KindlingAppThreadsAndCoroutines Notes

This app demonstrates downloading an image and displaying it in the app created in the mobile forum (fundamentals part) without using 3rd part libraries

Show main activity layout

Show image that we are going to display <https://raw.githubusercontent.com/sahilNaran/demos-hosting-images/master/cat_selfie.jpg?token=ABXDFPDPFW3EOFDF4THDL326ZODRA>  
  
**1. Code block 1**  
Android is strict about doing work on the main thread

In newer versions of android, Android will throw a .NetworkOnMainThreadException

Run the app to show the exception

**2. Code block 2.1**

COMMENT OUT THE CODE IN CODE BLOCK 1   
UNCOMMENT OUT CODE BLOCK 2.1

Let’s download the image

We begin by manually creating our own new Thread and using a Runnable.

We start the thread.   
  
Run the app to show it fails after it downloads (few seconds later after the image is downloaded)

The reason this fails is because we are trying to display the image using ‘setImageBitmap’ on main thread

**3. Code block 2.2**

COMMENT OUT THE CODE IN CODE BLOCK 2.1  
UNCOMMENT OUT CODE BLOCK 2.2

There’s two methods to switch back to the main thread,  
Let’s look at method one, using ‘runOnUiThread’  
Seems amazing right? The problem is that we can only use run ’runOnUiThread’ if we have access to the activity  
  
Run the app to show it works

Scroll down

**4. Code block 2.3**

COMMENT OUT THE CODE IN CODE BLOCK 2.2  
UNCOMMENT OUT CODE BLOCK 2.3

Method two: using Loopers and Handlers.  
Loopers are part of the android ecosystem, loopers loop through all the messages or signals any thread receives and processes those messages  
Android has a main looper that we can use and access anywhere   
  
Note we STILL need to spawn our own Thread and start it outselves  
  
Run the app to show it works

**5. Code block 3**

COMMENT OUT THE CODE IN CODE BLOCK 2.3  
UNCOMMENT OUT CODE BLOCK 3

Using a coroutine to run our background processing in another thread  
In this example we use a GlobalScope which is bound to the application lifecycle  
In this example, we are not specifying which dispatcher or thread pool to use, therefore it will run using the default Dispatcher (limited to 4 threads, the default is 2)  
  
It is important to remember the IO dispatcher sometimes shares threads with the default dispatcher, meaning that it will sometimes use threads from the default dispatcher thread pool  
  
  
Run the app to show it works

Go through the log

Don’t forget to filter logs by the ‘MainActivityThread’ tag

As we can see in the logs, we start off in the main thread, then we are in default dispatcher pool and then back in the main thread

**5. Code block 3.1**

COMMENT OUT THE CODE IN CODE BLOCK 3  
UNCOMMENT OUT CODE BLOCK 3.1

In this example we removed the handler and explicitly defined which thread pools we want to use.   
In the launch() method we specified the context of the scope using the IO dispatcher.   
Within in the .launch method we have access to the coroutine scope therefore it’s easy for us to display the our image on the main thread using the Main dispatcher

Run the app to show it works

Show logs

The problem with manual thread management is that it doesn’t scale well is memory intensive