13

Android Multi-Threading

Victor Matos
Cleveland State University

Notes are based on:

The Busy Coder's Guide to Android Development
by Mark L. Murphy

ISBN: 978-0-9816780-0-9
&
Android Developers
http://developer.android.com/index.html

Multi-Threading

Threads

http://developer.android.com/reference/java/lang/Thread.html

1. A Thread is a **concurrent** unit of execution.

2. It thread has its own call stack for methods being invoked, their arguments and local variables.

3. Each virtual machine instance has at least one main Thread running when it is started; typically, there are several others for housekeeping.

4. The application might decide to launch additional Threads for specific purposes.

Multi-Threading

Threads

http://developer.android.com/reference/java/lang/Thread.html

Threads in the same VM interact and synchronize by the use of **shared objects** and **monitors** associated with these objects.

There are basically two main ways of having a Thread execute application code.

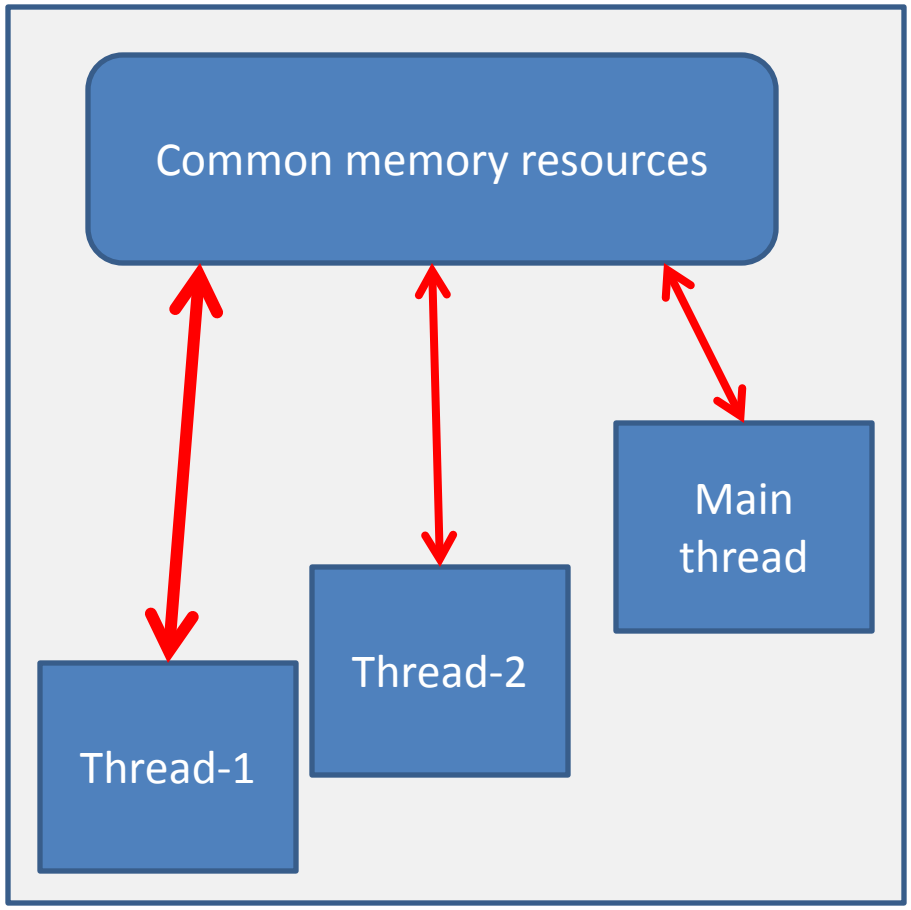
1. Create a new class that *extends* Thread and override its **run()** method.

2. Create a new Thread instance passing to it a **Runnable** object.

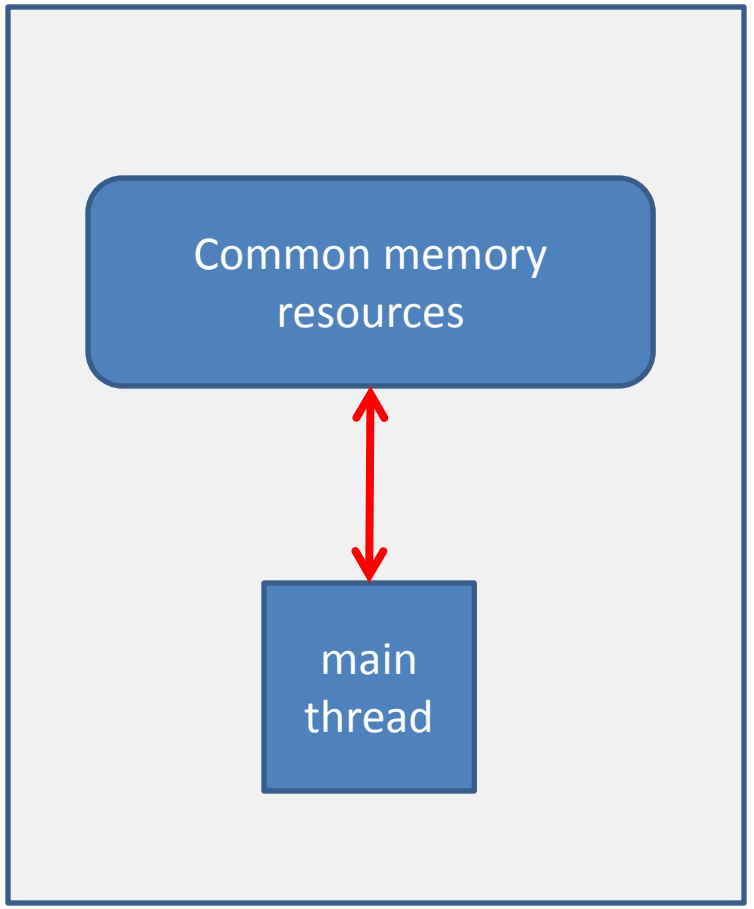
In both cases, the **start()** method must be called to actually execute the new Thread.

Multi-Threading

Process 1 (Dalvik Virtual Machine 1)



Process 2 (Dalvik Virtual Machine 2)



Multi-Threading

Advantages of Multi-Threading

1. Threads share the process' resources but are able to execute independently.
2. Applications responsibilities can be separated
   - main thread runs UI, and
   - slow tasks are sent to background threads.
3. Threading provides an useful abstraction of concurrent execution.
4. Particularly useful in the case of a single process that spawns multiple threads on top of a *multiprocessor* system. In this case *real parallelism* is achieved.
5. Consequently, a multithreaded program operates *faster* on computer systems that have *multiple CPUs.*

Multi-Threading

Disadvantages of Multi-Threading

1. Code tends to be more complex

2. Need to detect, avoid, resolve **deadlocks**

Multi-Threading

Android's Approach to Slow Activities

An application may involve a time-consuming operation, however we want the **UI** to be responsive to the user. Android offers two ways for dealing with this scenario:

1. Do expensive operations in a background *service*, using *notifications* to inform users about next step

2. Do the slow work in a *background thread*.

Interaction between Android threads is accomplished using (a) ***Handler*** objects and (b) posting ***Runnable*** objects to the main view.

Multi-Threading

Handler Class

http://developer.android.com/reference/android/os/Handler.html

- When a process is created for your application, its ***main thread*** is dedicated to running a ***message queue*** that takes care of managing the top-level application objects (activities, intent receivers, etc) and any windows they create.

- You can create your own secondary threads, and communicate back with the main application thread through a **Handler**.

- When you create a new Handler, it is bound to the message queue of the thread that is creating it -- from that point on, it will deliver *messages* and *runnables* to that message queue and execute them as they come out of the message queue.

Multi-Threading

Handler Class

http://developer.android.com/reference/android/os/Handler.html

There are two main uses for a Handler:

(1) to schedule messages and runnables to be executed as some point in the future; and

(2) to enqueue an action to be performed on another thread

Multi-Threading

Threads and UI

Warning

Background threads are not allowed to interact with the UI.

Only the main process can access the (main) activity's view.

(Global) class variables can be seen and updated in the threads

Multi-Threading

Handler‘s MessageQueue

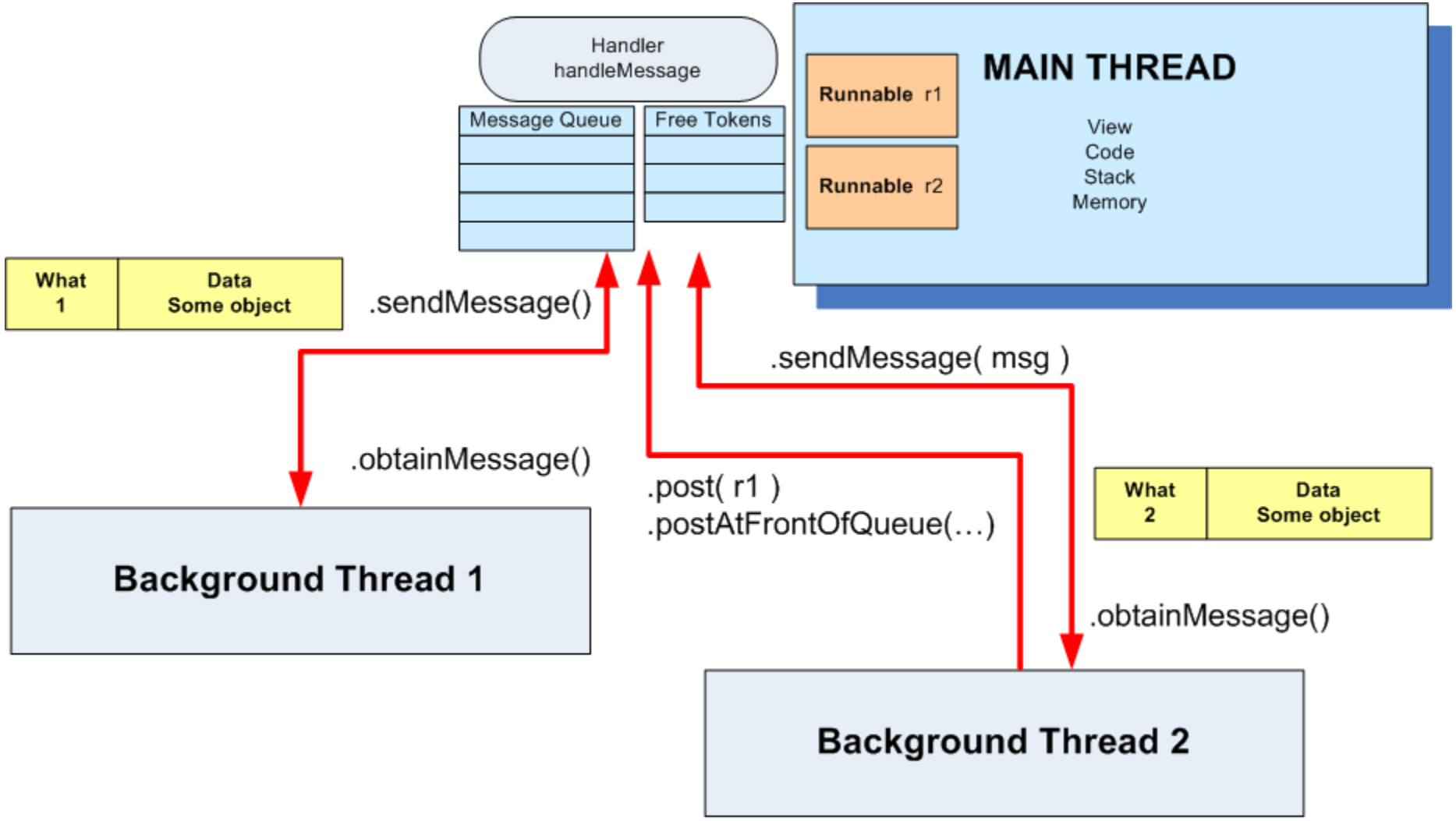
A secondary thread that wants to communicate with the main thread must request a message token using the *obtainMessage*() method.

Once obtained, the background thread can fill data into the message token and attach it to the Handler’s message queue using the *sendMessage*() method.

The Handler uses the *handleMessage*() method to continuously attend new messages arriving to the main thread.

A message extracted from the process’ queue can either return some data to the main process or request the execution of runnable objects through the *post*() method.

Multi-Threading



Multi-Threading

Using Messages

Main Thread

```
...
Handler myHandler = new Handler() {

   @Override
   public void handleMessage(Message msg) {

      // do something with the message...
      // update GUI if needed!
      ...
   }//handleMessage

};//myHandler

...
```

Background Thread

```
...
Thread backgJob = new Thread (new Runnable (){

   @Override
   public void run() {
    //...do some busy work here ...
    //get a token to be added to
     //the main's message queue
    Message msg = myHandler.obtainMessage();
      ...
     //deliver message to the
    //main's message-queue
    myHandler.sendMessage(msg);
   }//run

});//Thread

//this call executes the parallel thread
backgroundJob.start();
...
```

Multi-Threading

Using Post

Main Thread

```
...
Handler     myHandler = new Handler();
@Override
public void onCreate(
          Bundle savedInstanceState) {
 ...
 Thread myThread1 =
        new Thread(backgroundTask,
                   "backAlias1");
 myThread1.start();

}//onCreate

...
//this is the foreground runnable
private Runnable foregroundTask
   = new Runnable() {
  @Override
  public void run() {
   // work on the UI if needed
}
...
```

Background Thread

```
// this is the "Runnable" object
// that executes the background thread

private Runnable backgroundTask
                  = new Runnable () {
  @Override
  public void run() {
    ... Do some background work here
    myHandler.post(foregroundTask);

  }//run

};//backgroundTask
```

Multi-Threading

Messages

To send a Message to a Handler, the thread must first invoke obtainMessage() to get the Message object out of the pool.

There are a few forms of **obtainMessage()**, allowing you to just create an empty Message object, or messages holding arguments

Example

```
//  thread 1 produces some local data
String localData = “Greeting from thread 1”;
//  thread 1 requests a message & adds localData to it
Message mgs = myHandler.obtainMessage (1, localData);
```

Multi-Threading

sendMessage Methods

You deliver the message using one of the **sendMessage...()** family of methods, such as …

- **sendMessage()** puts the message at the end of the queue immediately

- **sendMessageAtFrontOfQueue()** puts the message at the front of the queue immediately (versus the back, as is the default), so your message takes priority over all others

- **sendMessageAtTime()** puts the message on the queue at the stated time, expressed in the form of milliseconds based on system uptime (SystemClock.uptimeMillis())

- **sendMessageDelayed()** puts the message on the queue after a delay, expressed in milliseconds

Multi-Threading

Processing Messages

To process messages sent by the background threads, your Handler needs to implement the listener

handleMessage(. . .)

which will be called with each message that appears on the message queue.

There, the handler can update the UI as needed. However, it should still do that work quickly, as other UI work is suspended until the Handler is done.

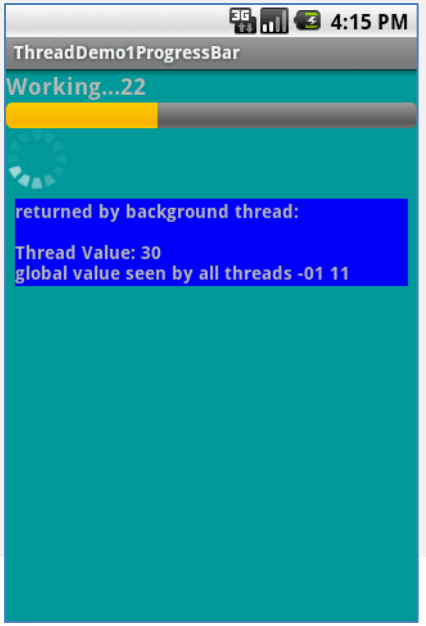
Multi-Threading

Example 1. Progress Bar – Using Message Passing

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
android:id="@+id/widget28"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:background="#ff009999"
android:orientation="vertical"
xmlns:android="http://schemas.android.com/apk/res/android"
>
<TextView
       android:id="@+id/TextView01"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:text="Working ...."
       android:textSize="18sp"
       android:textStyle="bold"  />
<ProgressBar
       android:id="@+id/progress"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       style="?android:attr/progressBarStyleHorizontal" />

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

```
<TextView
       android:id="@+id/TextView02"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:text="returned from thread..."
       android:textSize="14sp"
       android:background="#ff0000ff"
       android:textStyle="bold"
       android:layout_margin="7px"/>

</LinearLayout>
```



Multi-Threading

Example 1. Progress Bar – Using Message Passing

```
// Multi-threading example using message passing
package cis493.threads;

import java.util.Random;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.View;
import android.widget.ProgressBar;
import android.widget.TextView;

public class ThreadDemo1ProgressBar extends Activity {
ProgressBar bar1;
ProgressBar bar2;
TextView msgWorking;
TextView msgReturned;

boolean isRunning = false;
final int MAX_SEC =  60; // (seconds) lifetime for background thread

String strTest = "global value seen by all threads ";
int intTest = 0;
```

Multi-Threading

Example 1. Progress Bar – Using Message Passing

```
Handler handler = new Handler() {
     @Override
     public void handleMessage(Message msg) {
          String returnedValue = (String)msg.obj;
          //do something with the value sent by the background thread here ...
          msgReturned.setText("returned by background thread: \n\n"
                 + returnedValue);
          bar1.incrementProgressBy(2);

          //testing thread's termination
          if (bar1.getProgress() == MAX_SEC){
               msgReturned.setText("Done \n back thread has been stopped");
               isRunning = false;
          }
          if (bar1.getProgress() == bar1.getMax()){
               msgWorking.setText("Done");
               bar1.setVisibility(View.INVISIBLE);
               bar2.setVisibility(View.INVISIBLE);
               bar1.getLayoutParams().height = 0;
               bar2.getLayoutParams().height = 0;
          }
          else {
               msgWorking.setText("Working..." +
               bar1.getProgress() );
          }
     }
}; //handler
```

Multi-Threading

Example 1. Progress Bar – Using Message Passing

```
@Override
public void onCreate(Bundle icicle) {

     super.onCreate(icicle);
     setContentView(R.layout.main);

     bar1 = (ProgressBar) findViewById(R.id.progress);
     bar2 = (ProgressBar) findViewById(R.id.progress2);
     bar1.setMax(MAX_SEC);
     bar1.setProgress(0);

     msgWorking = (TextView)findViewById(R.id.TextView01);
     msgReturned = (TextView)findViewById(R.id.TextView02);

     strTest += "-01"; // slightly change the global string
     intTest = 1;

}//onCreate



public void onStop() {

     super.onStop();
     isRunning = false;
}
```

Multi-Threading

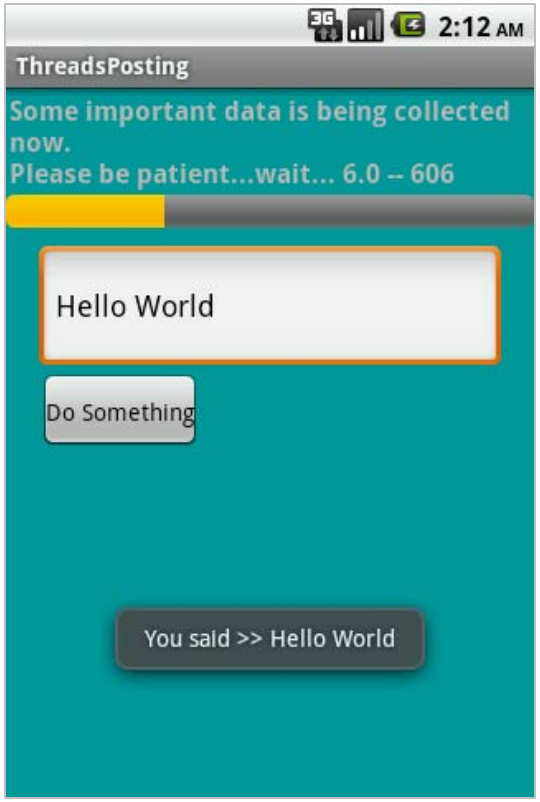
Example 1. Progress Bar – Using Message Passing

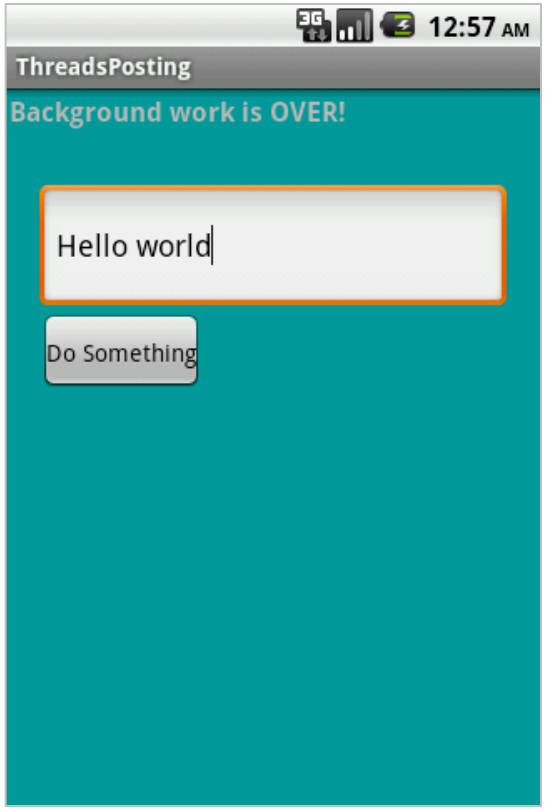
```
  public void onStart() {
      super.onStart();
      // bar1.setProgress(0);
      Thread background = new Thread(new Runnable() {
                  public void run() {
                  try {
                        for (int i = 0; i < MAX_SEC && isRunning; i++) {
                              //try a Toast method here (will not work!)
                              //fake busy busy work here
                              Thread.sleep(1000);  //one second at a time
                              Random rnd = new Random();

                              // this is a locally generated value
                              String data = "Thread Value: " + (int) rnd.nextInt(101);

                              //we can see and change (global) class variables
                              data += "\n" + strTest + " " + intTest;
                              intTest++;

                              //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);
                              }
                        }
                  } catch (Throwable t) {
                        // just end the background thread
                  }
                  }//run
            });//background
      isRunning = true;
      background.start();
  }//onStart
} //class
```

Multi-Threading

Example 2. 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*.





Multi-Threading

Example2. Using Handler post(...) Method

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
       android:id="@+id/linearLayout1"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent"
       android:background="#ff009999"
       android:orientation="vertical"
       xmlns:android=http://schemas.android.com/apk/res/android >
<TextView
       android:id="@+id/lblTopCaption"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:padding="2px"
       android:text="Some important data is being collected now. Patience please..."
       android:textSize="16sp"
       android:textStyle="bold"  />
<ProgressBar
       android:id="@+id/myBar"
       style="?android:attr/progressBarStyleHorizontal"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content" />
<EditText
       android:id="@+id/txtBox1"
       android:layout_width="fill_parent"
       android:layout_height="78px"
       android:layout_marginLeft="20px"
       android:layout_marginRight="20px"
       android:textSize="18sp" android:layout_marginTop="10px"  />
<Button
       android:id="@+id/btnDoSomething"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:padding="4px"
       android:layout_marginLeft="20px"
       android:text="Do Something" />
</LinearLayout>
```

Multi-Threading

Example2. Using Handler post(...) Method

```
// using Handler post(...) method to execute
// foreground/background runnables
package cis493.threads;

import . . .

public class ThreadsPosting extends Activity {
  ProgressBar myBar;

  TextView   lblTopCaption;
  EditText   txtBox1;
  Button     btnDoSomething;

  int        globalVar = 0;   // to be used by threads to exchange data

  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();
```

Multi-Threading

Example2. Using Handler post(...) Method

```
// ////////////////////////////////////////////////////////////////
@Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      setContentView(R.layout.main);
      lblTopCaption = (TextView)findViewById(R.id.lblTopCaption);

      myBar = (ProgressBar) findViewById(R.id.myBar);
      myBar.setMax(100); // range goes from 0..100

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

      btnDoSomething = (Button)findViewById(R.id.btnDoSomething);
      btnDoSomething.setOnClickListener(new OnClickListener() {
          @Override
          public void onClick(View v) {
           Editable txt = txtBox1.getText();
           Toast.makeText(getBaseContext(),
                "You said >> " + txt, 1).show();
          }//onClick
      });//setOnClickListener

  }//onCreate
```

Multi-Threading

Example2. Using Handler post(...) Method

```
@Override
protected void onStart() {
     super.onStart();
     // create & execute background thread were the busy work will be done

     Thread myThreadBack = new Thread(backgroundTask, "backAlias1"  );

     myThreadBack.start();

     myBar.incrementProgressBy(0);
}
```

Multi-Threading

Example2. Using Handler post(...) Method

```
private Runnable foregroundTask = new Runnable() {
 @Override
 public void run() {

     try {
     int progressStep = 5;
     double totalTime = (System.currentTimeMillis() - startingMills)/1000;

     synchronized(this) {
       globalVar += 100;
     };

     lblTopCaption.setText(PATIENCE + totalTime + " -- " + globalVar);

     myBar.incrementProgressBy(progressStep);

     accum += progressStep;
     if (accum >= myBar.getMax()){
       lblTopCaption.setText("Background work is OVER!");
       myBar.setVisibility(View.INVISIBLE);
     }
   } catch (Exception e) {
     Log.e("<<foregroundTask>>", e.getMessage());
   }
 }

}; //foregroundTask
```

synchronizing

Runnable is defined but not started !
Back thread will requests its execution later

Multi-Threading

Example2. Using Handler post(...) Method

```
//this is the "Runnable" object that executes the background thread
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);
            // now talk to the main thread
            // optionally change some global variable such as: globalVar

            synchronized(this) {
              globalVar += 1;
            };

            myHandler.post(foregroundTask);
        }
      } catch (InterruptedException e) {
        Log.e("<<foregroundTask>>", e.getMessage());
      }

    }//run
    };//backgroundTask


}//ThreadsPosting
```

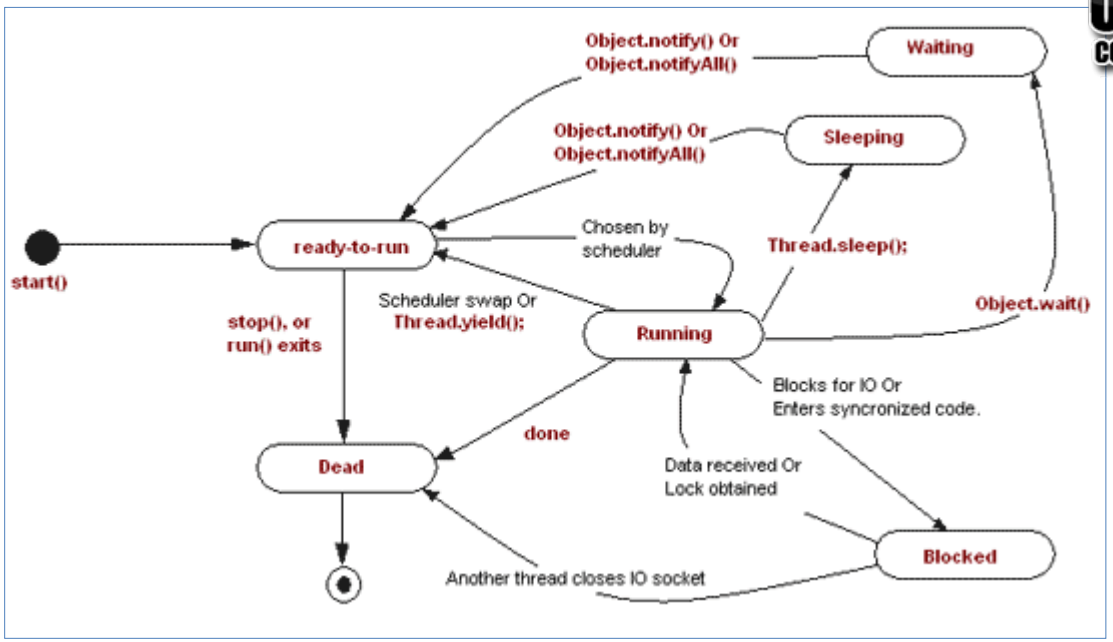
synchronizing

Tell foreground runnable to do something for us…

Multi-Threading

Thread States

Android's threads run in a manner similar to common Java threads



Thread.State	Description
BLOCKED	The thread is blocked and waiting for a lock.
NEW	The thread has been created, but has never been started.
RUNNABLE	The thread may be run.
TERMINATED	The thread has been terminated.
TIMED_WAITING	The thread is waiting for a specified amount of time.
WAITING	The thread is waiting.

Multi-Threading

Using the AsyncTask class

```
private class VerySlowTask extends AsyncTask<String, Long, Void> {

    // Begin - can use UI thread here
    protected void onPreExecute() {

    }

    // this is the SLOW background thread taking care of heavy tasks
    // cannot directly change UI
    protected Void doInBackground(final String... args) {
    ... publishProgress((Long) someLongValue);
    }

    // periodic updates - it is OK to change UI
    @Override
    protected void onProgressUpdate(Long... value) {

    }

    // End - can use UI thread here
    protected void onPostExecute(final Void unused) {

    }
}
```

Multi-Threading

Using the AsyncTask class

1. **AsyncTask** enables proper and easy use of the UI thread.

2. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

3. An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

4. An asynchronous task is defined by

3 Generic Types	4 Main States	1 Auxiliary Method
Params, **Progress**, **Result**	**onPreExecute**, **doInBackground**, **onProgressUpdate** **onPostExecute**.	**publishProgress**

Multi-Threading

```
AsyncTask <Params, Progress, Result>
```

AsyncTask's generic types	
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.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type **Void**

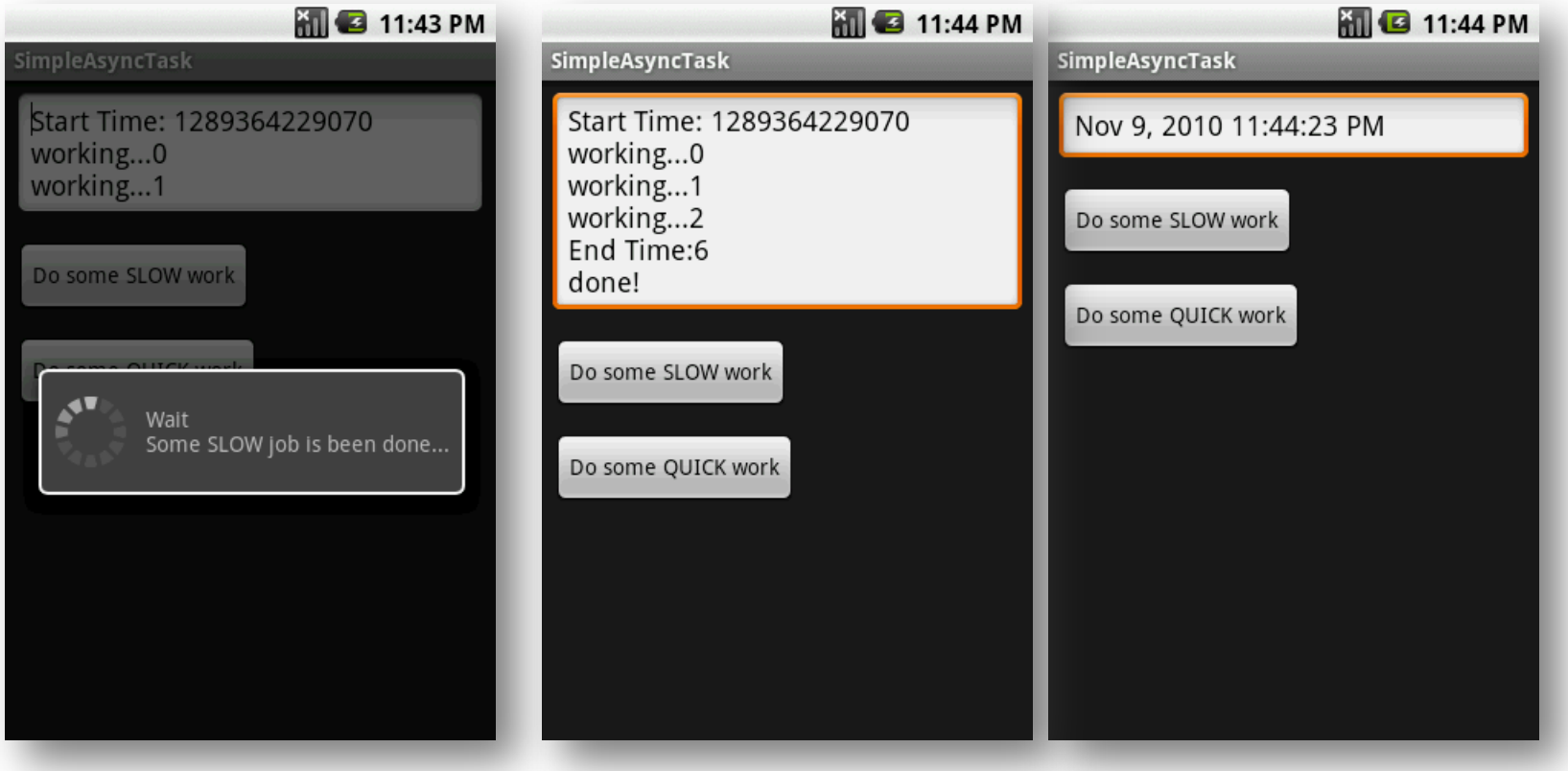
Note:
Syntax "**`String ...`**" indicates (Varargs) array of String values, similar to "**`String[]`**"

Multi-Threading

AsyncTask's methods
onPreExecute(), invoked on the UI thread immediately after the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
doInBackground(Params...), invoked on the background thread immediately after *onPreExecute*() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use *publishProgress(Progress...)* to publish one or more units of progress. These values are published on the UI thread, in the *onProgressUpdate(Progress...)* step.
onProgressUpdate(Progress...), invoked on the UI thread after a call to *publishProgress(Progress...)*. The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Multi-Threading

Example: Using the AsyncTask class



The main task invokes an AsyncTask to do some slow job. The AsyncTask methods do the required computation and periodically update the main's UI. In our the example the background activity negotiates the writing of the lines in the text box, and also controls the circular progress bar.

Multi-Threading

Example: Using the AsyncTask class

```
public class Main extends Activity {
Button btnSlowWork;
Button btnQuickWork;
EditText etMsg;
Long  startingMillis;

@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      etMsg = (EditText) findViewById(R.id.EditText01);

      btnSlowWork = (Button) findViewById(R.id.Button01);
      // slow work...for example: delete all data from a database or get data from Internet
      this.btnSlowWork.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                  new VerySlowTask().execute();
            }
      });

      btnQuickWork = (Button) findViewById(R.id.Button02);
      // delete all data from database (when delete button is clicked)
      this.btnQuickWork.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                  etMsg.setText((new Date()).toLocaleString());
            }
      });

}// onCreate
```

Multi-Threading

Example: Using the AsyncTask class

```
private class VerySlowTask extends AsyncTask <String, Long, Void> {

     private final ProgressDialog dialog = new ProgressDialog(Main.this);

     // can use UI thread here
     protected void onPreExecute() {
          startingMillis = System.currentTimeMillis();
          etMsg.setText("Start Time: " + startingMillis);
          this.dialog.setMessage("Wait\nSome SLOW job is being done...");
          this.dialog.show();
     }

     // automatically done on worker thread (separate from UI thread)
     protected Void doInBackground(final String... args) {
       try {
          // simulate here the slow activity
          for (Long i = 0L; i < 3L; i++) {
               Thread.sleep(2000);
               publishProgress((Long)i);
               }
       } catch (InterruptedException e) {
               Log.v("slow-job interrupted", e.getMessage())
       }
       return null;
     }
```

Multi-Threading

Example: Using the AsyncTask class

```
      // periodic updates - it is OK to change UI
      @Override
      protected void onProgressUpdate(Long... value) {
            super.onProgressUpdate(value);

            etMsg.append("\nworking..." + value[0]);
      }


      // can use UI thread here
      protected void onPostExecute(final Void unused) {

            if (this.dialog.isShowing()) {
                  this.dialog.dismiss();
            }

            // cleaning-up, all done
            etMsg.append("\nEnd Time:"
                        + (System.currentTimeMillis()-startingMillis)/1000);
            etMsg.append("\ndone!");
      }

 }//AsyncTask


}// Main
```

Multi-Threading

Example: Using the AsyncTask class

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

<EditText
     android:id="@+id/EditText01"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:layout_margin="7px" />
<Button
     android:text="Do some SLOW work"
     android:id="@+id/Button01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_margin="7px"  />
<Button
     android:text="Do some QUICK work"
     android:id="@+id/Button02"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_margin="7px"  />
</LinearLayout>
```

Multi-Threading

Questions

Multi-Threading

Appendix A.

What is the difference between implementing Runnable and extending Thread?

"One difference between implementing *Runnable* and extending *Thread* is that by extending *Thread*, each of your threads has a unique object associated with it, whereas implementing *Runnable*, many threads can share the same object instance."

"In most cases, the *Runnable* interface should be used if you are only planning to override the *run()* method and no other *Thread* methods."

More at link (visited Oct-19)
http://www.xyzws.com/Javafaq/what-is-the-difference-between-implementing-runnable-and-extending-thread/29

Multi-Threading

Appendix A.

What is the difference between implementing Runnable and extending Thread?

```
public class Program {
  public static void main (String[] args) {
    Runner r = new Runner();
    Thread t1 = new Thread(r, "Thread A");
    Thread t2 = new Thread(r, "Thread B");
    Thread s1 = new Strider("Thread C");
    Thread s2 = new Strider("Thread D");
    t1.start();
    t2.start();
    s1.start();
    s2.start();
  }
}
```

Multi-Threading

Appendix A.

What is the difference between implementing Runnable and extending Thread?

```
class Runner implements Runnable {
  private int counter;
  public void run() {
    try {
      for (int i = 0; i != 2; i++) {
        System.out.println(Thread.currentThread().getName() + ": "
              + counter++);
        Thread.sleep(1000);
      }
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Multi-Threading

Appendix A.

What is the difference between implementing Runnable and extending Thread?

```
class Strider extends Thread {
  private int counter;
  Strider(String name)    {
    super(name);
  }
  public void run()   {
    try {
      for (int i = 0; i != 2; i++) {
        System.out.println(Thread.currentThread().getName() + ": "
            + counter++);
        Thread.sleep(1000);
      }
    }
    catch(InterruptedException e)     {
       e.printStackTrace();
    }
  }
}
```

Multi-Threading

Appendix B.

What is the difference between synchronized and volatile

The value of a **volatile** variable is **not locally cached by a thread** (all reads and writes will go to "main memory" (may produce ‘lost-update’ problem).

Access to the volatile variables is **similar** to code enclosed in a synchronized block, but without locking states.

volatile is **not** suitable for **complex operations** where you need to **prevent simultaneous access** to a variable for the duration of the operation: in such cases, you should use object synchronization (sync methods or statements).