developers

Updating the UI from a Timer

Background: While developing my first useful (though small) application for Android, which was a port of an existing utility I use when podcasting, I needed a way of updating a clock displayed on the UI at regular intervals, but in a lightweight and CPU efficient way.

Problem: In the original application I used java.util.Timer to update the clock, but that class is not such a good choice on Android. Using a Timer introduces a new thread into the application for a relatively minor reason. Thinking in terms of mobile applications often means re-considering choices that you might make differently for a desktop application with relatively richer resources at its disposal. We would like to find a more efficient way of updating that clock.

The Application: The original application is a Java Swing and SE application. It is like a stopwatch with a lap timer that we use when recording podcasts; when you start the recording, you start the stopwatch. Then for every mistake that someone makes, you hit the flub button. At the end you can save out the bookmarked mistakes which can be loaded into the wonderful Audacity audio editor as a labels track. You can then see where all of the mistakes are in the recording and edit them out.

The article describing it is: http://www.developer.com/java/ent/print.php/3589961

In the original version, the timer code looked like this:

```
class UpdateTimeTask extends TimerTask {
   public void run() {
        long millis = System.currentTimeMillis() - startTime;
        int seconds = (int) (millis / 1000);
        int minutes = seconds / 60;
        seconds     = seconds % 60;

        timeLabel.setText(String.format("%d:%02d", minutes, seconds));
   }
}
```

And in the event listener to start this update, the following Timer() instance is used:

```
if(startTime == 0L) {
   startTime = evt.getWhen();
   timer = new Timer();
   timer.schedule(new UpdateTimeTask(), 100, 200);
}
```

In particular, note the 100, 200 parameters. The first parameter means wait 100 ms before running the clock update task the first time. The second means repeat every 200ms after that, until stopped. 200 ms should not be too noticeable if the second resolution happens to fall close to or on the update. If the resolution was a second, you could find the clock sometimes not updating for close to 2 seconds, or possibly skipping a second in the counting, it would look odd).

When I ported the application to use the Android SDKs, this code actually compiled in Eclipse, but failed with a runtime error because the Timer() class was not available at runtime (fortunately, this was easy to figure out from the error messages). On a related note, the String.format method was also not available, so the eventual solution uses a quick hack to format the seconds nicely as you will see.

Fortunately, the role of Timer can be replaced by the android.os.Handler class, with a few tweaks. To set it up from an event listener:

```
private Handler mHandler = new Handler();

...
```

```
OnClickListener mStartListener = new OnClickListener() {
   public void onClick(View v) {
        if (mStartTime == 0L) {
             mStartTime = System.currentTimeMillis();
             mHandler.removeCallbacks(mUpdateTimeTask);
             mHandler.postDelayed(mUpdateTimeTask, 100);
        }
   }
};
```

A couple of things to take note of here. First, the event doesn't have a .getWhen() method on it, which we handily used to set the start time for the timer. Instead, we grab the System.currentTimeMillis(). Also, the Handler.postDelayed() method only takes one time parameter, it doesn't have a "repeating" field. In this case we are saying to the Handler "call mUpdateTimeTask() after 100ms", a sort of fire and forget one time shot. We also remove any existing callbacks to the handler before adding the new handler, to make absolutely sure we don't get more callback events than we want.

But we want it to repeat, until we tell it to stop. To do this, just put another postDelayed at the tail of the mUpdateTimeTask run() method. Note also that Handler requires an implementation of Runnable, so we change mUpdateTimeTask to implement that rather than extending TimerTask. The new clock updater, with all these changes, looks like this:

```
private Runnable mUpdateTimeTask = new Runnable() {
   public void run() {
        final long start = mStartTime;
        long millis = SystemClock.uptimeMillis() - start;
        int seconds = (int) (millis / 1000);
        int minutes = seconds / 60;
        seconds     = seconds % 60;

        if (seconds < 10) {
            mTimeLabel.setText("" + minutes + ":0" + seconds);
        } else {
            mTimeLabel.setText("" + minutes + ":" + seconds);
        }

        mHandler.postAtTime(this,
                start + (((minutes * 60) + seconds + 1) * 1000));
   }
};
```

and can be defined as a class member field.

The if statement is just a way to make sure the label is set to 10:06 instead of 10:6 when the seconds modulo 60 are less than 10 (hopefully String.format() will eventually be available). At the end of the clock update, the task sets up another call to itself from the Handler, but instead of a hand-wavy 200ms before the update, we can schedule it to happen at a particular wall-clock time — the line: start + (((minutes * 60) + seconds + 1) * 1000) does this.

All we need now is a way to stop the timer when the stop button is pressed. Another button listener defined like this:

```
OnClickListener mStopListener = new OnClickListener() {
   public void onClick(View v) {
        mHandler.removeCallbacks(mUpdateTimeTask);
   }
};
```

will make sure that the next callback is removed when the stop button is pressed, thus interrupting the tail iteration.

Handler is actually a better choice than Timer for another reason too. The Handler runs the update code as a part of your main thread, avoiding the overhead of a second thread and also making for easy access to the View hierarchy used for the user interface. Just remember to keep such tasks small and light to avoid slowing down the user experience.