Handlers, Loopers, MessageQueues:

Probably the most important APIs of Android suited for Multithreading and offloading the tasks to worker threads- Handlers and Loopers.

The Android architecture has the Main Thread AKA UI thread, which updates the UI after every 16ms frame. Failure to update within this window will reflect as the “lag”, and even worse if it fails for 5secs, then the “Application not responding” shown to the user- marking the app crash. To avoid these types of issues, please resort to offloading the heavy tasks to worker threads.

To explain: the Android architecture has one main/UI thread which should ideally only be responsible for updating the UI, and other tasks should be done on separate threads(called worker threads). These worker threads upon completion should send the result back to the main thread and the main thread can then use it to update the UI, etc. There are some Android APIs to do so like Async tasks, Services(Intent services actually), Executors, etc, etc. But there are even cooler APIs available and are the core of all the above-mentioned APIs- Handlers and Loopers. Even Async and Intent Services rely heavily on them. Let’s see how they work.

How to start a Thread?

```
//starting a thread

public class MainActivity extends AppCompatActivity {

    private Runnable runnable = new Runnable() {
        @Override
        public void run() {
            // This runs in background thread. Do heavy operations here
        }
            // This thread will die once it comes out of the run method
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

How to keep a thread alive like the main thread in Android ?

To know how to keep a thread alive, we need to first understand what are Loopers in Android.

What are Loopers ?

Class used to run a message loop for a thread. Threads by default do not have a message loop associated with them; to create one, call prepare() in the thread that is to run the loop, and then loop() to have it process messages until the loop is stopped.

Consider you have to perform 10 tasks in the background thread in a single thread, So how would you do perform this operation ? Also remember you cannot start the same thread more than once.

Suppose if you want to you use the thread and re use it any n of times it can be done by associating a message queue with the thread.

How do you associate a thread with a message queue ?

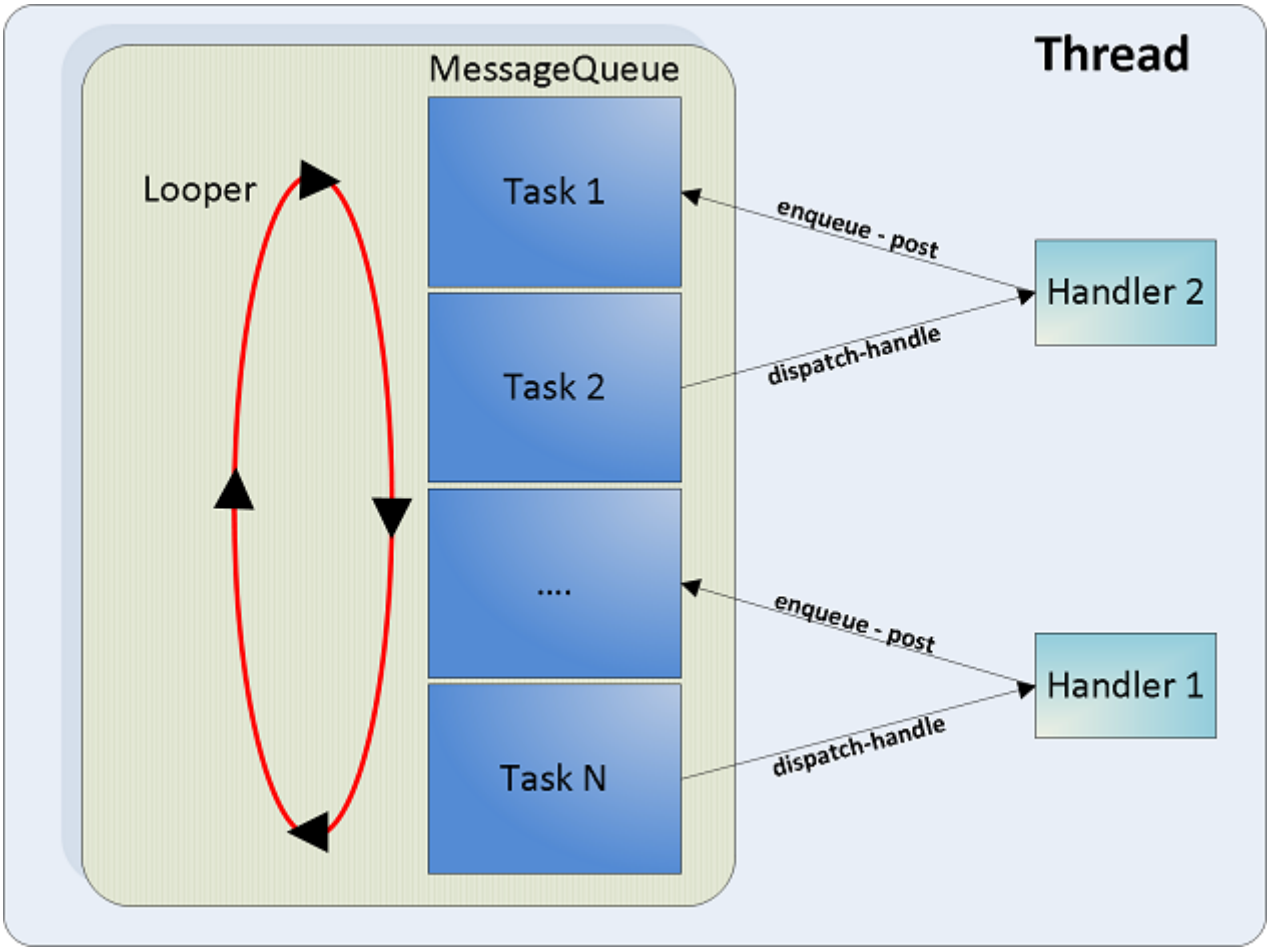
As the official docs says, we need to call Looper.prepare() and to loop through a message queue we need to loop using Looper.loop()

```
//BackgroundThread.java

import android.os.Looper;
import android.util.Log;

public class BackgroundThread extends Thread {

    private static final String TAG = BackgroundThread.class.getSimpleName();

    @Override
    public void run() {
        super.run();
        Looper.prepare();
        for (int i = 0; i < 5; i++) {
            Log.d(TAG, i + "");
        }
        Looper.loop();
        Log.d(TAG, "Thread is killed ?");
    }
}
```

```
//MainActivity.java

import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private BackgroundThread backgroundThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        backgroundThread = new BackgroundThread();
        /*
        This will call the run method in the BackgroundThread.java which will run
in the background thread
         */
        backgroundThread.start();
    }
}

So when you run the above code, we get the below output

0
1
2
3
4
```

But Thread is killed ? is never called (Log inside run()) that means the Thread is still running.



Why we need Handlers and loopers?

So Imagine that you run into a problem where you want your thread to be running and you being able to post/send task to it so that it executes them basically what I mean is you want to reuse your thread. The way to do this in Java is

Keep a thread alive i.e. do not let it come out of run method. Maintain a queue and post your task in it. Process the queue in your run method ie execute task one by one or when it comes Finish/Terminate the thread once done

Handler

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue .

Now in the above example we have associated thread with a MessageQueue , **Now what if we want to perform multiple tasks in the same thread ? How would you pass the task to the same threads messagequeue ? Well we can use handler for this.**

```
// MainActivity2.java
import android.os.Bundle;
import android.os.SystemClock;

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity2 extends AppCompatActivity {

    private BackgroundThread backgroundThread;

    /**
     * Task 1 will be started in the thread
     */
    private Runnable task1 = () -> {
        // this is called in the thread
        for (int i = 0; i < 10; i++) {
            SystemClock.sleep(1000); // some delay
            // Assume this is a heavy task which is performed
        }
    };

    /**
     * Task 2 will be started in the same thread after task1 is finished executing
     */
    private Runnable task2 = () -> {
        // this is called in the thread
        for (int i = 0; i < 100; i++) {
            // Assume this is a heavy task which is performed
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
        backgroundThread = new BackgroundThread();
        backgroundThread.start(); // Start a new thread
        backgroundThread.addTaskToMessageQueue(task1); // Add task1 to the message
queue of the thread
```

```
            backgroundThread.addTaskToMessageQueue(task2); // Add task2 to the message
queue of the same thread
        }
}

**********----------------------------------------**********
//BackgroundThread.java
import android.os.Handler;
import android.os.Looper;

public class BackgroundThread extends Thread {

    private Handler handler;

    @Override
    public void run() {
        super.run();
        Looper.prepare(); // Associating thread
        handler = new Handler();
        Looper.loop();
    }

    /**
     * This method is used to add different tasks to the message queue
     *
     * @param task
     */
    public void addTaskToMessageQueue(Runnable task) {
        handler.post(task); // First task1 is added to the queue then task2.
    }
}
```

In the above example we are adding multiple task (task1 and task2) to the threads message queue. As Queue is FIFO (first in first out), task1 will be added first and next task2 will be added. Ultimately, we are performing both the tasks in the same thread.

I guess you see the issue that you have to manage lots of things and if not lot then at-least you should be very careful with this approach. But the good thing about Handler thread is it you can avoid all these. It comes with a message queue associated which can be used to send any message/task to this thread once it's started. So how does Handler Thread does it?

Android has 3 main components to handle these which is used by HandlerThread. Let's see them once.

Looper:

Looper is a worker that keep a thread alive, It loops over message queue and send the message to respective Handler.

Handler:

This class is responsible for enqueuing any task to message queue and processing them. Each Handler can be associated with one single thread and that thread's message queue.

Message Queue:

This class holds the list of messages to be dispatched by the looper. You can just call Looper.myqueue() to get list of messages. We do not normally deal with it.

Of-course you can use above three components with normal thread class also. You can create your own thread and use all above mentioned components. The process to do this is mentioned below

```
1.Create a Thread class
2.Call Looper.prepare inside run method.
3.instantiate your Handler class and implement handleMessage method.
Call Looper.loop()

I can write the code to do this but since I do not want you to follow that
approach I will skip that. As this is not very efficient way, so Android has made
job simpler by introducing HandlerThread.
```

We can write our own class which extends HandlerThread.. Each HandlerThread class has a looper associated with. We can use this looper to create Handler and then we can start enqueuing/sending message to this thread.

Steps to do it.

Create a class that extend HandlerThread Instantiate your handlerThread class call start method // your thread is running now Create/Instantiate your Handler by using looper from handler thread created above

```
public abstract class BaseHandlerThread extends HandlerThread
{
public BaseHandlerThread(String name)
{
    super(name);
}
@Override
protected void onLooperPrepared()
{
    initHandler();

}
private void initHandler()
{
    mHandler = new Handler(getLooper()) {
        @Override
        public void handleMessage(Message msg)
        {
           //
        }
```

```
    };
}
}
```

We can simply create instance of BaseHandlerThread and then call start() method on it. Once start is called Looper gets prepared. After Looper is prepared we get a callback where we are initialising the Handler. A handler can only be initialised after looper os prepared. Now you can use this handler to enqueue message. When you want to terminate this thread just call quitSafely() method on your handler thread instance or quit() for OS version older then JELLY_BEAN_MR2.

When you do not want to create separate class and still want to use HandlerThread you can follow below step. Note that When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it. So be very sure about how you are creating the Handler.

```
//Create Handler
mHandlerThread = new HandlerThread("name");
mHandlerThread.start();
mHandler = new Handler(mHandlerThread.getLooper());
```

Message:

The Messages which can be sent to the HandlerThread. To avoid the number of Messages initializes, there is a pool maintained. Every time the looper is stopped, or the message is read by Looper, it is recycled(i.e. its fields are cleared off), and added to the pool. Next time we want to create a new Message, we get it from the pool(Message.obtain() method does that). This way the Android prevents us to initialize new Message instances every time.

The message is the LinkedList:

1. Message next;
2. long when;

Line 1: Its next points to the next Message. Line 2: It has long when signifying the timeInMillis of when should message be processed/read.

Please note that the LinkedList of these messages should be processed by their when, so it's sorted on when. When adding a new msg, we iterate the LinkedList to find the particular slot to insert this message and then insert it there.

Sending Data to UI thread(which is the Handler Thread too):

Say a worker thread(created at line 1) wants to send data(123) to UI thread:

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                Looper mainThreadLooper = Looper.getMainLooper(); // --> Looper of
```

```
the main/UI thread
                Handler mainThreadHandler = new Handler(mainThreadLooper); // -->
Get handler to main thread
                Message messageToSendToMainThread = Message.obtain(); // -->
Create a message to send to UI thread
                messageToSendToMainThread.obj = 123; // 123 -> actual msg value
                mainThreadHandler.sendMessage(messageToSendToMainThread);
            }
        }).start();
```

Line 1: Creates a new worker thread. Line 4: Get the Looper associated with the UI thread Line 5:Create the Callback which will be called by UI thread, once it reads the message we are going to send to it. Line 12: Create Handler with the Main Thread's Looper, so that we write the message to Main thread's MQ. Line 13: Create an empty Message from the pool of messages(to avoid GC overload). Line 14: Put some value into the Empty message — 123. Line 15: Setting the above handler(which has callback defined) as the Message Target, which will be called once the message is read by the UI looper. Line 16: Write the Message to the MQ via Main Looper via Handler.

This is pretty easy to write to Main thread from worker thread, as the Main thread is itself a handler thread which has its Looper/MQ etc defined. But what if we want to write our own Handler Thread to which other threads can communicate? In that case we need to initialize Loopers, MQ, etc ourselves.

So make a Thread a Handler thread, we need to do:

1. Looper.prepare() → basically initializes the Looper and creates an MQ for it.

```
public static void prepare() {
        new Looper();
    }

    private Looper() {
        mQueue = new MessageQueue();
    }
```

2. Looper.loop() → This will start the looper, which will keep on reading the Message Queue until Stopped.

```
public static void loop() {
        final Looper me = myLooper();
        if (me == null) {
            throw new RuntimeException("No Looper; Looper.prepare() wasn't called
on this thread.");
        }
        final MessageQueue queue = me.mQueue;

        for (; ; ) {
            Message msg = queue.next(); // might block
            if (msg == null) {
                // No message indicates that the message queue is quitting.
```

```
                return;
            }

            msg.target.dispatchMessage(msg);

            msg.recycleUnchecked();
        }
    }
```

Line 2: This will get the MQ for that Looper. Line 8: Read from MQ infinitely until stopped(in that case msg read from Queue will be null). Line 9: gets blocked until the message has arrived in the MQ. queue.next will be blocked until it sends a msg, it will send null only if Looper is stopped(using Looper.quit() method). Line 15: Once msg is read, send it to its callback(defined in msg.target and hence in Handler's callback). Line 17: Once msg is handled, recycle it to clear its data off, and add it to the pool.

These above 2 methods are sufficient to create a Handler Thread.

Another easy way is to extend the Thread by HandlerThread, which under the hood calls these 2 above methods.

```
//HandlerThread.java
Override
    public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        synchronized (this) {
            mLooper = Looper.myLooper();
            notifyAll();
        }
        Process.setThreadPriority(mPriority);
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
    }
```

This is the HandlerThread run method: which calls Looper.prepare — then Looper.loop and in between onLooperPrepared to do something in this thread.

How to send data to such a Handler thread? Simple, instead of passing the MaineLooper to the Handler, pass the Worker Handler Thread's Looper like:

```
     final HandlerThread handlerThread = new HandlerThread("MyHandlerThread");

        new Thread(new Runnable() {
            @Override
            public void run() {
                Looper workerThreadLooper = handlerThread.getLooper(); // -->
Looper of the Worker Handler thread
                Handler.Callback UICallback = new Handler.Callback() {
```

```
                    @Override
                    public boolean handleMessage(final Message msg) {
                        System.out.println(msg.obj);
                        return true;
                    }
                };
                Handler mainThreadHandler = new Handler(workerThreadLooper,
UICallback); // --> Get handler to Worker Handler thread
                Message messageToSendToMainThread = Message.obtain(); // -->
Create a message to send to Worker Handler thread
                messageToSendToMainThread.obj = 123; // 123 -> actual msg value
                messageToSendToMainThread.setTarget(mainThreadHandler);
                mainThreadHandler.sendMessage(messageToSendToMainThread);
            }
        }).start();
```

Line 6 is the only difference, here the looper belongs to the Handler thread we created and not that of the default UI thread.

Let's explore the working of other Methods too:

MessageQueue::next Looper.loop depends heavily on Queue.next() method. Below is the stripped-down version of next:

MessageQueue has the LinkedList of Messages(sorted on Message.when). Message.when corresponds to the time this message should be processed.

```
//MessageQueue.java
Message next() {
        while (true) {
            synchronized (this) {
                // Try to retrieve the next message.  Return if found.
                final long now = System.currentTimeMillis();
                final Message msg = mMessages;
                if (msg != null) {
                    if (now >= msg.when()) {
                        // Got a message.
                        mMessages = msg.next;
                        msg.next = null;
                        msg.markInUse();
                        return msg;
                    }
                }

                if (mStopping) {
                    return null;
                }
            }
        }
    }
```

Line 6: Get the LinkedList of messages written into the Queue. Line 7 and 8: If the message exists and message read has when before the current time(I.e. it is eligible to be processed), read it. Line 10 and 11: LinkedList points to the next message now, as this message is read and could be removed from the Linked list(its next is set to null). line 12: set this message in use, so that no further action can be done on it, like updating it, etc. line 13: return it to the Looper. This will not happen until any message is read, as there is an infinite loop enclosing this code at line 2. Line 17 and 18: Returns null only if it is stopped.

Adding Messages to Queue: This is done via handler.sendMessage() or Handler.post()

2 types of messages can be sent to HandlerThread: a. Messages: which have when, data, target, etc fields. Upon being read by the Looper, it delegates it to msg.target to handle the message. This will run on the HandlerThread, and not on the worker thread which sends this message. b. Runnable: Upon read by the Looper, the Runnable.run() is called inside the Handler Thread. Under the hood this Runnable is converted into the empty message, and this runnable is set as its callback. So that this runnable also acts as a Message.

```
//MessagerThread.java
public final boolean post(final Runnable runnable) {
        return sendMessageDelayed(Message.obtain().withCallback(runnable), 0);
    }
```

Line 2 wraps this Runnable into the Message, and its the Message which gets written into the MQ.

Handler.sendMessage(Message)

```
    public final boolean sendMessage(final Message msg) {
        return sendMessageDelayed(msg, 0);
    }

    public final boolean sendMessageDelayed(final Message msg, long delayMillis) {
        if (delayMillis < 0) {
            delayMillis = 0;
        }
        return sendMessageAtTime(msg, System.currentTimeMillis() + delayMillis);
    }

    public boolean sendMessageAtTime(final Message msg, final long uptimeMillis) {
        if (mQueue == null) {
            return false;
        }
        return enqueueMessage(mQueue, msg, uptimeMillis);
    }

    private boolean enqueueMessage(final MessageQueue queue, final Message msg,
final long uptimeMillis) {
        return queue.enqueueMessage(msg, uptimeMillis);
    }
```

As shown, this basically calls queue.enqueueMessage to write this message into the mQueue(MessageQueue for this handler thread).

MessageQueue.enqueueMessage(Message)

```
boolean enqueueMessage(final Message msg, final long when) {
        if (msg.isInUse()) {
            throw new IllegalStateException(msg + " This message is already in
use.");
        }

        synchronized (this) {
            if (mStopping) {
                // Do not add further msgs after the close is executed.
                msg.recycle();
                return false;
            }

            msg.markInUse();
            msg.setWhen(when);
            Message p = mMessages;
            if (p == null || when == 0 || when < p.when()) {
                msg.next = p; // as it has when before the first msg, add it at
the head
                mMessages = msg; // and reset head to this.
            } else {
                // Find the appropriate slot to place this message, based on its
when.
                Message prev;
                do {
                    prev = p;
                    p = p.next;
                } while (p != null && when >= p.when());
                msg.next = p;
                prev.next = msg;
            }
            notifyAll();
        }
        return true;
    }
```

Line 2: If it's already being read(next() sets this boolean true), or already added into Queue(line 13), then don't add the same message again. Line 7–10: If the Queue/Looper is quitting, then don't add msgs into it. Line 13: Set this msg to be in use so that it's not added into the queue again. Line 16: If this msg is to be processed before or at the current time, add it at the front. Line 17–18: LinkedList code to add msg at the head. Line 21–27: Find the slot to add this msg based on its when, as the LL is sorted on its when. This is a general LinkedList iteration code.

Please note that all the above methods are stripped-down to ease the understanding.

So the basic design follows:

1. Handler thread can receive messages from other threads.
2. Handler thread has a looper that reads queue for any new messages.
3. Handler Thread has a Message Queue where the messages are written by other threads via a handler.
4. Looper loops until it reads the message, and then it calls the callback function set by the worker thread, within the Handler Thread.
5. And thread can write to this HandlerThread if it has its looper. He can do so by creating a handler(with its looper as param), and then sending messages via this handler.
6. Looper.getMainLooper() returns the looper for mainThread, which can be passed into the handler to write to the MainThread directly from any other thread.