Programming of Mobile Services, Spring 2012

HI1017

Lecturer: Anders Lindström,
anders.lindstrom@sth.kth.se

Lecture 5
Today's topics

- Concurrent programming
  - Threads
  - Handler
  - ASyncTask
- Services
- File system basics

Concurrent programming

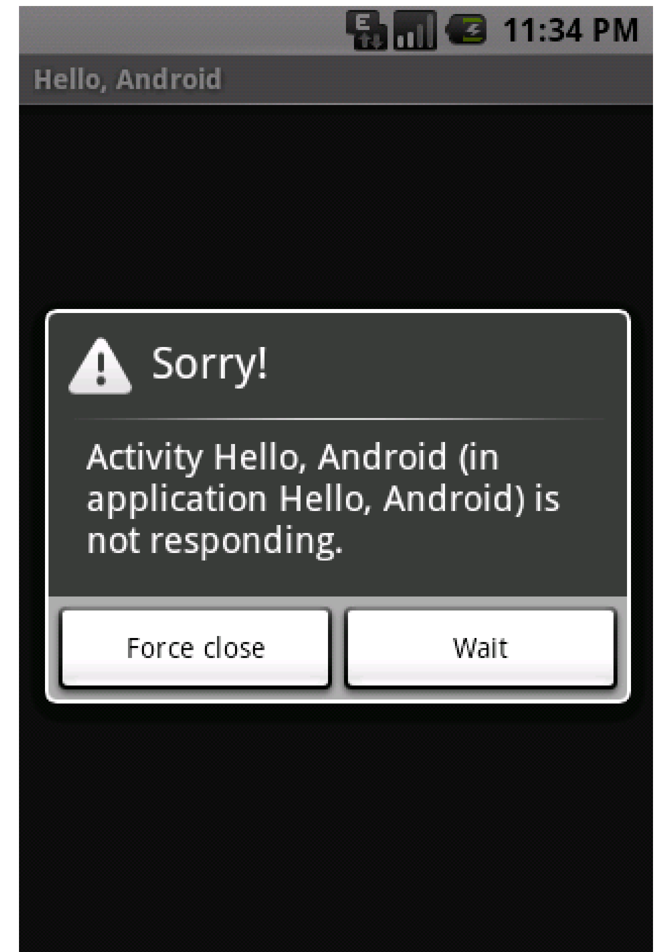
- Android applications normally run entirely on a single thread (the “main thread” or “UI thread”)
- The main thread handles all user input, executing code in event listners, rendering and life cycle call backs
- Code running in the mainthread should do as little work as possible to keep the application and it’s UI responsive

Concurrent programming

- Time consuming tasks: Spawn a new thread to do the work in the background
- Examples: Long calculations; network, file and database operations; game animations; …

Concurrent programming

- The Android system guards against non responsiveness
- Application Not Responding (ANR) dialog
- No response to an input event (e.g. key press, screen touch) within 5 seconds
- A BroadcastReceiver that hasn't finished executing within 10 seconds



Concurrent programming

- Create a new thread:
  - extend java.lang.Thread
  - override public void run()
  - call thread.start()
- Or:
  - implement interface java.lang.Runnable
  - override public void run()
  - Thread t = new Thread(new MyRunnable);
  - t.start();

Concurrent programming

Concurrency issue: Updating UI, or other application components, when the worker thread finishes (or during the execution)

- Manipulating UI components from another thread than the main thread might cause corrupted state (race conditions)
- Android only allows UI updates on the UI thread
- CalledFromWrongThreadException thrown otherwise

Concurrent programming

Solutions

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)
  View.postDelayed(Runnable, long)
- Handler
- AsyncTask

- http://developer.android.com/resources/articles/painless-threading.html

Example: post(Runnable)

- ```
  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();
  }
  ```

- Kind of messy…
  ```

android.os.Handler

- Use a android.os.Handler to post a UI update task from a worker thread
- The constructor associates the handler with the queue for the current thread
- The Handler allows you to send and process Message and *Runnable* objects to a threads message queue (from another thread)
- The Runnable object defines the task to execute on the target thread

Concurrent programming, idiom

```
public class WorkerActivity extends Activity {
    private Handler handler;

    public void onCreate(Bundle savedInstanceState) {
        . . .
        // A handler associated with UI thread
        handler = new Handler();
    }

    public void doWorkInBackground() {
        Thread worker = new Worker(); // Extends thread
        worker.start();
    }
    . . .
}
```

Concurrent programming, idiom *cont*.

```
public class WorkerActivity extends Activity {
   . . .
   // The background task
   private class Worker extends Thread {
         public void run() {
                  // Execute the time consuming task...
                  // All work and no play makes Jack a dull boy...

                  // The work is done, post back to UI-thread
                  handler.post(new UpdateUIOnWorkerFinished());
         }
    }

    // Runnable to post to UI thread
    private class UpdateUIOnWorkerFinished implements Runnable {
         public void run() {
                  // The work is done - update the UI
                  textView.setText(...);
                  . . .
         }
    }
}
```

Concurrent programming, life cycle

- Make sure (continuously long running) background threads are paused and restarted in the life cycle methods

- ```
  public class MyActivity extends Activity {
      private Animator animator;
      . . .
      public void onPause() {
          super.onPause();
          animator.stopAnimation();
          . . .
      }
      public void onResume() {
          super.onResume();
          animator.startAnimation();
          . . .
      }
      . . .
  }
  ```
  ```

AsyncTask

- Performs background operations and publish results on the main thread without the developer having to manipulate threads and/or handlers
- Extend AsyncTask and override the appropriate call back methods, e.g. doInBackground() and onPostExecute()
- Create instance and call task.execute(), to start the task, on the main thread

AsyncTask

When an asynchronous task is executed, the task goes through 4 steps:

- onPreExecute(), invoked on the main thread immediately after the task is started

- doInBackground(Params...), invoked on a background thread immediately after onPreExecute() finishes

- onProgressUpdate(Progress...), invoked on the main thread after a call to publishProgress(...)

- onPostExecute(Result), invoked *on the main thread* after the background computation finishes

Extending ASyncTask

- `android.os.AsyncTask<Params, Progress, Result>`

- Sub-classing, example:
```
private class BackgroundTask
    extends AsyncTask<Long, Void, String>
```

- AsyncTask's generic type parameters:
  - 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

- To mark a type as unused, simply use the type Void

ASyncTask, idiom

```
public class WorkerActivity extends Activity {
    . . .

    private void startWorkInBackground() {
        . . .
        BackgroundTask task = new BackgroundTask();
        task.execute(data); // Called on main thread
    }

    private class PrimeTask extends AsyncTask <Long,Void,String>{
        . . .
    }
}
```

ASyncTask, idiom *cont*.

```
public class WorkerActivity extends Activity {
    . . .
    private class PrimeTask extends AsyncTask <Long, Void, String> {

        protected String doInBackground(Long... limit) {
                long lim = limit[0]; // The argument is an array
                long n = Prime.calculateNumberOfPrimes(lim);
                String output = "Number of primes <= "+lim+" is "+n;
                return output;
        }

        protected void onPostExecute(String output) {
                textOutput.setText(output);
                . . .
        }
    }
}
```

Concurrent programming

- Examples
  - WorkerThreadExample1.zip: Thread + Handler
  - WorkerThreadExample2.zip: Anonymous inner classes + ProgressDialog
  - ASyncTaskExample.zip
- http://developer.android.com/resources/articles/painless-threading.html
- http://developer.android.com/guide/appendix/faq/commontasks.html#threading
- http://developer.android.com/guide/practices/design/responsiveness.html

Services

- A Service is a basic application component, like Activity, *without a user interface*
- A Service can be used for/as
  - A background process, performing some lengthy operation
  - An interface for a remote object, called from your application
- When the OS need to kill applications or application components to save resources, Services has high priority, only second to foreground Activities

Services

- A service runs in the main thread of its hosting process(!)
  - any CPU intensive work or blocking operations should be performed by spawning a worker thread
- The IntentService has its own thread where it schedules the work to be done

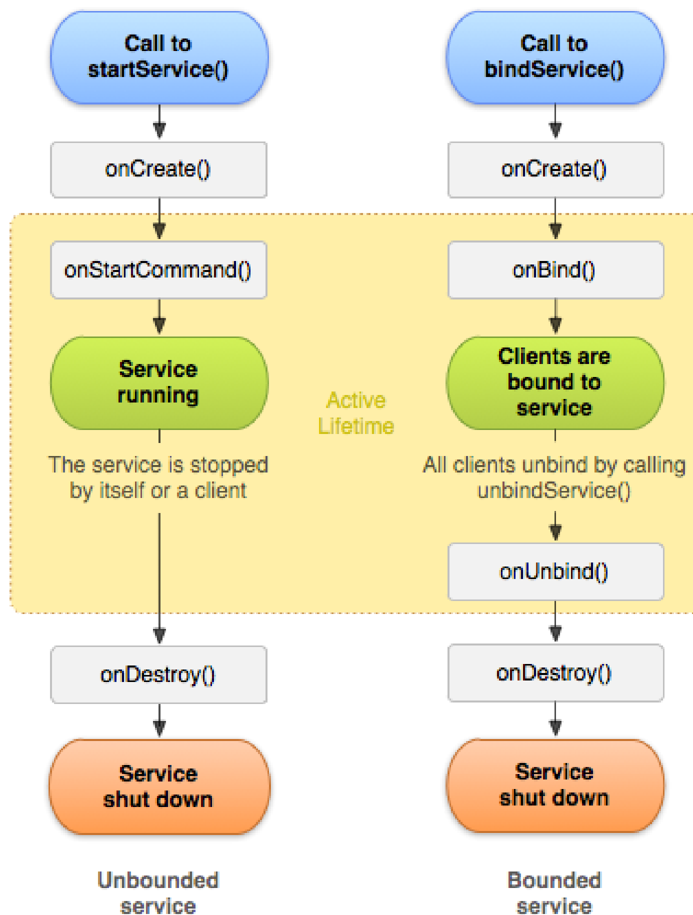
- Each service class must have a corresponding <service> declaration in its application's manifest file

Services

Consider using a Service when the application

- performs a lengthy or intensive processing, not requiring user interaction
- performs certain tasks at regular intervals, e.g. downloading updates of some content
- performs lengthy operations that shouldn't be canceled if the application exits, e.g. downloading a large file
- needs to provide data or information services to other applications (without an user interface)

Services, life cycle



A service can essentially take two forms

- Started (or unbounded) - when an application component starts it by calling startService()
- Bound - when an application component binds to it by calling bindService()

Service Lifecycle

- Started via startService(Intent) :
  - if needed, created and onCreate() called
  - onStartCommand(Intent, …) called
  - runs until Context.stopService() or stopSelf() is called

- Started via
  bindService (Intent, ServiceConnection, flags) to obtain a persistent connection to a service:
  - if needed, created and onCreate() called
  - the service will run until all connections are disconnected

Service, life cycle call backs

- onStartCommand() - called when another component, such as an activity, requests the service to be started, by calling startService()
- Started this way, the Service can run for ever; stop the service when its work is done, by calling stopSelf() or stopService()

- onBind() – called when another component wants to bind with the service (such as to perform RPC), by calling bindService()
  Returns an Ibinder, an interfce that clients use to communicate with the service
- Mandatory to implement; if you don't want to allow binding, return null

Service, life cycle call backs

- onCreate() – called when the service is first created, to perform one-time setup procedures
- onDestroy()- called when the service is no longer used and is being destroyed
  Clean up any resources such as threads, registered listeners, receivers, etc

Service example (unbound)

```
public class ExampleService extends Service {

    @Override
    public void onStartCommand(Intent intent, int flags,int id) {
        doSomeWorkInBackground();
        return START_STICKY;
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        // Stop threads, release allocated resources
        stopWorkingInBackground();
    }
    @Override
    public IBinder onBind(Intent intent) {
            return null; // Clients may not bind to this service
    }
    . . .
```

Service example, *cont*

```
public class ExampleService extends Service {
   . . .
   private void doSomeWorkInBackground() {
      if(timer == null) {
      timer = new Timer();
      timer.scheduleAtFixedRate(new Task(), 0, INTERVAL);
      }
   }

   private class Task extends TimerTask {
     public void run() {
      // Do whatever you want to do every "INTERVAL"
      . . .
     }
   }

   private void stopWorkingInBackground() {
     if (timer != null) {
      timer.cancel();
     }
   }
}
```

Service example, the “manager”

```
public class ManageServiceActivity extends Activity {
   . . .

   private void startExampleService() {
        Intent intent = new Intent(
              this, ExampleService.class);
        startService(intent);
   }

   private void stopExampleService() {
        Intent intent = new Intent(
              this, ExampleService.class);
        stopService(intent);
   }
   . . .
}
```

IntentService

- IntentService creates a worker thread to handle all start requests, delivered to onStartCommand(), one at a time

- *Implement onHandleIntent() to do the work provided by the client*

- The IntentService is stopped after all start requests have been handled, no need to call stopSelf()
- Default implementations of onStartCommand() and onBind()

IntentService

```java
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns,
   * IntentService stops the service, as appropriate.
  */
  @Override
  protected void onHandleIntent(Intent intent) {
      // This is were we execute the task...
      // All work and no play makes Jack a dull boy...
      // ...
  }
}
```

Bound Service

- When an Activity is bound to a Service it maintains a reference to the Service instance

```
public class MyBoundService extends Service {

   private final IBinder binder = new MyBinder();

   @Override
   public IBinder onBind(Intent intent) {
      return binder;
   }

   public class MyBinder extends Binder {
      MyBoundService getService() {
         return MyBoundService.this;
      }
   }
}
```

Bound Service

- The Actvity(ies) need to implement a ServiceConnection

```
public class MyActivity extends Activity {
   private MyBoundService serviceBinder;

   @Override
   public void onCreate(Bundle icicle) { ...
      // Bind to the service
      Intent bindIntent = new Intent(MyActivity.this, MyBoundService.class);
      bindService(bindIntent, mConnection, Context.BIND_AUTO_CREATE);
   }

   // Handles the connection between the service and activity
   private ServiceConnection mConnection = new ServiceConnection() {
      public void onServiceConnected(ComponentName className, IBinder service) {
         // Called when the connection is made.
         serviceBinder = ((MyBoundService.MyBinder) service).getService();
      }

      public void onServiceDisconnected(ComponentName className) {
         // Received when the service unexpectedly disconnects.
         serviceBinder = null;
      }
   };
   ...
```

More on Service

- http://developer.android.com/guide/topics/fundamentals/services.html
- http://developer.android.com/guide/topics/fundamentals/bound-services.html

Background work in Thread, AsyncTask or Service?

	Thread	AsyncTask	Service	IntentService
When to use ?	- Long task in general. - For tasks in parallel use Multiple threads (traditional mechanisms)	- Long task having to communicate with main thread.	Task with no UI, but shouldn't be too long. Use threads within service for long tasks.	- Long task usually with no communication to main thread - Limitation: tasks executed sequentially
Trigger	Thread start() method	Call to method execute()	Call to method onStartService()	Intent
Triggered From (thread)	Any Thread	Main Thread	Any thread	Main Thread
Runs On	Its own thread	Worker thread. However, Main thread methods may be invoked in between to publish progress.	Main Thread	Separate worker thread is automatically spawned

Adapted from: http://techtej.blogspot.com/2011/03/android-thread-constructspart-4.html

File basics, internal storage

- By default, files saved to the internal storage are private to the application
- When the user uninstalls the application, these files are removed ☺
- FileOutputStream fos = openFileOutput(
      fileName, Context.MODE_PRIVATE);
  fos.write(string.getBytes());
  fos.close();

File basics, internal storage

- MODE_PRIVATE, MODE_APPEND, MODE_WORLD_READABLE, MODE_WORLD_WRITEABLE
- getFilesDir() Gets the absolute path to the filesystem directory where your internal files are saved
- getDir() Creates (or opens an existing) directory within your internal storage space
- deleteFile() Deletes a file saved on the internal storage
- fileList() Returns an array of file names

File basics, internal storage

- getFileInput/getFileOutput returns a primitive stream object
- Use the classes in java.io to add filter or wrapper streams
- Example, writing to a text file:

```
PrintWriter pw = new PrintWriter(
    getFileOutput(. . .));
pw.println(info);
```

Writing to a text file

```
PrintWriter writer = null;
try {
      OutputStream os = this.openFileOutput(
               fileName, Context.MODE_PRIVATE);
      writer = new PrintWriter(os);

      writer.println("Lots of information to remember");
      . . .
}
catch(IOException ioe) {
   showToast("Error while writing to file");
}
finally {
   if(writer != null) writer.close();
}
```

Reading from a text file

```
BufferedReader reader = null;
try {
    InputStream is = this.openFileInput(fileName);
    reader = new BufferedReader(new InputStreamReader(is));

    String line = reader.readLine();
    while(line != null) {
        . . . // Process the information
        line = reader.readLine();
    }
}
catch(IOException ioe) {
    showToast("Error while writing to file");
}
finally {
    try {
        if(reader != null) reader.close();
    }
    catch(IOException ioe) {}
}
```