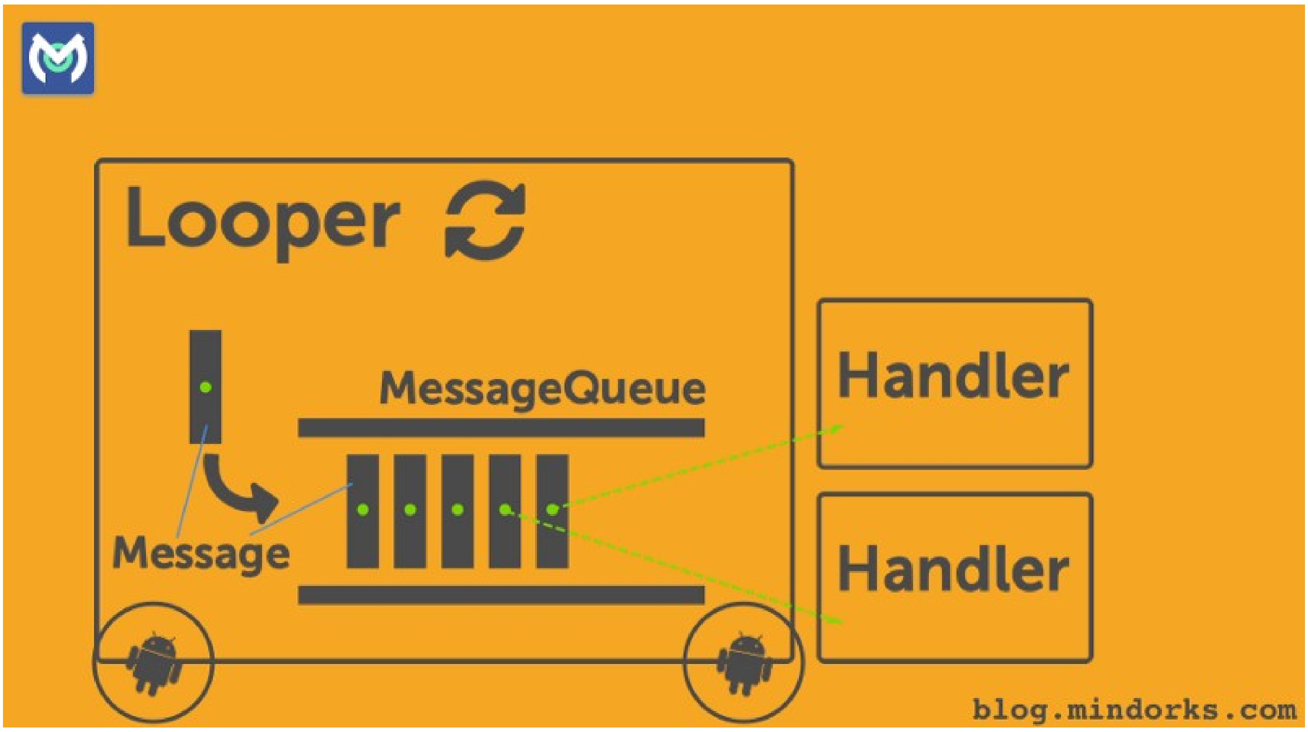
Understanding Android Core: Looper, Handler, and HandlerThread



This Article covers Android Looper, Handler, and HandlerThread. These are among the building blocks of Android OS.

> *In my own experience, I have used them in a very limited context until recently. My use case involved sending tasks to the main/ui thread, primarily to update the UI from any other thread. The other aspects of the multi-threaded operation were handled through alternate ways like ThreadPoolExecutor, IntentService, and AsyncTask.*

MultiThreading and task running are old subjects. Java itself has **java.util.concurrent** package and **Fork/Join** framework to facilitate it. Several libraries have been written to streamline asynchronous operations. **RxJava** is the most popular library today for reactive programming and designing an asynchronous application.

So, why am I writing about the old school?

`Looper`, `Handler`, and `HandlerThread` are the Android's way of solving the problems of asynchronous programming. They are not old school, but a neat structure on which a complex android framework is built.

For new developers, it's highly recommended to understand the principles behind them and experienced one's should revisit this topic to recollect the minor details.

Use Cases:

1. The main thread in Android is built with a `Looper` and `Handlers`. So, the understanding of it is essential to create an unblocked responsive UI.
2. The developers writing libraries cannot afford to use third party libraries because of the library size. So, for them, the best option is to utilize the existing available resource. Writing own solution for it may not always get that level of efficiency and optimization.
3. The same argument can also be made for companies/individuals shipping out SDKs. The clients can have varied implementations, but all of them will share the common android framework APIs.
4. Understanding them fully will enhance the capacity to follow the Android SDK and package classes in general.

Let's start the exploration/revision with a questionnaire.

> *I expect the reader to have the basic understanding of java threads. If you need, then get a quick overview of java Thread and Runnable.*

What is the problem with java thread?

Java threads are one-time use only and die after executing its run method.

Can we improve upon it?

The Thread is a double edged sword. We can speed up the execution by distributing the tasks among threads of execution, but can also slow it down when threads are in excess. Thread creation in itself is an overhead. So, the best option is to have an optimum number of threads and reuse them for tasks execution.

Model for thread reusability:

1. The thread is kept alive, in a loop via it's `run()` method.
2. The task is executed serially by that thread and is maintained in a queue (MessageQueue).
3. The thread must be terminated when done.

What is the Android's way of doing it?

The above model is implemented in the Android via `Looper`, `Handler`, and `HandlerThread`. The System can be visualized to be a vehicle as in the article's cover.

1. `MessageQueue` is a queue that has tasks called messages which should be processed.
2. `Handler` enqueues task in the `MessageQueue` using `Looper` and also executes them when the task comes out of the `MessageQueue`.
3. `Looper` is a worker that keeps a thread alive, loops through `MessageQueue` and sends messages to the corresponding `handler` to process.
4. Finally `Thread` gets terminated by calling Looper's `quit()` method.

> *One thread can have only one unique Looper and can have many unique Handlers associated with it.*

Creating Looper and MessageQueue for a Thread:

A thread gets a `Looper` and `MessageQueue` by calling `Looper.prepare()` after its running. `Looper.prepare()` identifies the calling thread, creates a `Looper` and `MessageQueue` object and associate the `thread` with them in [`ThreadLocal`] storage class. `Looper.loop()` must be called to start the associated looper. Similarly, the `looper` must be terminated explicitly through `looper.quit()`.

```
class LooperThread extends Thread {
      public Handler mHandler;

      public void run() {
          Looper.prepare();

          mHandler = new Handler() {
              public void handleMessage(Message msg) {
                 // process incoming messages here
                 // this will run in non-ui/background thread
              }
          };

          Looper.loop();
      }
  }
```

Creating Handler for a Thread:

A `Handler` gets implicitly associated with the thread that instantiates it via thread's `Looper`, but we can explicitly tie it to a thread by passing the thread's `looper` in the constructor of the `Handler`.

```
handler = new Handler() {
@Override
public void handleMessage(Message msg) {
        // process incoming messages here
        // this will run in the thread, which instantiates it
    }
};
```

Sending messages to the `MessageQueue` via `Handler` can be done by two modes:

1. `Message`: It is a class that defines various useful methods to deal with message data. To send an object we set the **obj** variable.

```
Message msg = new Message();
msg.obj = "Ali send message";
handler.sendMessage(msg);
```

Detailed overview of `Message` *class can be found here:* https://developer.android.com/reference/android/os/Message.html

2. `Runnable`: A runnable can also be posted in the `MessageQueue`. Ex: posting and running a task in the main thread.

```
new Handler(Looper.getMainLooper()).post(new Runnable() {
@Override
public void run() {
        // this will run in the main thread
    }
});
```

In the above example, we create a `Handler` and provide `Looper` associated with the main thread. This associate this handler to the main thread. When we post the `Runnable`, it gets queued in the main thread's `MessageQueue` and then executed in the main thread.

Handler is capable of message manipulation in a wide variety of ways, which can found here: https://developer.android.com/reference/android/os/Handler.html

Creating an own thread and providing `Lopper` and `MessageQueue` is not the right way to deal with the problem. So, Android has provided `HandlerThread`(*subclass of* `Thread`) to streamline the process. Internally it does the same things that we have done but in a robust way. So, always use `HandlerThread`.

One of the ways to create the HandlerThread is to subclass it and most of the time you will be using this method.

```
private class MyHandlerThread extends HandlerThread {

    Handler handler;

    public MyHandlerThread(String name) {
        super(name);
    }

    @Override
    protected void onLooperPrepared() {
        handler = new Handler(getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                // process incoming messages here
                // this will run in non-ui/background thread
            }
        };
    }
}
```

Note: We have instantiated the Handler when the `onLooperPrepared()` is called. So, that `Handler` can be associated with that `Looper`.

1. `Looper` is only prepared after HandlerThread's `start()` is called i.e. after the thread is running.
2. A `Handler` can be associated with a `HandlerThread`, only after it's `Looper` is prepared.

Other way to create the HandlerThread:

```
HandlerThread handlerThread = new HandlerThread("MyHandlerThread");
handlerThread.start();
Handler handler = new Handler(handlerThread.getLooper());
```

Note: HandlerThread needs to call `myHandlerThread.quit()` to free the resources and stop the execution of the thread.

I have also created a video tutorial for this subject, and I highly recommend to watch it. Click here to watch now.