Android Programming
Lecture 20:
Threads, Part II

11/16/2011
In this example, GUI thread doesn’t count – button is responsive; I can trigger interleaved counting threads

```java
public class HandlerExampleActivity extends Activity implements View.OnClickListener {
    Button button;
    TextView textView;
    Handler guiHandler;
    Runnable countingRunnable;
    Runnable updateStartRunnable;
    Runnable updateDoneRunnable;

    public void onClick(View view) {
        // start the 2nd thread counting
        Thread countingThread = new Thread(null, countingRunnable, "BackgroundCounter");
        countingThread.start();
    }

    public void count(String name) {
        for (int j = 0; j < 10000000; j++) {
            if ((j % 100000) == 0) {
                Log.v(name, "TICK " + j);
            }
        }
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        button = (Button) findViewById(R.id.button);
        textView = (TextView) findViewById(R.id.text_view);
        button.setOnClickListener(this);

        // handler for GUI thread
        guiHandler = new Handler();

        // code that GUI thread should run (requested by 2nd thread)
        updateStartRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Start");
            }
        };

        updateDoneRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Done");
            }
        };

        // actual 2nd thread work, and request to update GUI
        countingRunnable = new Runnable() {
            public void run() {
                guiHandler.post(updateStartRunnable);
                count("CounterThread");
                guiHandler.post(updateDoneRunnable);
            }
        };
    }

```

Start counting...
Other Uses of Handler

- Handler also supports a `postDelayed(Runnable runnable, long delayInMillis)` method
  - Allows delay before executes

```java
// read preferences
String preferenceName = getString(R.string.user_settings);
SharedPreferences sp = getSharedPreferences(preferenceName, 0);
if (sp.contains(getString(R.string.refreshRateSetting)))
{
    String refreshRatePref = getString(R.string.refreshRateSetting);
    int refreshRateIndex = sp.getInt(refreshRatePref, 0);
    int[] refreshRateValues = getResources().getIntArray(R.array.refreshRateValues);
    delay = refreshRateValues[refreshRateIndex];
}
if (sp.contains(getString(R.string.currentBusID)))
{
    // if this is true, they have used the tool at least once
    // so load their most recently viewed bus and start updating
    String currentBusPref = getString(R.string.currentBusID);
    currentBusID = sp.getInt(currentBusPref, 0);
    mHandler.postDelayed(mUpdateTimeTask, 100);
}
```

Shuttle bus app reads bus id and delay from saved SharedPreferences, then starts a loop (below) of updates at the designated rate loops by spawning itself again
CityApp Example

• In the CityApp, it seems reasonable that we could push the network (web service) and parsing components into a separate thread
• Should make GUI much more responsive

• However, for the CityApp, once the network query has been started, we don’t necessarily need to do anything else as a user
  – It would be nice to have some feedback about progress though
    • Repeated use of post method in Handler - Fetching data, Parsing data, ... messages?????
Runnables

- *Runnable* as used in the example are pretty static
  - we can’t parameterize them in meaningful ways
    - Could arguably create classes that implement Runnable interface, and the classes could be parameterized

- On next slide, I take the Runnables from a few slides back that updated the GUI with “Start”/”Done” messages and make them parameterized so that I can write any message
Parameterized Runnable Example

Before

```java
public class HandlerExampleActivity extends Activity implements View.OnClickListener {
    Button button;
    TextView textView;
    Handler guiHandler;
    Runnable countingRunnable;
    Runnable updateStartRunnable;
    Runnable updateDoneRunnable;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        button = (Button) findViewById(R.id.button);
        textView = (TextView) findViewById(R.id.text_view);
        button.setOnClickListener(this);

        // handler for GUI thread
        guiHandler = new Handler();

        // code that GUI thread should run (requested by 2nd thread)
        updateStartRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Start");
            }
        };

        updateDoneRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Done");
            }
        };

        // actual 2nd thread work, and request to update GUI
        countingRunnable = new Runnable() {
            public void run() {
                guiHandler.post(updateStartRunnable);
                guiHandler.post(updateDoneRunnable);
            }
        };
    }
}
```

After

```java
public class MessageRunnable implements Runnable {
    String message;
    TextView textView;

    public MessageRunnable(String messageArg, TextView textViewArg) {
        message = messageArg;
        textView = textViewArg;
    }

    public void run() {
        textView.setText(textView.getText() + "\n" + message);
    }
}

public class HandlerExampleActivity extends Activity implements View.OnClickListener {
    Button button;
    TextView textView;
    Handler guiHandler;
    Runnable countingRunnable;
    Runnable updateStartRunnable;
    Runnable updateDoneRunnable;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        button = (Button) findViewById(R.id.button);
        textView = (TextView) findViewById(R.id.text_view);
        button.setOnClickListener(this);

        // handler for GUI thread
        guiHandler = new Handler();

        // code that GUI thread should run (requested by 2nd thread)
        updateStartRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Start");
            }
        };

        updateDoneRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Done");
            }
        };

        // actual 2nd thread work, and request to update GUI
        countingRunnable = new Runnable() {
            public void run() {
                guiHandler.post(new MessageRunnable("CounterThread Start", textView));
                guiHandler.post(new MessageRunnable("CounterThread Done", textView));
            }
        };
    }
}
```
AsyncTask

• Android supports a fairly simple mechanism that simplifies/hides the process of working directly with threads, the AsyncTask class (asynchronous task)

• One extends the AsyncTask class and implements several methods
  – doInBackground – do the actual work (required)
  – onProgressUpdate – report progress (optional)
  – onPostExecute – report results (optional)
  – onPreExecute – initialize (optional)
AsyncTasks

• Put your seatbelt on...

• AsyncTasks are conceptually straight-forward, but syntactically challenging
  – We’ll start with an example and think of it conceptually
Using AsyncTask

Let’s look at an example: counting to ten million again.
Varargs

AsyncTask makes use of varargs – variable number of arguments to a function

C/C++ printf:

```c
printf(“Hello!”);
printf(“Hello %d”, x);
printf(“Hello %d %d”, x, y)
```

Same function, takes arbitrary number of parameters as final argument
Varargs

In Java, write function header as follows:

```java
void f(type... parameter)
{... code ...}
```

meaning function \( f \) can take an arbitrary number of parameters of type \( \text{type} \) (\textit{but only that type, which is different from C/C++})

Can be passed as a comma separated series of values or as an array

Only allowed for last parameter
AsyncTask

• Remember: One extends the AsyncTask class and implements several methods
  – doInBackground – do the actual work (required)
  – onProgressUpdate – report progress (optional)
  – onPostExecute – report results (optional)
  – onPreExecute – initialize (optional)
AsyncTask

• In defining class, use extends AsyncTask providing as comma separated class parameters the following in this order:
  – Single class type of input parameter(s) to be used in doInBackground function
  – Single class type used to report progress
    • Usually Integer
  – Single class type used as result value from task, which must match return value type from doInBackground

• If your function doesn’t need parameters, indicate a type of Void (not Void…)

• class MyTask extends AsyncTask<String, Integer, Integer> means can use String parameters for doInBackground and progress and completion reporting will use ints (Integer class)

• Write AsyncTask extension as an inner class of Activity class so has access to Activity GUI components
Implementing AsyncTask

• Write protected ReturnType
doInBackground(ParameterType... params)
  – Not thread-safe (don’t change GUI directly)
  – Must return a value of ReturnType
  – Can process parameters using an array or iteration
    • Array:
      – for (j = 0; j < params.length; j++) { ... do work ... }
    • Iteration:
      – for (String x: params) { }
  – If want to update progress, call publishProgress
    method with appropriate arguments
    • Android passes this off to your onProgressUpdate method
Implementing AsyncTask

• Write `protected void onProgressUpdate(Type... progress)`
  – Is thread-safe (can change GUI directly)
    • Update progress bar???

• Write `protected void onPostExecute(Type... result)`
  – Is thread-safe (can change GUI directly)
Using AsyncTask

• Declare and construct an instance of your AsyncTask subclass

• Call the *execute* method on that instance, sending in a comma separated list of values of the type that *doInBackground* takes as parameters
  – If no parameters to be sent, use *(Void[]) null*

• A few last rules:
  – The task must be allocated, and *executed* on the main GUI thread
  – Any allocated task can only be run once
  – Internal methods should not be called directly
Using AsyncTask

- Let’s look at an example: counting to ten million again.
Using AsyncTask with Progress Updates

• Added onProgressUpdate method and publishProgress call

```java
private class CountingTask extends AsyncTask<Integer, Integer, Void> {
    protected void onPostExecute(Void result) {
        textView.setText(textView.getText() + "\n" + "Counting Done!");
    }

    protected void onProgressUpdate(Integer... update) {
        textView.setText(textView.getText() + "\n" + update[0]);
    }

    protected Void doInBackground(Integer... parameters) {
        int upperLimit = parameters[0];
        int checkpoint = parameters[1];

        for (int i = 0; i < upperLimit; i = i + 1) {
            if ((i % checkpoint) == 0) {
                Log.v("CountingTask", ""+i);
                publishProgress(i);
            }
        }

        return (Void) null;
    }
}
```

Multiple counting tasks executing (2\textsuperscript{nd}, 3\textsuperscript{rd} interleaved)