

Android Process and Threads

Today's class:

Concepts
(Lecture)

- What is a process? Thread?
- In Android?

Problem
(code)

- What is the limitation of the main thread?
- Creating the problem scenario.

Back-to-Concepts
(lecture)

- Running on UI thread.
- Async Task

Solution
(code: all)

- Create the problem scenario
- Use Async Task

Concepts

Program vs. Process

A **program** is a set of instructions + data stored in an executable image.

A **process** is a program “in action”

- Program counter
- CPU registers
- Stacks
- States

For more details:

<http://www.tldp.org/LDP/tlk/kernel/processes.html>

Program vs. Process

Tea Making Recipe:

1. Add water in pan.
2. Add sugar, tea leaves, spices.
3. Bring to boil and simmer.
4. Add milk.
5. Bring to boil and simmer.
6. Strain tea in teapot.

Program or Process?

Program vs. Process

Process:

- 1. Add water in pan.
- 2. Add sugar, tea leaves, spices.
- 3. Bring to boil and simmer.
- 4. Add milk.
- 5. Bring to boil and simmer.
- 6. Strain tea in teapot.



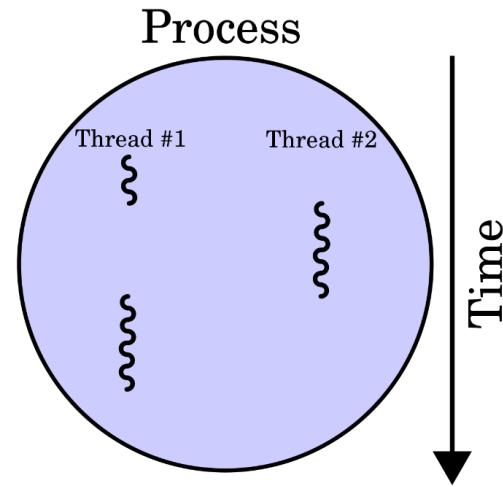
Process:

- 1. Add water in pan.
- 2. Add sugar, tea leaves, spices.
- 3. Bring to boil and simmer.
- 4. Add milk.
- 5. Bring to boil and simmer.
- 6. Strain tea in teapot.



Thread

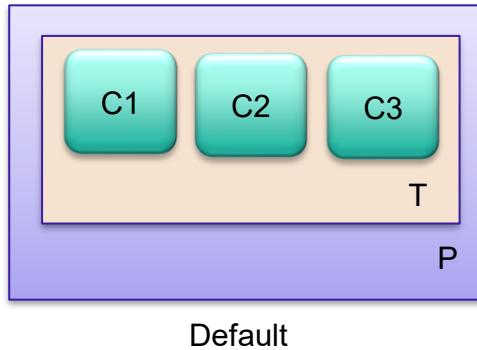
- A **Thread** is a flow of execution inside a process.
- Threads in a process share the same **virtual memory** address space.



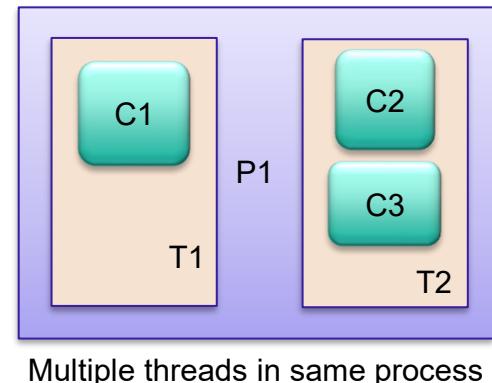
Android Processes and Threads

Default Behavior:

- All components of an App run in the same thread (the main/UI thread) and the same process.



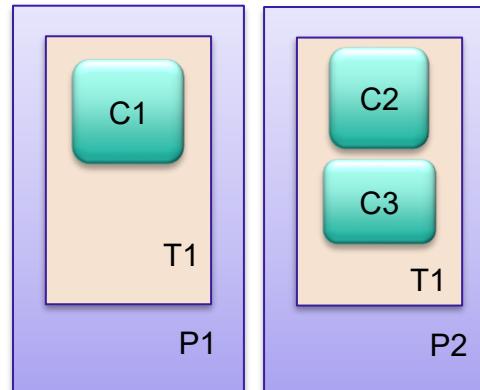
Default



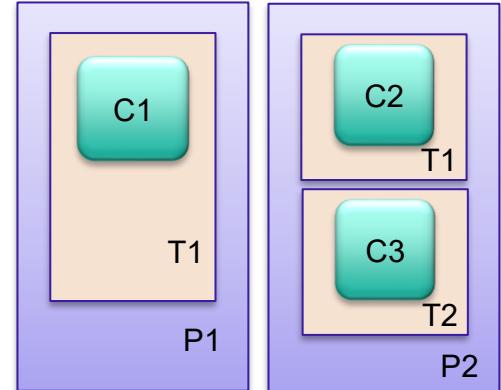
Multiple threads in same process

However, you can:

- Arrange for components to run in different processes.
- Create multiple threads per process.



Multiple processes with single thread in each



Multiple processes with multi threads

Process creation and removal

- **Creation:** When the first component of an App is run and there is currently no process for that App.
- **Removal:** Remove old processes to reclaim memory for new or more important processes.

Which process
should be killed?



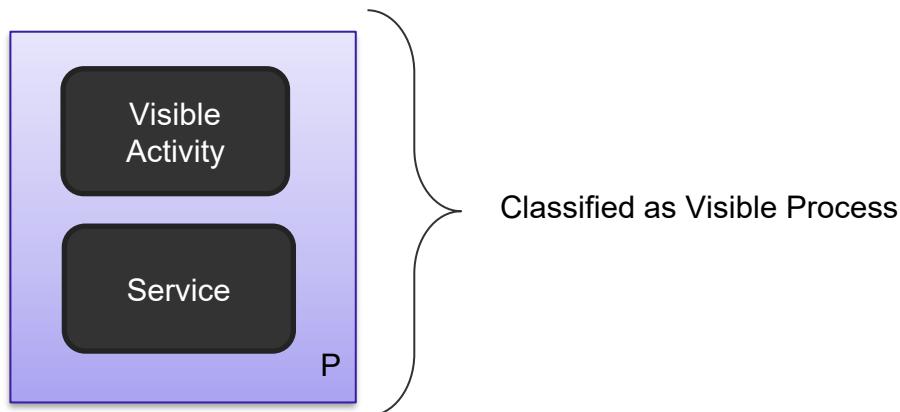
Process Hierarchy

- From High to Low Priority

Priority	Type	Description
1	Foreground	On-going interactions; activity, service, broadcast
2	Visible	Not in the foreground, but visible
3	Service	Playing music, downloading stuffs
4	Background	onStop() called, not visible, LRU kill
5	Empty	Lowest priority



Kill them first



Base Thread

- Act much like usual Java Threads
- Can't act directly on external User Interface objects
 - Throws the Exception CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views”
- Can't be stopped by executing destroy() nor stop()
 - Uses instead interrupt() or join() (by case)
- Two main ways of having a Thread execute application code:
 - Providing **a new class that extends Thread and overriding its run() method**
 - Providing **a new Thread instance with a Runnable object** during its creation
 - In both cases, the start() method must be called to actually execute the new Thread.

Base Thread: example1

overriding run()

```
protected void startDownloadThread() {  
    Thread t = new Thread() {  
        public void run() {  
            mResult = "This is new result";  
        }  
    };  
    t.start();  
}
```

Base Thread: example2

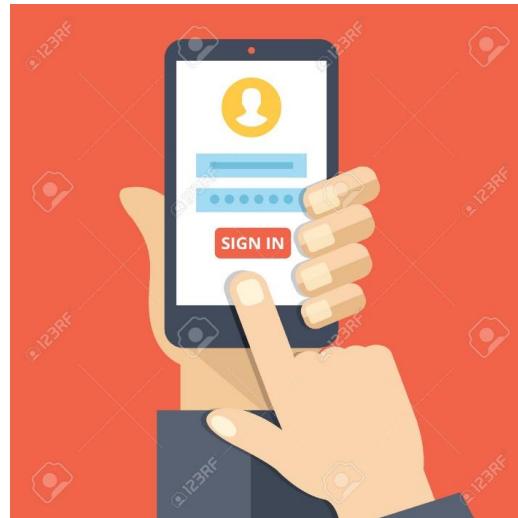
Thread instance with a Runnable

```
class Multi3 implements Runnable{
    public void run(){
        System.out.println("thread is running...");
    }

    public static void main(String args[]){
        Multi3 m1 = new Multi3();
        // Using the constructor Thread(Runnable r)
        Thread t1 = new Thread(m1);
        t1.start();
    }
}
```

Main Thread

- Created when an App is launched
- Often called the **UI thread**



Problem

Keeping Apps Responsive

Problem: keeping an App responsive

- Testing the limitation of the UI thread:

- How big a task we **should** do in UI thread?
- How big a task we **can** do in UI thread?
- What is the **alternative**?
- What is the **limitation of this alternative**?



```
void clickEventHandler(View button)
{
    //what is my limit?
    //Let's experiment!
}
```

Worker Thread

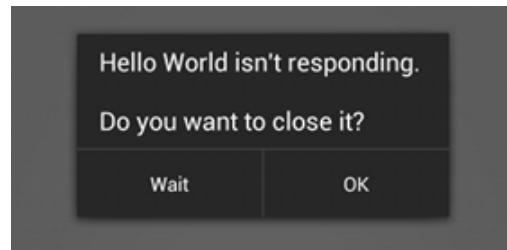
- Used to make UI thread light-weight, responsive.
- Limitation:** Cannot access UI toolkit elements (e.g. views declared in UI thread) from another thread.

```
Runs in UI Thread                                Runs in the new worker Thread  
public void onClick(View v) {  
  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/i.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

Do not try this (i.e. accessing an ImageView which was created in UI thread and now being accessed in a worker thread)

So ... two lessons to remember

- Do not block the UI thread



Caution: Don't run long running (5 sec+) task in the UI thread

- Do not access the Android UI toolkit from outside the UI thread.

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/i.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

Solutions

Solution 1. Handler: Consider UI Thread

- ❖ A main thread **Handler** object
 - Schedules messages and runnable to be executed at some point in the future

Handler Object: Post example

```
public class MainActivity extends AppCompatActivity {
    Handler mHandler;

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

        mHandler = new Handler();

        new Thread(new Runnable() {
            @Override
            public void run() {
                // perform long running task here
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        mProgressBar.setProgress(currentProgressCount);
                    }
                });
            }
        }).start();
    }
}
```

Handler Object: Post Delayed example

```
public class MainActivity extends AppCompatActivity {
    Handler mHandler;

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

        mHandler = new Handler();

        new Thread(new Runnable() {
            @Override
            public void run() {
                // perform long running task here
                mHandler.postDelayed(new Runnable() {
                    @Override
                    public void run() {
                        mProgressBar.setProgress(currentProgressCount);
                    }
                }, 1000);
            }
        }).start();
    }
}
```

Solution 2. Handler: Consider UI Thread

- ❖ posting ***Runnable*** objects to the main view
 - To add an action into a queue performed on a different thread

Posting *Runnable* objects

Access UI Thread from Other Threads

- Use one of these three:
 - **Activity.runOnUiThread (Runnable)**
 - **View.post (Runnable)**
 - **View.postDelayed (Runnable, long)**

The diagram illustrates the execution flow of the provided Java code. It features two main regions: a grey background representing the 'Runs in UI Thread' and a blue background representing the 'Runs in worker Thread'. A dashed purple arrow points from the UI Thread region to the 'mImageView.post(...)' call, indicating that this code runs on the UI thread. A solid purple arrow points from the worker Thread region to the 'mImageView.setImageBitmap(bitmap);' call, indicating that this code runs on the worker thread.

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap =
                loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

Solution 3. Async Task

- Performs the blocking operations in a worker thread, and publishes the results on the UI thread using different built-in methods.

```
public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a worker thread and
     * delivers it the parameters given to AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    /** The system calls this to perform work in the UI thread and delivers
     * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

Runs in worker Thread

Runs in UI Thread

AsyncTask

- Created on the UI thread (any worker thread can be created from UI thread)
- Can be executed only once
- Long running tasks run on a background thread and result is published on the UI thread
- Extended as `AsyncTask<Void, Void, Void>`
 - The three types used by an asynchronous task are the following
 - Params, the type of the parameters sent to the task upon execution
 - Progress, the type of the progress units published during the background computation
 - Result, the type of the result of the background computation

AsyncTask

- Go through 4 steps:
 - **onPreExecute()**: invoked on the UI thread immediately after the execution of the task is started
 - **doInBackground(Param ...)**: invoked on the background thread immediately after onPreExecute() finishes executing
 - **onProgressUpdate(Progress...)**: invoked on the UI thread after a call to publishProgress(Progress...)
 - **onPostExecute(Result)**: invoked on the UI thread after the background computation finishes

AsyncTask example

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

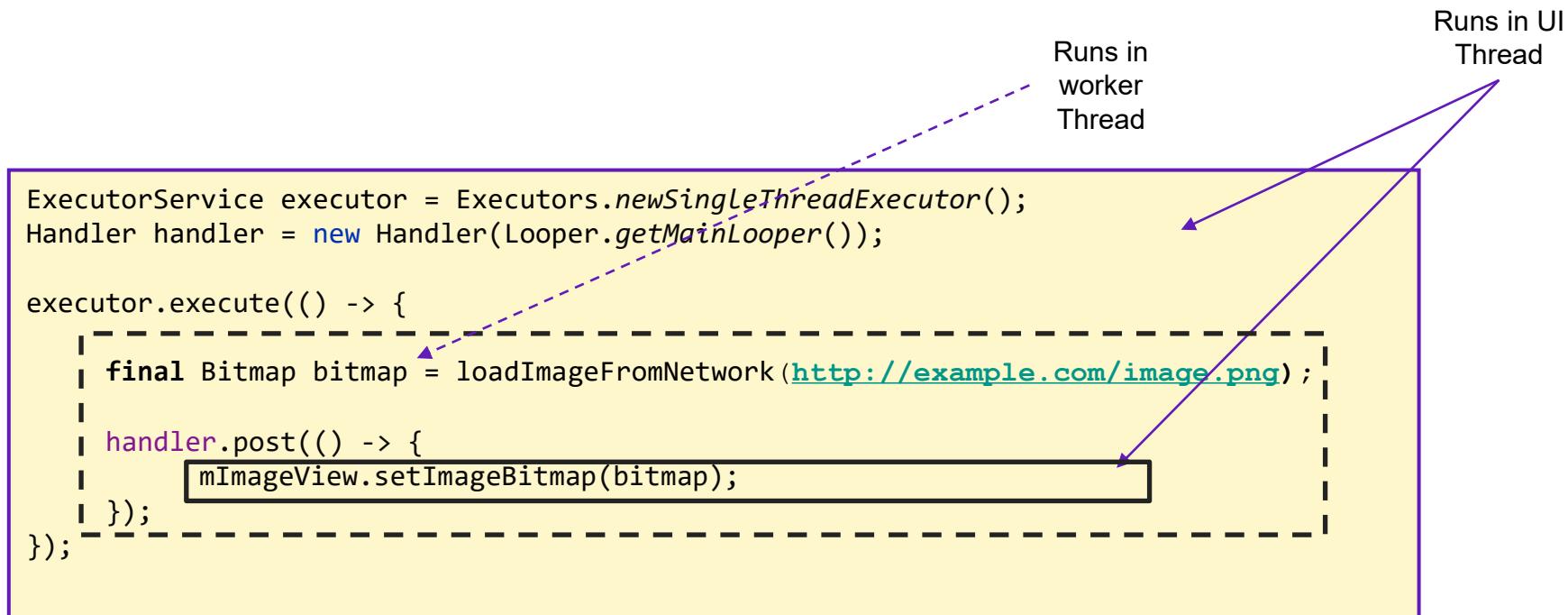
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}

new DownloadFilesTask().execute(url1, url2, url3);
```

Solution 4. Executor and MainLooper (Preferrable)

- Performs the blocking/long running operations in a worker thread (executor), and publishes the results on the UI thread (MainLooper).



```
private void downloadFilesTask() {
    ExecutorService executor = Executors.newSingleThreadExecutor();
    Handler handler = new Handler(Looper.getMainLooper());
    isDownloading = true;
    executor.execute(() -> {
        URL url = new URL(fileUrl);
        urlConnection = (HttpURLConnection) url.openConnection();
        urlConnection.connect();
        InputStream inputStream = urlConnection.getInputStream();
        FileOutputStream fileOutputStream = parent.openFileOutput(fileName, Context.MODE_PRIVATE);
        long totalSize = urlConnection.getContentLength();
        int MEGABYTE = 1024 * 128;
        byte[] buffer = new byte[MEGABYTE];
        long bufferLength;
        long downloaded = 0;
        lastProgress = 0;
        while ((bufferLength = inputStream.read(buffer)) > 0) {
            downloaded += bufferLength;
            int progress = (int) ((downloaded * 100) / totalSize);
            handler.post(() -> {
                selectedTxtProgress.setText(progress + "%");
                selectedProgressBar.setProgress(progress);
            });
            fileOutputStream.write(buffer, 0, (int) bufferLength);
        }
        fileOutputStream.close();
        urlConnection.disconnect();
        handler.post(() -> {
            Toast.makeText(parent, "Download is completed", Toast.LENGTH_LONG).show();
        });
    });
}
```

Different Types of Executor

- **Executors.newCachedThreadPool()** — An ExecutorService with a thread pool that creates new threads as required but reuses previously created threads as they become available.
- **Executors.newFixedThreadPool(int numThreads)** — An ExecutorService that has a thread pool with a fixed number of threads. The numThreads parameter is the maximum number of threads that can be active in the ExecutorService at any one time. If the number of requests submitted to the pool exceeds the pool size, requests are queued until a thread becomes available.
- **Executors.newScheduledThreadPool(int numThreads)** — A ScheduledExecutorService with a thread pool that is used to run tasks periodically or after a specified delay.
- **Executors.newSingleThreadExecutor()** — An ExecutorService with a single thread. Tasks submitted will be executed one at a time and in the order submitted.
- **Executors.newSingleThreadScheduledExecutor()** — An ExecutorService that uses a single thread to execute tasks periodically or after a specified delay.

Revisiting Multithreading in Android

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork();  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

- Violate the single thread model: the Android UI toolkit is not thread-safe and must always be manipulated on the UI thread.
- In this piece of code, the ImageView is manipulated on a worker thread, which can cause really weird problems. Tracking down and fixing such bugs can be difficult and time-consuming

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap b = loadImageFromNetwork();  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(b);  
                }  
            });  
        }  
    }).start();  
}
```

- Classes and methods also tend to make the code more complicated and more difficult to read.
- It becomes even worse when our implemented complex operations that require frequent UI updates

```
public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask {
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

Cancel Task

- DownloadFilesTask dft = new DownloadFilesTask().execute(url1, url2, url3);
- dft.cancel(true);

Summary

- Executing a long running **blocking task** inside the click event **freezes the UI**.
- Starting a **new thread** to do the blocking task solves the responsiveness problem, but the new thread **cannot access UI elements** declared in UI thread.
- **Async Task** solves all the problems. It executes **doInBackground()** on a worker thread and **onPostExecute()** on the UI thread.

Example 2. Progress Bar – Using Message Passing Layout 1/2

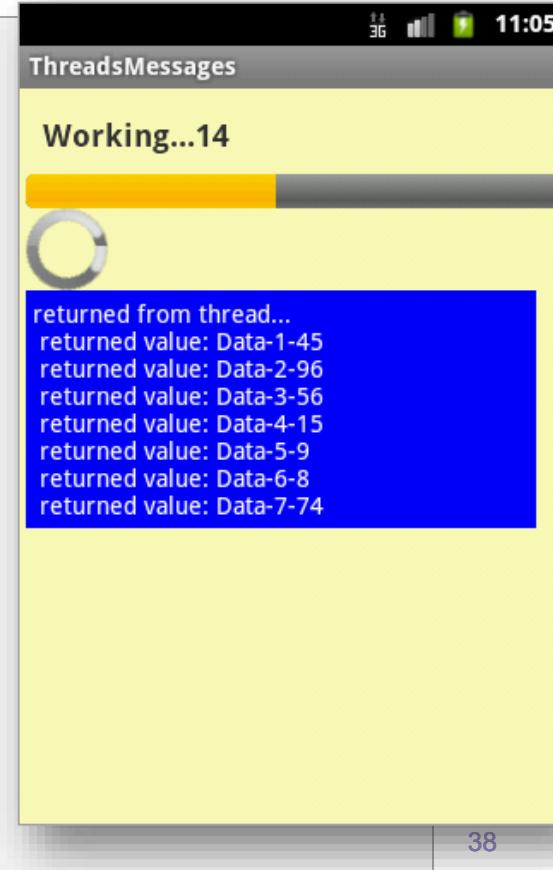
The main thread displays a horizontal and a circular *progress bar widget* showing the progress of a slow background operation. Some random data is periodically sent from the background thread and the messages are displayed in the main view.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#44ffff00"
    android:orientation="vertical"
    android:padding="4dp" >

    <TextView
        android:id="@+id/txtWorkProgress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:text="Working ...."
        android:textSize="18sp"
        android:textStyle="bold" />

    <ProgressBar
        android:id="@+id/progress1"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <ProgressBar
        android:id="@+id/progress2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

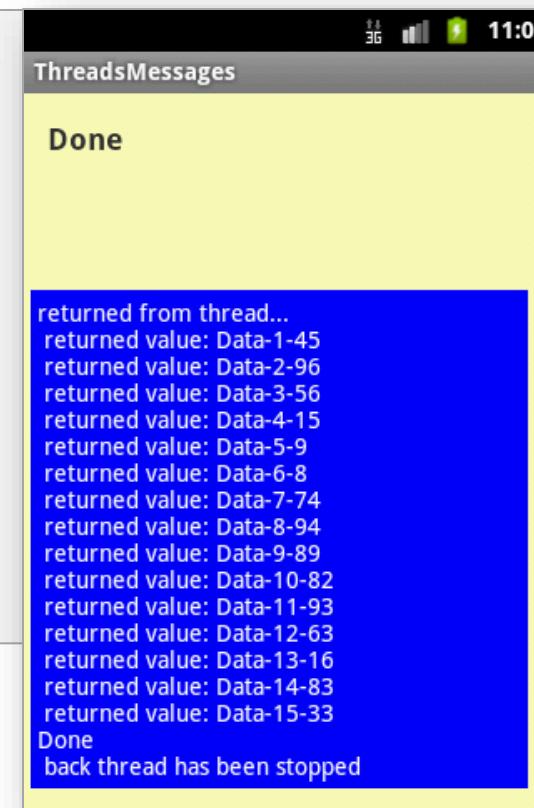


Example 2. Progress Bar – Using Message Passing

Layout 2/2

The main thread displays a horizontal and a circular *progress bar* widget showing the progress of a slow background operation. Some random data is periodically sent from the background thread and the messages are displayed in the main view.

```
<ScrollView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" >  
  
    <TextView  
        android:id="@+id/txtReturnedValues"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_margin="7dp"  
        android:background="#ff0000ff"  
        android:padding="4dp "  
        android:text="returned from thread..."  
        android:textColor="@android:color/white"  
        android:textSize="14sp" />  
    </ScrollView>  
  
</LinearLayout>
```

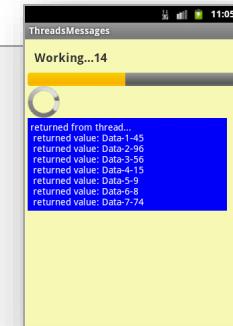


Example 2. Progress Bar – Using Message Passing

Activity 1/5

The main thread displays a horizontal and a circular *progress bar widget* showing the progress of a slow background operation. Some random data is periodically sent from the background thread and the messages are displayed in the main view.

```
public class ThreadsMessages extends Activity {  
  
    ProgressBar bar1;  
    ProgressBar bar2;  
  
    TextView msgWorking;  
    TextView msgReturned;  
  
    // this is a control var used by backg. threads  
    boolean isRunning = false;  
  
    // lifetime (in seconds) for background thread  
    final int MAX_SEC = 30;  
  
    //String globalStrTest = "global value seen by all threads ";  
    int globalIntTest = 0;
```



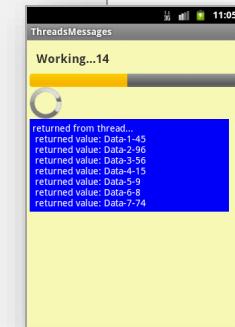
Example 2. Progress Bar – Using Message Passing

Activity 4/5

```
public void onStart() {
    super.onStart();
    // this code creates the background activity where busy work is done
    Thread background = new Thread(new Runnable() {
        public void run() {
            try {
                for (int i = 0; i < MAX_SEC && isRunning; i++) {
                    //try a Toast method here (it will not work!)
                    //fake busy busy work here
                    → Thread.sleep(1000); //one second at a time

                    // this is a locally generated value between 0-100
                    Random rnd = new Random();
                    int localData = (int) rnd.nextInt(101);
                    //we can see and change (global) class variables
                    String data = "Data-" + globalIntTest + "-" + localData;
                    globalIntTest++;
                    //request a message token and put some data in it
                    → Message msg = handler.obtainMessage(1, (String)data);

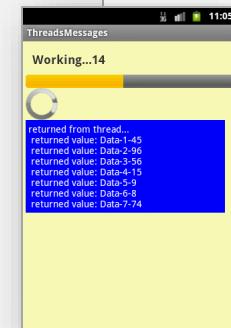
                    // if thread is still alive send the message
                    → if (isRunning) {
                        handler.sendMessage(msg);
                    }
                }
            }
        }
    });
}
```



Example 2. Progress Bar – Using Message Passing

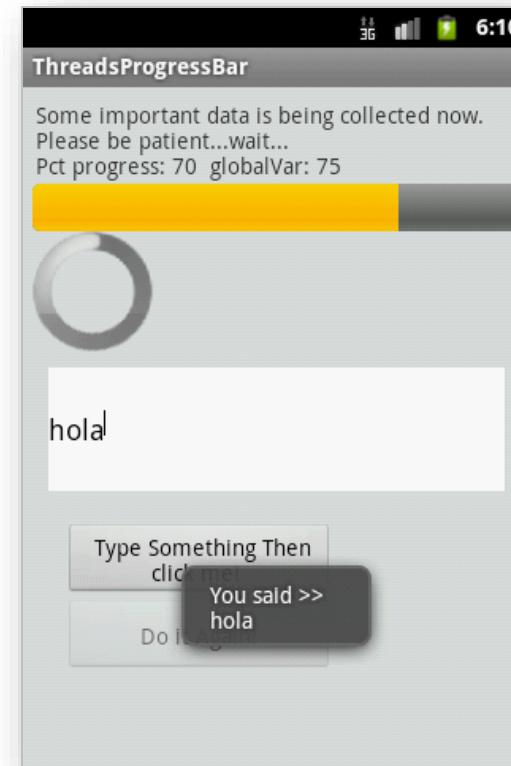
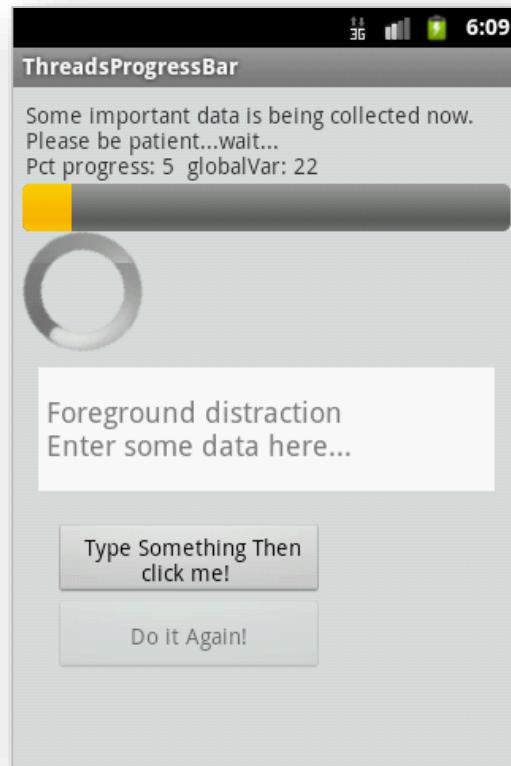
Activity 5/5

```
catch (Throwable t) {  
    // just end the background thread  
    isRunning = false;  
}  
}  
}); // Thread  
  
isRunning = true;  
background.start();  
  
}//onStart  
  
  
public void onStop() {  
    super.onStop();  
    isRunning = false;  
} //onStop  
} //class
```



Example 3. Using Handler post(...) Method

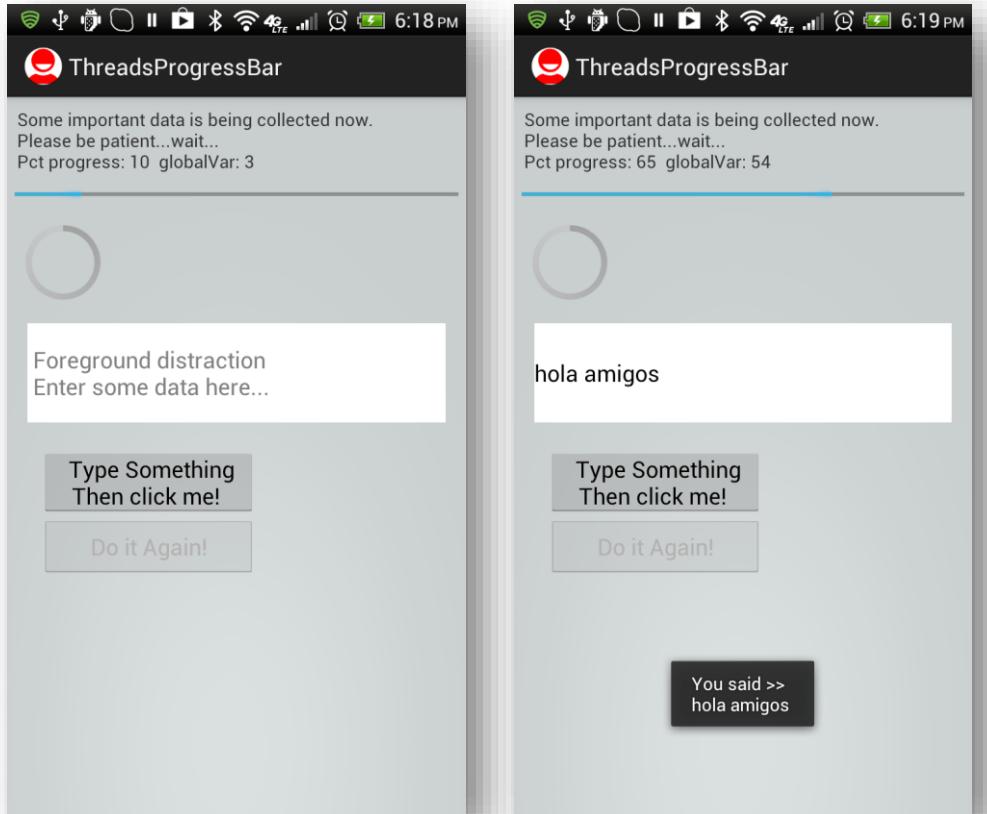
We will try the same problem presented earlier (a slow background task and a responsive foreground UI) this time using the **posting mechanism** to execute foreground *runnables*.



Images obtained from a GingerBread based emulator

Example 3. Using Handler post(...) Method

We will try the same problem presented earlier (a slow background task and a responsive foreground UI) this time using the **posting mechanism** to execute foreground *runnables*.



Images obtained from an IceCream 4.x based device

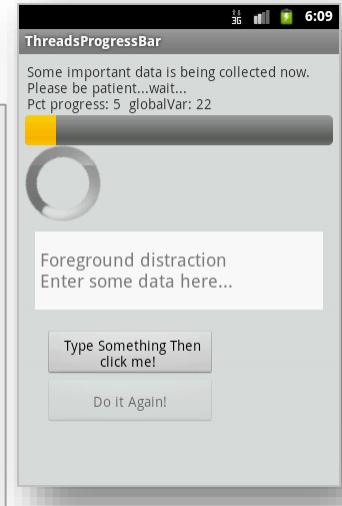
Example 3. Using post - layout: main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#22002222"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/LblTopCaption"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="2dp"
        android:text=
            "Some important data is been collected now. Patience please..." />

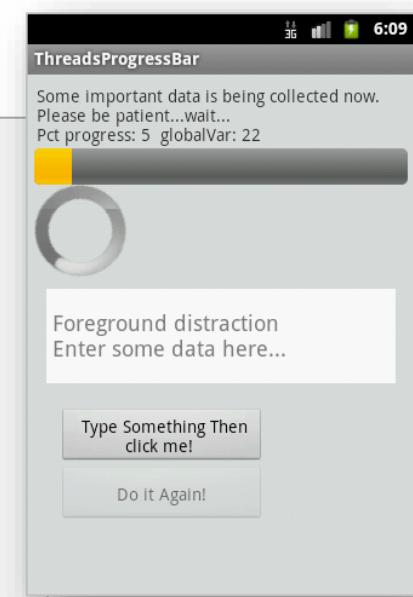
    <ProgressBar
        → android:id="@+id/myBarHor"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="30dp" />

    <ProgressBar
        → android:id="@+id/myBarCir"
        style="?android:attr/progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```



Example 3. Using post - layout: main.xml

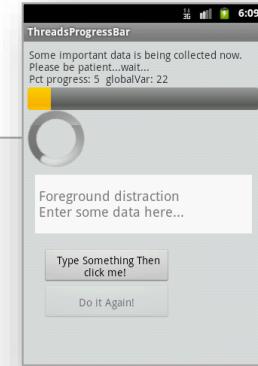
```
<EditText  
    android:id="@+id/txtBox1"  
    android:layout_width="match_parent"  
    android:layout_height="78dp"  
    android:layout_margin="10dp"  
    android:background="#ffffffff"  
    android:textSize="18sp" />  
  
<Button  
    android:id="@+id/btnDoSomething"  
    android:layout_width="170dp"  
    android:layout_height="wrap_content"  
    android:layout_marginLeft="20dp"  
    android:layout_marginTop="10dp"  
    android:padding="4dp"  
    android:text=" Type Something Then click me! " />  
  
<Button  
    android:id="@+id/btnDoItAgain"  
    android:layout_width="170dp"  
    android:layout_height="wrap_content"  
    android:layout_marginLeft="20dp"  
    android:padding="4dp"  
    android:text=" Do it Again! " />  
  
</LinearLayout>
```



Example 3. Using post - Main Activity

1/5

```
public class ThreadsPosting extends Activity {  
    ProgressBar myBarHorizontal;  
    ProgressBar myBarCircular;  
  
    TextView lblTopCaption;  
    EditText txtDataBox;  
    Button btnDoSomething;  
    Button btnDoItAgain;  
    int progressStep = 5;  
  
    int globalVar = 0;  
    int accum = 0;  
  
    long startingMills = System.currentTimeMillis();  
    boolean isRunning = false;  
    String PATIENCE = "Some important data is being collected now. "  
        + "\nPlease be patient...wait... ";  
  
    → Handler myHandler = new Handler();  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
  
        lblTopCaption = (TextView) findViewById(R.id.lblTopCaption);
```



Example 3. Using post - Main Activity

2/5

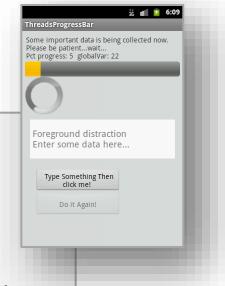
```
myBarHorizontal = (ProgressBar) findViewById(R.id.myBarHor);
myBarCircular = (ProgressBar) findViewById(R.id.myBarCir);

txtDataBox = (EditText) findViewById(R.id.txtBox1);
txtDataBox.setHint(" Foreground distraction\n Enter some data here...");

btnDoItAgain = (Button) findViewById(R.id.btnDoItAgain);
btnDoItAgain.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        onStart();
    }// onClick
}); // setOnClickListener

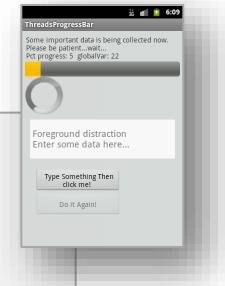
btnDoSomething = (Button) findViewById(R.id.btnDoSomething);
btnDoSomething.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Editable text = txtDataBox.getText();
        Toast.makeText(getApplicationContext(), "You said >> \n" + text, 1).show();
    }// onClick
}); // setOnClickListener

} // onCreate
```



Example 3. Using post - Main Activity

3/5



```
@Override  
protected void onStart() {  
    super.onStart();  
    // prepare UI components  
    txtDataBox.setText("");  
    btnDoItAgain.setEnabled(false);  
  
    // reset and show progress bars  
    accum = 0;  
    myBarHorizontal.setMax(100);  
    myBarHorizontal.setProgress(0);  
    myBarHorizontal.setVisibility(View.VISIBLE);  
    myBarCircular.setVisibility(View.VISIBLE);  
  
    // create background thread were the busy work will be done  
    Thread myBackgroundThread = new Thread( backgroundTask, "backAlias1" );  
    myBackgroundThread.start();  
}  
→
```

Example 3. Using post - Main Activity

4/5

```
// FOREGROUND
// this foreground Runnable works on behalf of the background thread
// updating the main UI which is unreachable to it
private Runnable foregroundRunnable = new Runnable() {
    @Override
    public void run() {
        try {
            // update UI, observe globalVar is changed in back thread
            lblTopCaption.setText( PATIENCE
                + "\nPct progress: " + accum
                + " globalVar: " + globalVar);

            // advance ProgressBar
            myBarHorizontal.incrementProgressBy(progressStep);
            accum += progressStep;

            // are we done yet?
            if (accum >= myBarHorizontal.getMax()) {
                lblTopCaption.setText("Background work is OVER!");
                myBarHorizontal.setVisibility(View.INVISIBLE);
                myBarCircular.setVisibility(View.INVISIBLE);
                btnDoItAgain.setEnabled(true);
            }
        } catch (Exception e) {
            Log.e("<>foregroundTask><", e.getMessage());
        }
    }
}; // foregroundTask
```

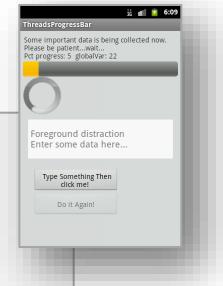


Foreground
Runnable is
defined but not
started !

Background
thread will
request its
execution later

Example 3. Using post - Main Activity

5/5



```
// BACKGROUND
// this is the back runnable that executes the slow work

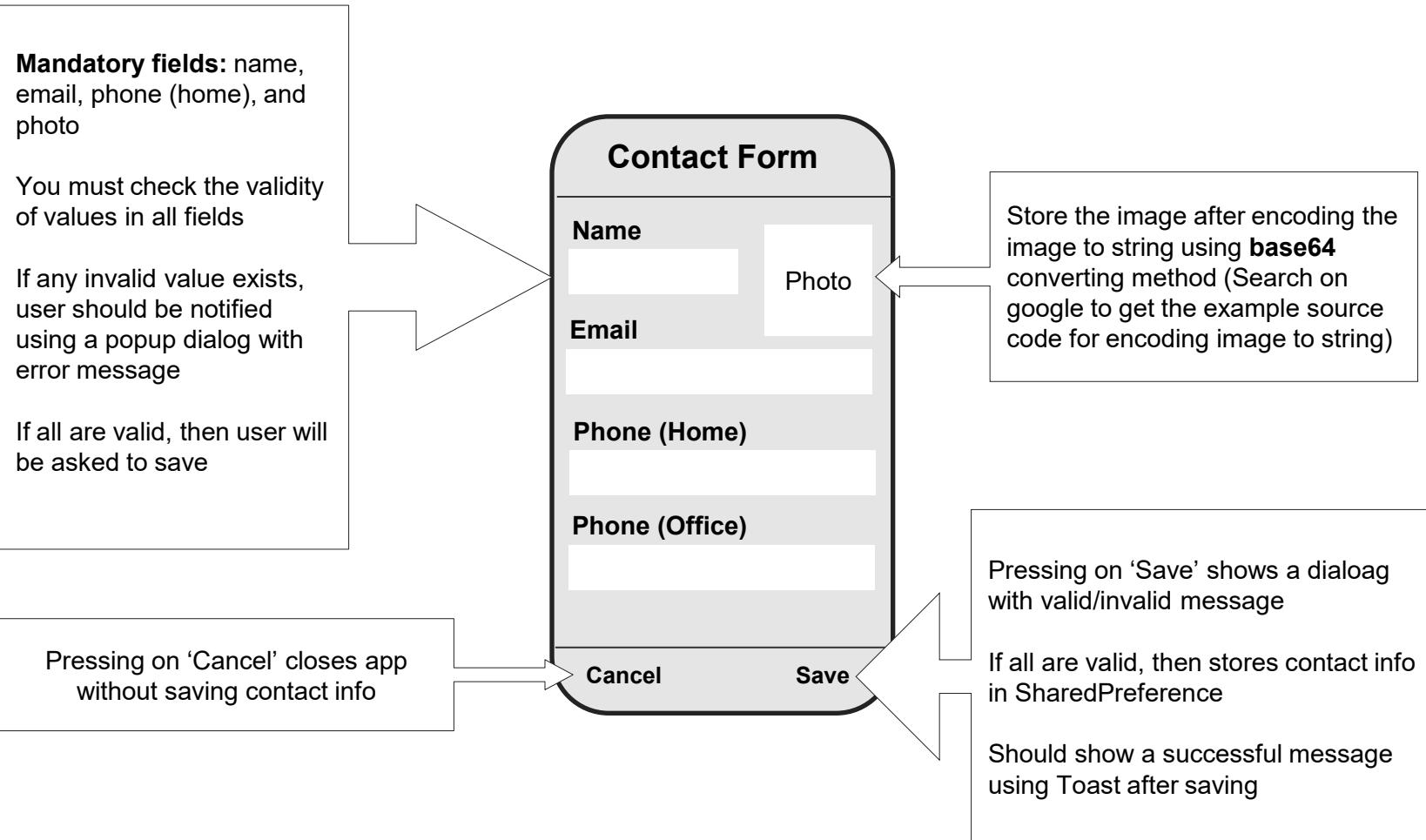
private Runnable backgroundTask = new Runnable() {
    @Override
    public void run() {
        // busy work goes here...
        try {
            for (int n = 0; n < 20; n++) {
                // this simulates 1 sec. of busy activity
                Thread.sleep(1000);
                // change a global variable from here...
                globalVar++;
                // try: next two UI operations should NOT work
                // Toast.makeText(getApplicationContext(), "Hi ", 1).show();
                // txtDataBox.setText("Hi ");

                // wake up foregroundRunnable delegate to speak for you
                myHandler.post(foregroundRunnable);
            }
        } catch (InterruptedException e) {
            Log.e("<><>foregroundTask>>", e.getMessage());
        }
    }
}; // run
}; // backgroundTask

} // ThreadsPosting
```

Tell foreground
runnable to do
something for us...

Assignment-2: Use SQLite



END