# Threads and Handlers

C.-Z. Yang
http://syslab.cse.yzu.edu.tw/~czyang

# 6.1 multithreading and Multicore Processing

- Android uses processes and thread management models to manage running applications, services, and the other elements of the system.
- When an application does several things at once it is called multithreading.
- Other terms are used for multithreading, such as parallelism and concurrency.
- A multithreaded Android application contains two or more threads.

# Multithreading

- Multithreading enables programmers to write very efficient applications because it allows the use of any idle time that may accrue while other segments of code are being processed.
- Each thread runs as a separate path of execution that is managed by its own stack, the call stack.
- The call stack is used to manage method calling, parameter passing, and storage for a called method's local variables.

# Multitasking

- Multitasking can be subdivided into two categories:
  - Process-based multitasking
  - Thread-based multitasking
- Process-based multitasking is the feature that allows a device to execute two or more programs concurrently.
- In process-based multitasking, an app is the smallest unit of code that can be dispatched by the scheduler.

# Multitasking

- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks at once.

# Threads in Android

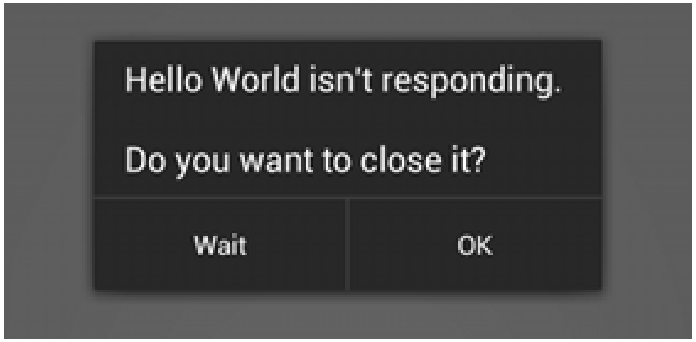
- In Android, the main thread is the UI thread.
  - This single thread is responsible for handling all the UI events.
  - Even a simple single-threaded application can benefit from parallel processing on different cores.
- In Android, users demand responsive applications.
  - Time-intensive operations such as networking should not block the main UI thread.

# ANR problem

- In Android, the system guards against applications that are insufficiently responsive for a period of time.
- If an app has been unresponsive for a considerable period of time, an ANR (Application Not Responding) dialog will appear.
  - No response to an input event (such as key press or screen touch events) within 5 seconds.
  - A BroadcastReceiver hasn't finished executing within 10 seconds.

# ANR problem

- An ANR dialog displayed to the user



Hello World isn't responding.

Do you want to close it?

- In any situation in which your app performs a potentially lengthy operation, you should not perform the work on the UI thread, but instead create a worker thread and do most of the work there.

# Concurrency

- To utilize the maximum potential of the available processing power on multicore devices, applications should be written with concurrency in mind.
- Categories of operations that can be carried out on separate background threads are as follows:
  - Heavy calculations
  - An Object's long initialization
  - Networking
  - Database operations

# 6.2 Main thread and Background threads

- Android UI threads are distinct from background threads.
- When an activity is created, it runs by default in the UI thread of the application.
- All the commands issued by the Android operating system, such as onClick, onCreate, etc., are sent to and processed by this UI thread.

# UI thread

- When writing multithreaded applications in Android, it is a good idea to keep several things in mind about the UI thread:
  - The UI thread should not perform tasks that take longer than a few seconds.
  - The user interface cannot be updated from a background thread.
  - An Android application can be entered from an Activity, Service or a Broadcast Receiver, all of which run on the UI thread.

# UI thread

- Additionally, the Android UI toolkit is not thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread.
- There are simply two rules to Android's single thread model:
  - Do not block the UI thread.
  - Do not access the Android UI toolkit from outside the UI thread.

# Java thread

- Java’s multithreading system is built on the Thread class and its companion interface, Runnable.
  - Both are packaged in java.lang.
- From the main UI thread, programmers can create other threads by instantiating an object of type Thread.

# 6.3 Thread Approaches

- The Thread class encapsulates an object that is runnable.
- Two ways in which a runnable object can be created are:
  - Implement the Runnable interface.
  - Extend the Thread class.
- The start() method must be called to execute a new thread, regardless of the approach.

# 6.3.1 Implementing a Runnable Interface

- The Runnable interface abstracts a unit of executable code.
  - Runnable defines only one method called run().

```
public class MyActivity extends Activity {
@Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView (R.layout.activity_main);

    Runnable myRunnable1 = new MyRunnableClass();
    Thread t1 = new Thread(myRunnable);
    t1.start();
}
```

```
public class MyRunnableClass implement Runnable {
  @Override
  public void run() {
    // operations to be performed on a background thread
  }
}
```

# Example: prime number

- A thread that computes primes larger than a stated value could be written as follows:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
      this.minPrime = minPrime;
    }

    public void run() {
      // compute primes larger than minPrime
       . . .
    }
  }
```

  - Create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

# 6.3.2 Extend the Thread Class

- A Thread class can be also constructed by declaring a class to be a subclass of Thread.

```
public class MyActivity extends Activity {
@Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView (R.layout.activity_main);

      MyThreadClass thread2 = new MyThreadClass();
      thread2.start();
}
```

```
public class MyThreadClass extends Thread {
   @Override
   public void run() {
      // operations to be performed on a background thread
   }
}
```

# Example: prime number

- The code

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

  - Create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

# 6.4 UI Modification and the Handler Class

- The UI thread is the main thread of execution for a given Android application.
- Only the UI thread can modify the user interface.
  - Modification to the UI cannot be directly performed form a background thread.
- The following code will end in an application crash.
  - The layout contains a TextView and a Button.
  - When the user clicks the button, the button onClick() event is handled by the method updateText().

```
public class MyActivity extends Activity {
    private TextView mTextview;
    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView (R.layout.activity_main);

        mTextview = (TextView) findViewById(R.id.textView1);
    }

    public void updateText(View view) {
        new Thread(new Runnable(){
            @Override
            public void run() {
                mTextview.SetText("Just clicked");
            }
         }).start();
    }
}
```

A solution to this problem is to communicate to the UI thread that an update to the TextView needs to be performed. The UI thread can then act on that request.

# Handlers

- A Handler is part of the Android system's framework for managing threads and is designed for inter-thread communication.
  - It combines features from a BlockingQueue and a message listener.
- A Handler object receives messages and runs code to handle the messages.
  - Interaction between an Android thread and the UI thread is accomplished using a UI thread Handler object and posting Messages and Runnable objects to a message queue

# Handler Objects

- A Handler object created on the UI thread exposes a thread-safe message queue on which background threads can asynchronously add either messages or requests for foreground runnables to act on their behalf.
- When a Handler is created for a new thread, it is bound to the message queue of the thread.
  - The Handler will deliver messages and runnables to this message queue and execute them as they are retrieved off the queue.

```
public class MyActivity extends Activity {
   private TextView mTextview;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView (R.layout.activity_main);

      mTextview = (TextView) findViewById(R.id.textView1);
      Button mButton = (Button) findViewById(R.id.button);
      mButton.setOnClickListener (new View.onClickListener(){
         @Override
         public void onClick(View view){
            new Thread() {
               public void run() {
                  // thread work occurs here
                  mHanlder.sendEmptyMessage(0);
               }
            }.start();
         }
      });
   }
```

```
public Handler mHandler = new Handler() {
   public void handleMessage(android.os.Message message) {
      super.handlemessage(message);

      // UPDATE UI COMPONENTS
      textView.setText(textString);
   }
};
```

# Communication with Handlers

- A handler's message queue uses the obtainMessage() method to control communication.

```
Thread backgroundThread = new Thread (new Runnable() {
    @Override
    public void run() {
        // do background work
              . . .

        Message msg = threadHandler.obtainMessage();    // requesting msg token
        // Deliver message to main's message queue
        threadHandler.sendMessage(msg);
     }
});

backgroundThread.start();
```

# Communication with Handlers

- The following code is the Handler definition.
  - The UI component textView and imageView are updated within this method.

```
public Handler threadHandler = new Handler() {
   public void handleMessage(android.os.Message msg) {
      super.handleMessage(msg);

      // UPDATE UI COMPONENTS
      textView.setText(textString);
      imageView.setImageBitmap(bitmap);
   }
};
```

# 6.5 Loopers

- Looper is a class within the Android user interface that can be used in tandem with the Handler class to provide a framework for implementing a concurrency pattern.
- Background threads can push new units of work onto the MessageQueue at any time.
- The UI thread processes the queued units of work one after another
  - If there are no work units on the MessageQueue, it waits until one appears in the queue.

# Looper Class

- A Looper is the mechanism that allows these units of work to be executed or processed sequentially on a single thread.
- A Handler is used to schedule those units of work for execution by pushing them onto a MessageQueue.
  - 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.

# Example code

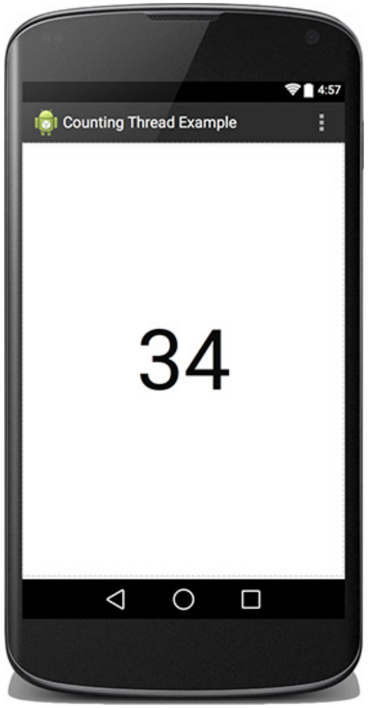
```
class BackgroundThread extends Thread {
    public Handler mHandler;

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

        mHnadler = new Handler(){
            public void handleMessage (Messge msg){
                // process incoming message here
            }
        };
        Looper.loop();
    }
}
```

# Looper

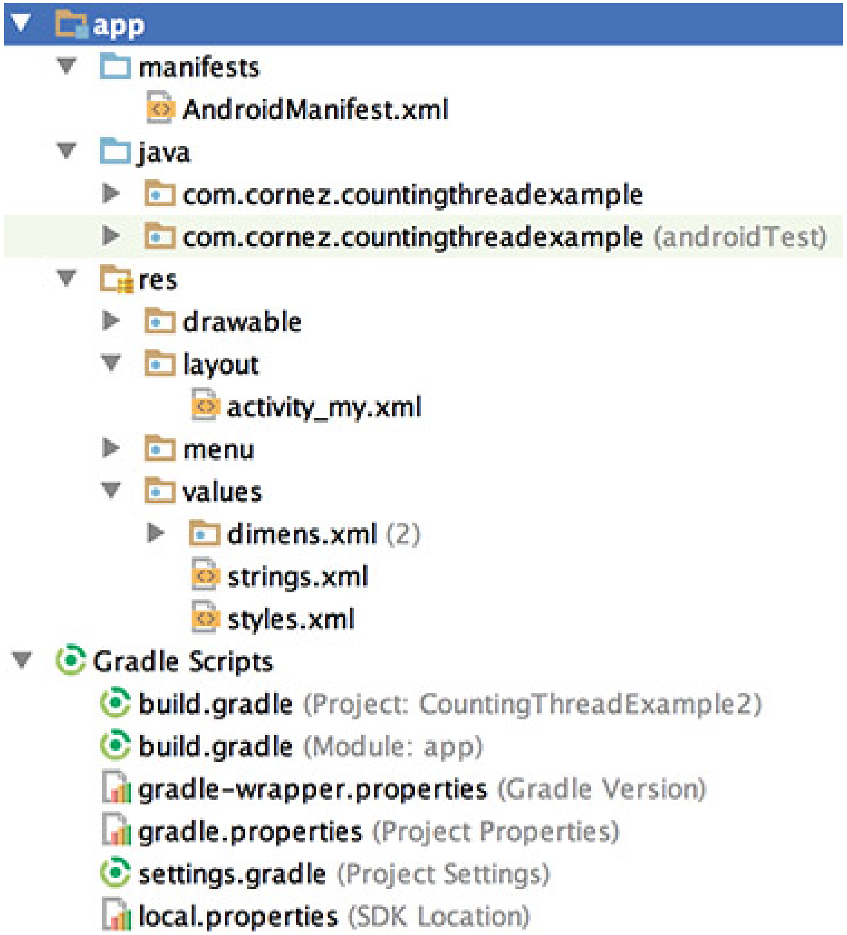
- The default Looper does not need to be initiated.
  - It is an automatic component in an Android application.
- In addition, the default Looper is directly attached to the UI Thread.

# Lab example 6-1: Background Thread and Handler - Counting

- This application features a TextView element that is updated every second. The following figure shows the app 34 seconds after the app has launched.



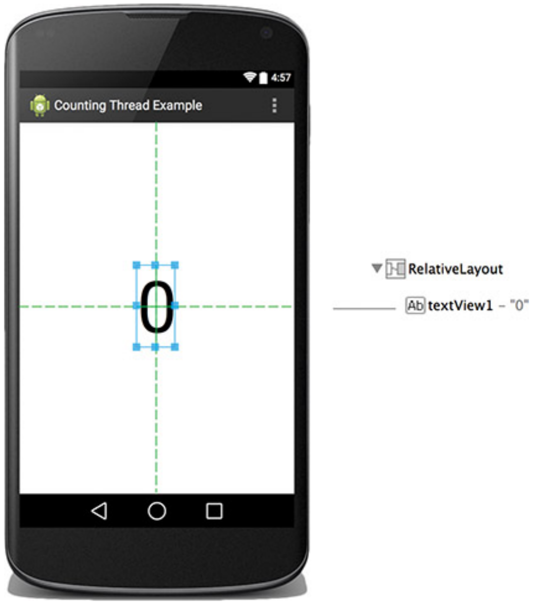
# Project structure



```
▼ app
    ▼ manifests
          AndroidManifest.xml
    ▼ java
        ▶ com.cornez.countingthreadexample
        ▶ com.cornez.countingthreadexample (androidTest)
    ▼ res
        ▶ drawable
        ▼ layout
              activity_my.xml
        ▶ menu
        ▼ values
            ▶ dimens.xml (2)
              strings.xml
              styles.xml
▼ Gradle Scripts
      build.gradle (Project: CountingThreadExample2)
      build.gradle (Module: app)
      gradle-wrapper.properties (Gradle Version)
      gradle.properties (Project Properties)
      settings.gradle (Project Settings)
      local.properties (SDK Location)
```

# activity_my.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MyActivity">


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="34"
        android:id="@+id/textView"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:textSize="100sp" />
</RelativeLayout>
```



# MyActivity.java

```
public class MyActivity extends Activity {

    //DECLARE UI TEXTVIEW AND COUNT OBJECT
    private TextView countTextView;
    private Integer count;

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

        //REFERENCE THE TEXTVIEW UI ELEMENT ON THE LAYOUT
        countTextView = (TextView) findViewById(R.id.textView);

        //INITIALIZE THE COUNTER
        count = 0;

        //CREATE A THREAD AND START IT
        Thread thread = new Thread (countNumbers);
        thread.start();
    }
```

# MyActivity.java

```
//INITIALIZE THE COUNTER TO ZERO EACH TIME THE
//APPLICATION LAUNCHES
@Override
protected void onStart() {
   super.onStart();
   count = 0;
}
```

# Runnable Interface

```
//*************RUNNABLE **************/
  private Runnable countNumbers = new Runnable () {
    private static final int DELAY = 1000;
    public void run() {
      try {
        while (true) {
          count ++;
          Thread.sleep (DELAY);
          threadHandler.sendEmptyMessage(0);
        }
      } catch (InterruptedException e){
        e.printStackTrace();
      }
    }
  };
```

# Handler

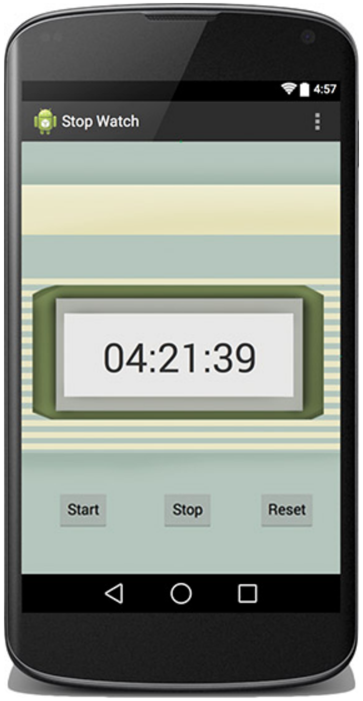
```
//**************HANDLER****************/
public Handler threadHandler = new Handler() {
   public void handleMessage (android.os.Message message){
      countTextView.setText(count.toString());
   }

};
```

# Menu Options

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.my, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

# Lab example 6-2: Digital StopWatch

- This lab example explores the creation of a digital stopwatch.
- This app requires continual updates to the UI timer display once the stopwatch is started.



# User interface

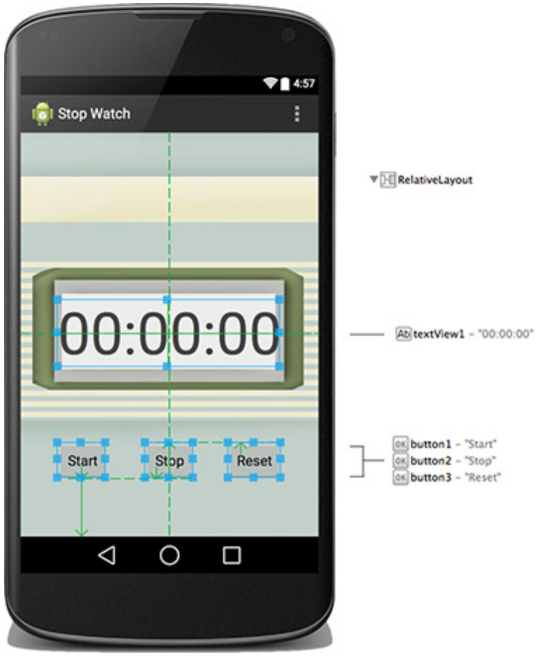
- strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <string name="app_name">Stop Watch</string>
  <string name="hello_world">Hello world!</string>
  <string name="action_settings">Settings</string>

  <string name="timer">00:00:00</string>
  <string name="start_btn">Start</string>
  <string name="stop_btn">Stop</string>
  <string name="reset_btn">Reset</string>

</resources>
```

# User interface

- activity_my.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MyActivity"
    android:background="@drawable/background">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/timer"
        android:id="@+id/textView1"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:textSize="75sp"/>

    <Button
     …
```



# The time data mode (WatchTime.java)

```java
package com.cornez.stopwatch;

public class WatchTime {

    // TIME ELEMENTS
    private long mStartTime;
    private long mTimeUpdate;
    private long mStoredTime;

    public WatchTime() {
        mStartTime = 0L;
        mTimeUpdate = 0L;
        mStoredTime = 0L;
    }

    public void resetWatchTime() {
        mStartTime = 0L;
        mStoredTime = 0L;
        mTimeUpdate = 0L;
    }
 public void setStartTime(long startTime){
        mStartTime = startTime;
    }
    public long getStartTime(){
        return mStartTime;
    }
    public void setTimeUpdate(long timeUpdate){
        mTimeUpdate = timeUpdate;
    }
    public long getTimeUpdate(){
        return mTimeUpdate;
    }
    public void addStoredTime(long timeInMilliseconds){
        mStoredTime += timeInMilliseconds;
    }
    public long getStoredTime(){
        return mStoredTime;
    }
}
```

# MyActivity.java

```
public class MyActivity extends Activity {

    // UI ELEMENTS: BUTTONS WILL TOGGLE IN VISIBILITY
    private TextView timeDisplay;
    private Button startBtn;
    private Button stopBtn;
    private Button resetBtn;

    // TIME ELEMENTS
    private WatchTime watchTime;
    private long timeInMilliseconds = 0L;

    // THE HANDLER FOR THE THREAD ELEMENT
    //private Handler handler = new Handler();
    private Handler mHandler;
```

# MyActivity.java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // TASK 1: ACTIVATE THE ACTIVITY AND THE LAYOUT
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_my);

   // TASK 2: CREATE REFERENCES TO UI COMPONENTS
   timeDisplay = (TextView) findViewById(R.id.textView1);
   startBtn = (Button) findViewById(R.id.button1);
   stopBtn = (Button) findViewById(R.id.button2);
   resetBtn = (Button) findViewById(R.id.button3);

   // TASK 3: HIDE THE STOP BUTTON
   stopBtn.setEnabled(false);

   // TASK 4: INSTANTIATE THE OBJECT THAT MODELS THE STOPWATCH TIME
   watchTime = new WatchTime();

   //TASK 5: INSTANTIATE A HANDLER TO RUN ON THE UI THREAD
   mHandler = new Handler();
}
```

By initiating the Handler within onCreate(), mHandler is bound to the UI thread and its default Looper.

# MyActivity.java

```
public void startTimer(View view) {
   // TASK 1: SET THE START BUTTON TO INVISIBLE
   //        AND THE STOP BUTTON TO VISIBLE
   stopBtn.setEnabled(true);
   startBtn.setEnabled(false);
   resetBtn.setEnabled(false);

   // TASK 2: SET THE START TIME AND CALL THE CUSTOM HANDLER
   watchTime.setStartTime(SystemClock.uptimeMillis());
   mHandler.postDelayed(updateTimerRunnable, 20);
}
```

postDelayed(Runnable r, long delayMillis)

Causes the Runnable r to be added to the message queue, to be run after the specified amount of time elapses.

# MyActivity.java

```
private Runnable updateTimerRunnable = new Runnable() {
    public void run() {
      // TASK 1: COMPUTE THE TIME DIFFERENCE
      timeInMilliseconds = SystemClock.uptimeMillis() - watchTime.getStartTime();
      watchTime.setTimeUpdate(watchTime.getStoredTime() + timeInMilliseconds);
      int time = (int) (watchTime.getTimeUpdate() / 1000);

      // TASK 2: COMPUTE MINUTES, SECONDS, AND MILLISECONDS
      int minutes = time / 60;
      int seconds = time % 60;
      int milliseconds = (int) (watchTime.getTimeUpdate() % 100);

      // TASK 3: DISPLAY THE TIME IN THE TEXTVIEW
      timeDisplay.setText(String.format("%02d", minutes) + ":"
          + String.format("%02d", seconds) + ":"
          + String.format("%02d", milliseconds));

      // TASK 4: SPECIFY NO TIME LAPSE BETWEEN POSTING
      mHandler.postDelayed(this, 0);
    }
  };
```

# MyActivity.java

```
public void stopTimer(View view) {
    // TASK 1: DISABLE THE START BUTTON
    //         AND ENABLE THE STOP BUTTON
    stopBtn.setEnabled(false);
    startBtn.setEnabled(true);
    resetBtn.setEnabled(true);

    // TASK 2: UPDATE THE TIME SWAP VALUE AND CALL THE HANDLER
    watchTime.addStoredTime(timeInMilliseconds);
    mHandler.removeCallbacks(updateTimerRunnable);
}
```

removeCallbacks(Runnable r)

Remove any pending posts of Runnable r that are in the message queue.

# MyActivity.java

```
public void resetTimer(View view) {
   watchTime.resetWatchTime();
   timeInMilliseconds = 0L;

   int minutes = 0;
   int seconds = 0;
   int milliseconds = 0;

   // TASK 3: DISPLAY THE TIME IN THE TEXTVIEW
   timeDisplay.setText(String.format("%02d", minutes) + ":"
         + String.format("%02d", seconds) + ":"
         + String.format("%02d", milliseconds));
}
```

# MyActivity.java

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu.
        getMenuInflater().inflate(R.menu.my, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

# Concluding Remarks