Jake Taggart
Week 3 Day 3 Research

Week 3 Day 3 Research

1. What are loaders and how do we implement loaders?

   Loaders are abstract class implementations of Loader, which is a set of API's provided by Android that have the main goal of helping the developer load data asynchronously in an activity or fragment. First to use it we have to implement either of the two abstract classes that implement Loader, which are the AsyncTaskLoader or CursorLoader. CursorLoader is used to deal with content provider's data and AsyncTaskLoader allows AsyncTask to perform operations in the background. There should only be one instance in each fragment or activity and is created via LoaderManager. LoaderManager allows multiple instances of loaders operate across an application. To use the manager, it should be initialized in either the onCreate() or onStart() functions. The method init(LOADER.ID, OPTION_ARGUMENT, CALLBACK_CLASS_REFERENCE) will call the onCreateLoader() method and ensure that there is not the same ID in existence and if it is present already, it will use the correct callback to ensure the loader isn't recreated from scratch. One can also create their own custom loader.

2. What is an AsyncTaskLoader?

   Touched upon in the previous answer, an AsyncTaskLoader is an abstract class that extends the Loader class. To use this, the developer needs to implement LoaderManager.LoaderCallbacks, then override the onCreateLoader, onLoadFinished, and onLoaderReset. In the onCreateLoader method, we initiate the AsyncTaskLoader to handle tasks in the background. In the onLoadFinished method, we get the result of the load and update the user interface. In the onLoaderReset, we have the option to reconfigure how the loader operates on the next reset, if there is any. Inside the onCreateLoader, a new AsyncTaskLoader object needs to be created with this class being passed. Then we need to override the methods loadInBackground (similar to doInBackground in that any background work should be defined here) and onStartLoading which is similar to AsyncTask's onPreExecute method (progress bars, updates for the user, and using forceLoad at the end). mIn the MainActivity after setting the view, we need to call the getSupportLoaderManager.initLoader method to initialize it.

3. What is a Handler Thread for?

First of all, in Android the application code all runs on the main thread. Anything dealing with the user interface and handling of the user interface actions is carried by the main thread. The handler allows the developer to program so that communication with the UI thread (main thread) and the worker thread (the thread doing background operations so as to not slow down the user experience) is possible. Without a tool like a handler, Android does not allow worker threads to communicate with the main UI thread.

4. What are some common threading restrictions in android?

   One common threading restriction is that without a tool to handle asynchronous tasks such as a handler, the worker thread and the main thread cannot communicate. There must be some technique to correctly deal with asynchronous tasks in order for this to be possible. Another common threading restriction is that creating a significant number of threads can cause CPU performance issues. Because an application uses a shared pool of CPU resources, it can only handle a certain amount of threads asynchronously. Thus, it's wise to only create as many threads as absolutely needed and to be creative when performance is starting to slow. Anything getting above five or so asynchronous threads is starting to push it a bit, but this is variable depending on the system. One way to help thwart this is to try to reuse existing threadpools which reduces memory and processing resources. Threadpooling is a way to group a number of worker threads and be able to use system processing and resources while treating the pool as one task. Often times trying to use an asynchronous process with multiple threads can run into issues such as deadlock or race condition. In order to get around this barrier there are ways to get around this such as using volatile variables, atomic primitives, approaching resources as locks which gives way to patterns such as mutex locks, semaphores, cyclic barriers countdown latches, and more.

5. What are thread pools and thread pool executor?

   As mentioned in the previous question, thread pools are a way to group multiple worker threads while being treated as a single FIFO (first in, first out) task queue by the system processing and resources. The main thread is responsible for sending tasks to the task queue. Once a worker thread in the thread pool is available, it is taken from the queue first and begins to run. The ThreadPoolExecutor allows the developer to specify how many core threads, how many maximum threads, and the time for idle threads to be kept alive. Thread pool executors are great for bounding and managing resources and threads. ThreadPoolExcutor keeps track of basic metrics like the amount of tasks which have already been completed. This allows it to keep track of the task queue much more easily than if it didn't track that number. When creating new threads for a ThreadPoolExecutor,

by default new threads are created using a ThreadFactory which automatically creates all threads to be in the same group and the same priority. We need to supply a custom ThreadFactory in order to alter priority, thread name, and group. When creating a ThreadPoolExecutor, it accepts the parameters of core pool size, maximum pool size, keep alive time, the unit of time being used, BlockingQueue as a runnable (this holds off tasks from being ran until their turn based on priority), and the ThreadFactory.