Mobile Application (Design and) Development

10th class

Prof. Stephen Intille
s.intille@neu.edu

Q&A

Today

- Overview of BroadcastReceivers, Services and Wakelock
- Demo
- Fahd's tips and tricks
  - A few things Fahd thinks he would have liked to have known when just starting out with Android
- Design paper presentation
  - Simple and Usable Displace Section Presenter: Sapna Krishnan

Schedule

- **Tomorrow: design assignment 3 due**
  - Preliminary project description and paper prototype
  - Make sure you include enough detail so it will be clear to someone who hasn't seen it how it works
- Thursday: location and sensing
- **Sunday: Programming assignment 3 due**

Building blocks of an app

- Activities
- **Services**
- Content providers
- Intents
- **Broadcast receivers**
- Widgets
- Notifications

Building blocks of an app

- In theory, by decoupling dependencies between app components you can share and interchange individual pieces

- In practice, may be a little tricky to do

BroadcastReceiver

- Component that responds to system-wide broadcast announcements.
- Example system broadcasts: screen has turned off, the battery is low, user is present using phone, or a picture was captured.

BroadcastReceiver

- Applications can initiate broadcasts—e.g., to let other applications know that some data has been downloaded to the device and is available for them to use.
- Don't display a UI, but can create a status bar notification to alert the user when a broadcast event occurs.

BroadcastReceiver

- Usually, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.
  - Important: you must complete tasks in a BroadcastReceiver in <10s. If you have a task that will take longer, you **must** start a new thread to avoid app assassin OS.

Services

- Handles operation and functionality invisibly
- Higher priority than inactive Activities, so less likely to be killed
- If they are killed, they can be configured to re-run automatically (when resources available)

Services

- You can use services to make your app run and respond to events when not in active use
- No dedicated UI, but they execute in the main thread of the application's process. This means they cannot cause processing lags.
  - CPU intensive tasks must be offloaded to background threads using Thread or AsyncTask

Services

- Threads can use Toasts and Notifications to send messages to the user
- Alarms can be used to fire Intents at set times (by OS, not your app). These can start services, open Activities, or broadcast Intents

Tracking Stephen's activity

- You will use Broadcast receivers to track Stephen's activity in Prog. Assignment 3
  - List of Intents
    - http://developer.android.com/reference/android/content/Intent.html
  - E.g.,
    - intentFilter.addAction(Intent.*ACTION_HEADSET_PLUG);*
    - intentFilter.addAction(Intent.*ACTION_MEDIA_BAD_REMOVAL);*
    - intentFilter.addAction(Intent.*ACTION_MEDIA_REMOVED);*
    - intentFilter.addAction(Intent.*ACTION_NEW_OUTGOING_CALL);*
    - intentFilter.addAction(Intent.*ACTION_PACKAGE_ADDED);*

Tracking Stephen's activity

- Intent tip:
  - Some Broadcast Intents don't work as advertised in the docs
  - Some Intents will only work if you add them via code, not via XML. You will not get an error if you define it in XML. It just won't work (e.g., Intent.*ACTION_TIME_TICK is definitely like this)*

Services

- The only reason Android will stop a service is to provide additional resources to foreground Activity
  - It is possible to increase the status of an Activity (such as music player) to a foreground activity
- Examples from OS: Location Manager, Media Controller, Notification Manager

Services

- The only reason Android will stop a service is to provide additional resources to foreground Activity
  - It is possible to increase the status of an Activity (such as music player) to a foreground activity
- Examples from OS: Location Manager, Media Controller, Notification Manager

Services

- The only reason Android will stop a service is to provide additional resources to foreground Activity
  - It is possible to increase the status of an Activity (such as music player) to a foreground activity
- Examples from OS: Location Manager, Media Controller, Notification Manager

Creating a service

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyService extends Service {

  @Override
  public void onCreate() {
    // TODO: Actions to perform when service is created.
  }

  @Override
  public IBinder onBind(Intent intent) {
    // TODO: Replace with service binding implementation.
    return null;
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    // TODO Launch a background thread to do processing.
    return Service.START_STICKY;
  }
}
```

onStartCommand

- Called whenever the Service is started with call to startService
  - So beware: may be executed several times in Service's lifetime!
  - Controls how system will respond if Service restarted (START_STICKY)
  - Run from main GUI thread so standard pattern is to create a new Thread from onStartCommand to perform processing and stop Service when complete

Stickyness

- START_STICKY
  - onStartCommand called anytime service restarts
- START_NOT_STICKY
  - For service started to execute specific action/command
  - Use stopSelf to terminate service when command complete
- START_RECEIVER_INTENT
  - Combines both above

Stickyness

- You can use the parameter passed to startService to determine if the service is a system-based restart
  - Null
    - Initial call
  - START_FLAG_REDILIVERY
    - OS terminated the service before it was stopped by stopSelf
  - START_FLAG_RETRY
    - Service restarted after an abnormal termination when service was set to START_STICKY

Determining start condition

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
  if ((flags & START_FLAG_RETRY) == 0) {
    // TODO If it's a restart, do something.
  }
  else {
    // TODO Alternative background process.
  }
  return Service.START_STICKY;
}
```

Manifest

- Always have to register services
- Also have to register Broadcast Recievers
- Permissions must be set correctly

Starting a Service

- Call startService

```
// Implicitly start a Service
Intent myIntent = new Intent(MyService.ORDER_PIZZA);
myIntent.putExtra("TOPPING", "Margherita");
startService(myIntent);

// Explicitly start a Service
startService(new Intent(this, MyService.class));
```

(To use this example, would need to include a MY_ACTION constant in MyService class and use an Intent Filter to register the Service as a provider of MY_ACTION)

Stopping a Service

- Call stopService

```
ComponentName service = startService(new Intent(this, BaseballWatch.class));
// Stop a service using the service name.
stopService(new Intent(this, service.getClass()));
// Stop a service explicitly.
try {
  Class serviceClass = Class.forName(service.getClassName());
  stopService(new Intent(this, serviceClass));
} catch (ClassNotFoundException e) {}
```

Binding Activities to Services

- Activity maintains reference to a Service
- Activity can make calls on the Service just as any other instantiated class
- To support this, implement onBind for the Service

```
private final IBinder binder = new MyBinder();

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

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

Binding Activities to Services

```
private MyService serviceBinder;   // Reference to the service

// 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 = ((MyService.MyBinder)service).getService();
  }

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

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  // Bind to the service
  Intent bindIntent = new Intent(MyActivity.this, MyService.class);
  bindService(bindIntent, mConnection, Context.BIND_AUTO_CREATE);
}
```

Binding Activities to Services

- Once Service is bound, all public methods and properties are available through the serviceBinder object obtained from the onServiceConnected handler

Background threads

- To make app responsive, move all time-consuming operations off main app thread to child thread. **Very important!**

- Two options:
  - AsyncTask
  - Write own Threads and Handler class

AsyncTask

- To make app responsive, move all time-consuming operations off main app thread to child thread. **Very important!** (You have 10s worst case before app killed ... Don't even want to be close)

- Two options:
  - AsyncTask
  - Write own Threads and Handler class

AsyncTask

- Simple, convenient mechanism for moving time-consuming operations to background thread
- Convenience of event handlers synched with GUI so you can update GUI on progress of thread computation
- AsyncTask handles thread creation, management, and synchronization

Creating AsyncTask

```
private class MyAsyncTask extends AsyncTask<String, Integer, Integer> {
  @Override
  protected void onProgressUpdate(Integer... progress) {  //Post interim updates to UI thread; access UI
    // [... Update progress bar, Notification, or other UI element ...]
  }

  @Override
  protected void onPostExecute(Integer... result) {    //Run when doInBackground completed; access UI
    // [... Report results via UI update, Dialog, or notification ...]
  }

  @Override
  protected Integer doInBackground(String... parameter) {   //Background thread. Do not interact with UI
    int myProgress = 0;
    // [... Perform background processing task, update myProgress ...]
    PublishProgress(myProgress)
    // [... Continue performing background processing task ...]

    // Return the value to be passed to onPostExecute
    return result;
  }
}
```

Important: power management

- Just because you have code in a BroadcastReceiver or Service doesn't mean it will run if the phone goes into a low-power state

- Common problem: create a Broadcast receiver. Create a thread from within it to run code....

Important: power management

- All works fine when phone on and plugged into computer during development
- Fails under normal use because phone shuts down quickly in power management state
- Need to use a WakeLock!

WakeLock

- Control the power state on device (somewhat)
- Used to
  - Keep the CPU running
  - Prevent screen dimming or going off
  - Prevent backlight from turning on
- Only use when necessary and release as quickly as possible

WakeLock

- If you start a service or broadcast an Intent with the onReceive handler of a BroadcastReceiver, it is posse the WakeLock it holds will be released before your service has started! To ensure the service is executed you will need to put a separate WakeLock policy in place.

Creating a WakeLock

```
PowerManager pm = (PowerManager)getSystemService(Context.POWER_SERVICE);
WakeLock wakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                                   "MyWakeLock");
wakeLock.acquire();
[ ... Do things requiring the CPU stay active ... ]
wakeLock.release();
```

- PARTIAL_WAKE_LOCK keeps the CPU running without the screen on

WakefulIntentService

- Step through example