Executor, thread, runnable:

Threads, executors and runnables have the ability to work in parallel. Oldes method is to make subclass of java.lang.thread class.

Example, from: https://www.geeksforgeeks.org/multithreading-in-java/

// Java code for thread creation by implementing

// the Runnable Interface

class MultithreadingDemo implements Runnable

{

public void run()

{

try

{

// Displaying the thread that is running

System.out.println ("Thread " +

Thread.currentThread().getId() +

" is running");

}

catch (Exception e)

{

// Throwing an exception

System.out.println ("Exception is caught");

}

}

}

// Main Class

class Multithread

{

public static void main(String[] args)

{

int n = 8; // Number of threads

for (int i=0; i<8; i++)

{

Thread object = new Thread(new MultithreadingDemo());

object.start();

}

}

}

java.lang.Tread

Java does not support multiple inheritances, this is a limitation of the threading option, but threading allows for yield(), interrupt () and others that are not available in the runnable interface.

You need to call it using start() and reference the run().

This shows an example and it’s not bad: <https://www.youtube.com/watch?v=Xj1uYKa8rIw>

java.lang.Runnable

Runnable tends to be more modular in design; making it easier to create and maintain.

Runnables do not use start()

Here in an example of a runnable interface doing multi threading, an extension of the video above. <https://www.youtube.com/watch?v=xvXbvrUUGMM>

Interfaces are intended to be simple. Call the runnable interface, and use the command public void run(); that’s it. Keep it simple.

Since runnables are computed so close to each other, you can not predict the order of output. (will very from jvm to jvm).

Starting and closing threads can add a lot of CPU time, executables are made to manage that better.

Remember that background tasks can’t update UI (android), so you will need to have the UI pull the results from the treads or use AsyncTask. AsyncTask acts as a parent task.