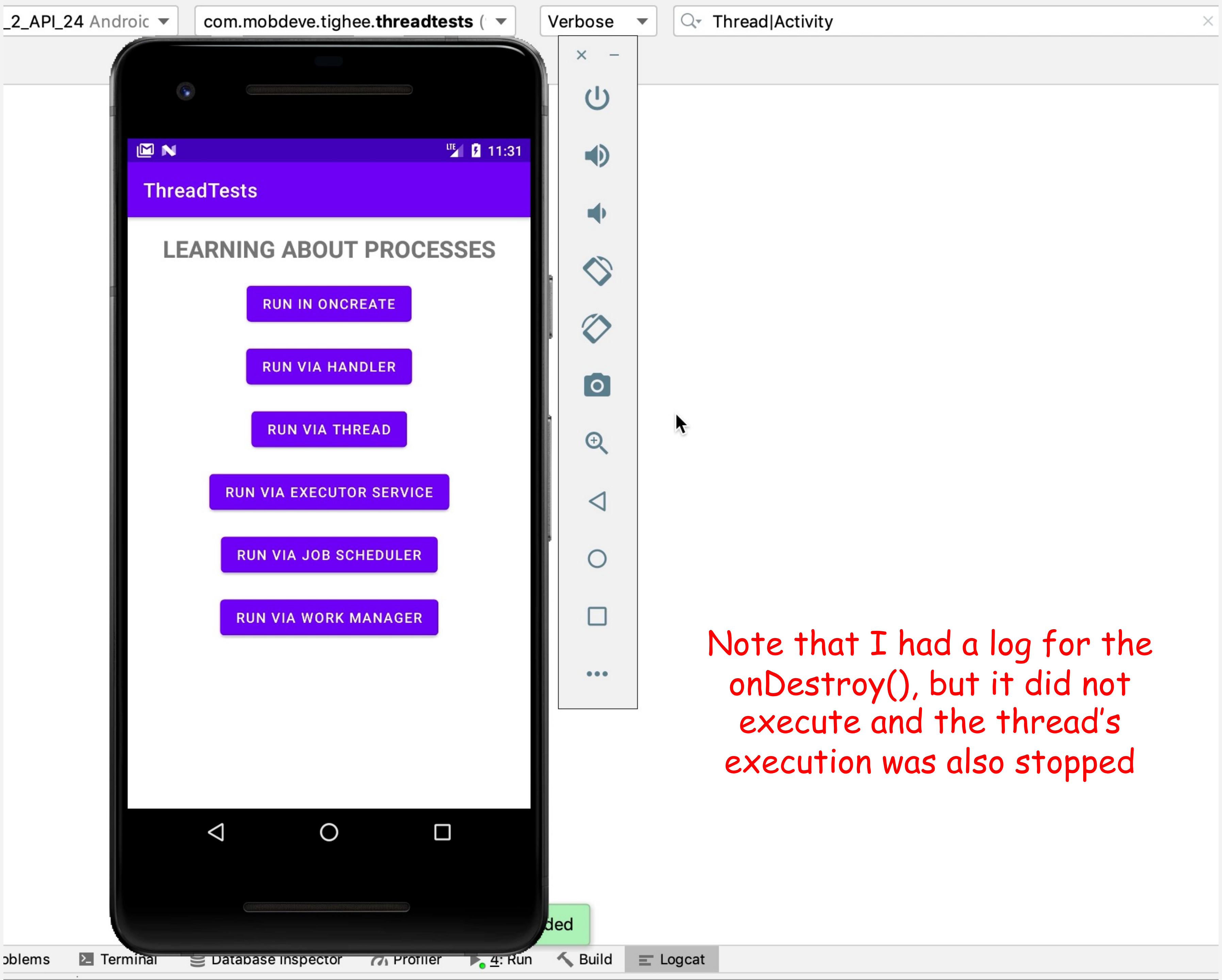
```
Xcheck:jni (already on)
ant for X86 using defaults: x86
ts W/System: ClassLoader referenced unknown path: /data/app/com.
ts W/art: Before Android 4.1, method android.graphics.PorterDuff
ts D/Thread Log: running
ts I/OpenGLRenderer: Initialized EGL, version 1.4
ts D/OpenGLRenderer: Swap behavior 1
ts W/OpenGLRenderer: Failed to choose config with EGL_SWAP_BEHAV
ts D/OpenGLRenderer: Swap behavior 0
ts D/EGL_emulation: eglCreateContext: 0x9784e6c0: maj 2 min 0 rc
ts D/EGL_emulation: eglMakeCurrent: 0x9784e6c0: ver 2 0 (tinfo 0
ts D/EGL_emulation: eglMakeCurrent: 0x9784e6c0: ver 2 0 (tinfo 0
ts D/Thread Log: running
ts D/Thread Log: running
```

2. Cleaning up ~~thread~~ background operations

- We should also have this mindset when dealing with any type of background operation
  - Threads
  - Other components discussed in pt2
    - Services
    - Jobs
    - Tasks
    - Workers
    - Etc.

3. Thread are killed when app is killed

- Despite being able to offload tasks from the main thread, threads are tied to its containing Activity / App
    - If the hosting processing of the app is destroyed, threads are destroyed



Note that I had a log for the onDestroy(), but it did not execute and the thread's execution was also stopped

3. Thread are killed when app is killed

- Threads are alright if you have simple tasks to do in app, but you wouldn’t want to use these for:
  - Operations outside of the app
  - Periodic activities
  - Activities that require specific settings
    - E.g. run only when connected to a network
- Android offers more powerful components…

… which we'll discuss in the 2nd part of the module ☺

Any questions?

I have a question for you all!

```
private void updateDataAndAdapter() {
    MyFirestoreReferences.getPostCollectionReference()
        .orderBy(MyFirestoreReferences.TIMESTAMP_FIELD)
        .get()
        .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
            @Override
            public void onComplete(Task<QuerySnapshot> task) {
                if(task.isSuccessful()) {
                    ArrayList<Post> p = new ArrayList<>();
                    for (QueryDocumentSnapshot document : task.getResult())
                        p.add(document.toObject(Post.class));

                    myAdapter.setData(p);
                    myAdapter.notifyDataSetChanged();
                }
            }
        }
    );
}
```

Pre-question Notes:

- This code snippet is from our discussion on Firebase
- Here we get a Firestore DB reference, perform a query, and add logic to handle the return

Observe:

- In onComplete(), we are modifying the data and informing the adapter to redraw the items in the RecyclerView.

Question:

- *As the UI is being modified, is the logic provided appropriate with respect to the design considerations we discussed for background processes?*

Answer:

- Yes! Firebase's API runs the network request on a separate thread, but onComplete runs on the main thread ☺

Brief Summary

- We discussed…
    - Concurrency with respect to Android applications
    - Main thread and worker threads
    - Communication between worker and main threads
        - runOnUiThread()
        - View.post()
        - Looper + Handler
    - General design considerations

Thanks everyone!

See you next meeting! ☺

programmers then
we send ppl to the moon with 4KB of RAM
programmers now
need 8GB RAM PC for hello world project.

Image sauce: *https://www.reddit.com/r/ProgrammerHumor/comments/htbfbt/open_chrome_and_android_studio_my_pc_go_brrrr/*