CSC-306 Mobile Applications

Activity

Lifecycle

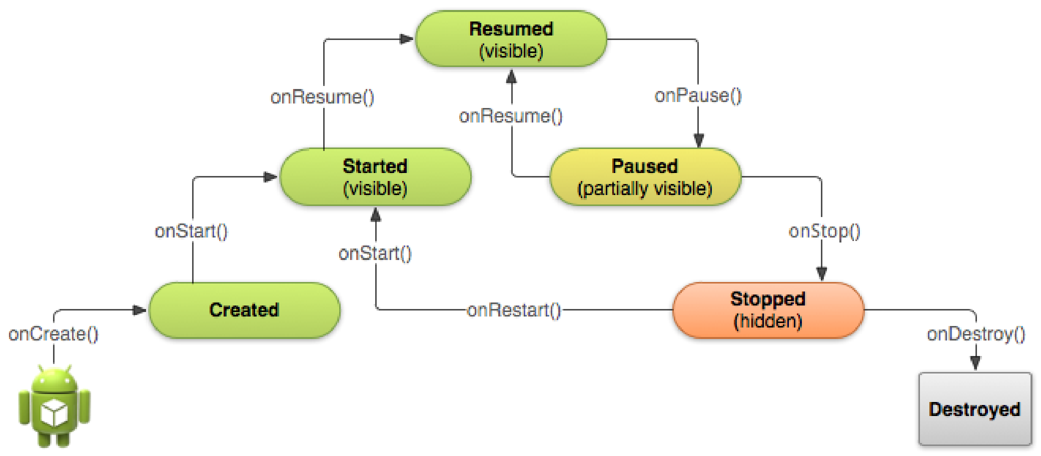
Resumed. In the foreground of the screen and has user focus. (running)

Paused. Another activity in the foreground and has focus, but this one is still visible.

- Completely alive
- retained in memory
- maintains all state and member information remains attached to the window manager
- can be killed by the system in extremely low memory situations.

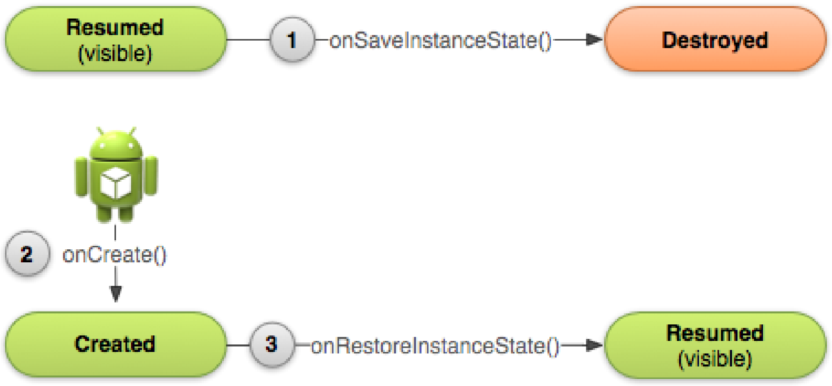
Stopped. Obscured by another activity

- still alive
- retained in memory
- maintains all state and member information
- *not* attached to the window manager.
- No longer visible to the user
- can be killed by the system when memory is needed elsewhere.



- onCreate() - Perform basic startup tasks such as creating an interface.
- onDestroy() - Cleanup operations such as killing threads, releasing locks.
- onPause() - Stop animations or other ongoing actions that could consume CPU. Release system resources, such as broadcast receivers, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.
- onResume() - Called every time Activity comes into focus (including when first started).
- onStop() - No longer visible. Should release all resources not needed while user not using the app.
- onStart() - always called when Activity becomes visible (including the first time).
- onRestart() - called when activity resumes from *stopped*.

Recreating Activity

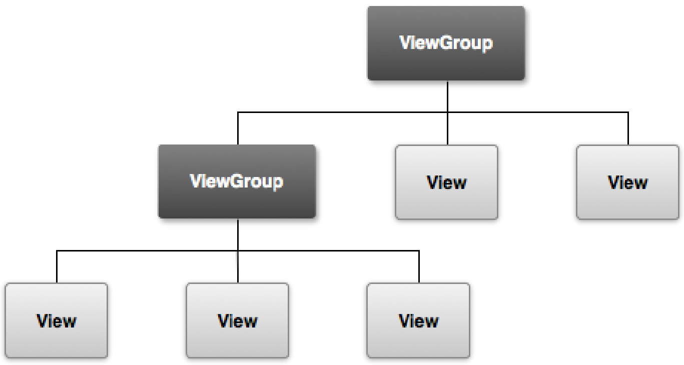


Use onSaveInstanceState() to save information other than view.

Use onCreate() or onRestoreInstanceState() to recover extra information.

Externalising

- XML files parsed by compiler and static file constants in **R**
- Good practice to keep functionality separate from appearance.
- Easier to maintain and debug.
- Easy to respond to e.g. orientation change (later in lecture)



Display Orientation

- When screen orientation changes current activity destroyed and recreated. Thus onCreate() is called.
- onCreate constructs View from R.java which reads the XML files.
- Thus write new XML file

Responding to Events

```
Public boolean onKeyDown(int keyCode, KeyEvent event){
        case KeyEvent.KEYCODE_DPAD_CENTER:
                Toast.makeText(getBaseContext(),
                        "Centre was clicked",
                        Toast.LENGTH_LONG.show();
                        break;
        …………
}
```

Registering Event Handlers

```
Button btn1 = (Button)findViewById(R.id.btn1);
btn1.setOnClickListener(btnListener);
Button btn2 = (Button)findViewById(R.id.btn2);
btn2.setOnClickListener(btnListener);
Private OnClickListener btnListener = new OnClickListener(){
        public void onClick(View v){
                Toast makeText(getBaseContext(),
                ((Button)v).getText() + ‟ was Clicked”,
                Toast.LENGTH_LONG.show();}};
```

```
<EditText android:id="@+id/edit_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
```

- android:id = unique identifier.
- Must start with **@** followed by type (in this case id)
- **+** needed when defining the resource. Not required for concrete resources such as strings.
- Android: the hint is default text for edit box.

Intents

Uses

- Declare intention that an Activity or Service start – usually on a piece of data
- Broadcast an event or action
- Explicitly start a Service or Activity

Explicit Intent - Create an Intent specifying the current application Context and name the **class** that is to start

Implicit - Ask the system to start an Activity to perform a task without knowing or caring which Activity. Nominate an action to perform and (optionally) supply the URI of the data upon which to perform it.

The Difference: *In the intent itself you name the explicit object or app that handles the intent whereas an implicit intent you expect the system to choose a system object or app for you.*

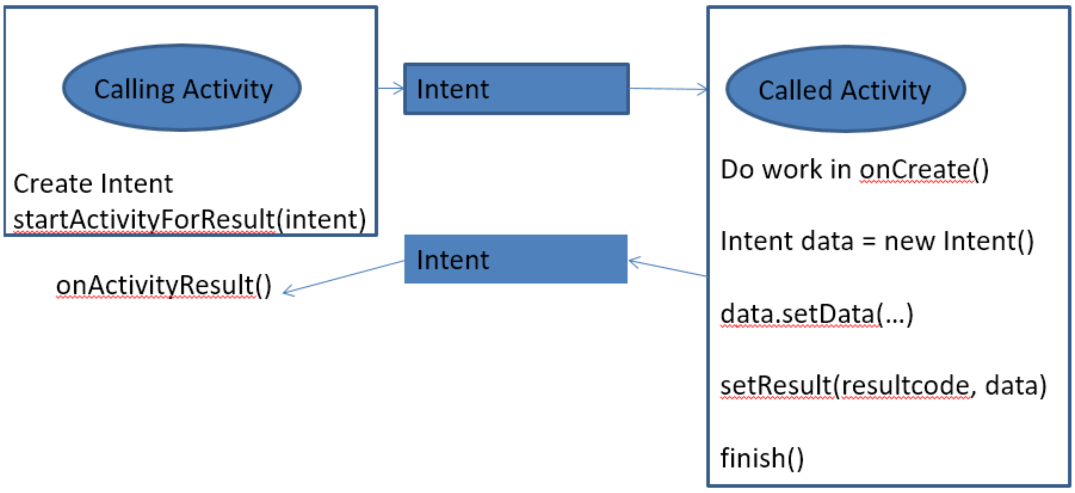
Returning Result from Intent

- The system reads intent; interrogates manifest.
- Activities started via Intents are independent of parent and won’t provide feedback.
- Start sub-Activity instead. Sub-Activity triggers an event within its parent when it closes.

```
private static final int ACTIVITYCODE = 1;
Intent intent = new Intent(this, SomeActivity.class);
startActivityForResult(intent, ACTIVITYCODE);
```

- When sub-activity ready to return call setResult() before finish()
- setResult takes two parameters, the result code and the result expressed as an Intent.
- Activity_RESULT_OK or Activity_RESULT_CANCELLED
- Can specify own result codes.

Handling Result



Three return parameters:

- Request Code – Code used to launch sub-activity
- Result Code – usually Activity.RESULT_OK or Activity.RESULT_CANCELLED
- Data – An Intent used to package returned data.

Data could be URI indicating where data is stored or could contain primitive values by using extras bundle.

Super is always called when handling result.

```
private static final int ACTIVITY_ONE = 1;
private static final int ACTIVITY_TWO = 2;
public void onActivityResult(int requestCode, int resultCode,
                            Intent data){
   super.onActivityResult(requestCode, resultCode, data);

   switch(requestCode){
      case(ACTIVITY_ONE):{
         if(resultCode == Activity_RESULT_OK)
            String phoneNumber = data.getStringExtra(STRING_ENTERED);
      }
   }
}
```

Intent Filter Resolution

- List exists of all Intent Filters registered
- Remove from list any who do not match action *or* category

    - Action – excluded if one or more actions defined and none match (possible to have no action specified)
    - Category - excluded unless *all* categories included
- Each part of URI compared to the data tag. Mismatches removed.

Persisting Data

Five methods (NESSI)

- Network
- External Memory
- Shared Preferences
- SQLite Database
- Internal Memory

Network

Stores the data via web-based services.

Pros

- Easy to share data between devices and computer
- Easy backup of data

Cons

- Not reliable, can't guarantee network connection
- Most difficult to setup with service and device components
- Uses up data bandwidth(which costs money for users)
- Slowest method as ping rates are usually much slower than local storage.

External Memory

Uses the device's external memory (i.e. SD Cards) to store data in files within the devices internal memory.

Pros

- Access to both public shared directories and application specific directory
- Can store binary files
- Space is not an issue with external storage.
- Easy access by the user to backup data to the computer if necessary.

Cons

- Usually, external storage is slower than internal storage
- No real built-in framework, so lots of customization necessary.
- Not suitable for retrieving/sorting through large amounts of data.
- Not reliable as the user has access and can modify data/remove the external storage at any time. So also requires more error checking.

SQLite Database

Uses the devices internal memory to store data in an SQLite database file.

Pros

- Quick to sort and retrieve large data sets
- Can easily be used to store customised data
- Can enforce relationships and integrity between data

Cons

- Uses space limited internal device memory
- More difficult to setup.
- More difficult and error prone when doing structure changes.

Types:

- Text = Java String
- Integer = Java Long
- Real = Java Double

No other types are available. All data must be converted to one of these.

All queries return a Cursor object. Provides: getCount(), moveToFirst(), moveToNext(), isAfterLast()

Shared Preferences

General framework in Android that will allow you to save/retrieve key-value pairs for primitive objects such as boolean, float, int, long and string.

Pros

- Built-in to the existing Android framework, so easy to use, and easy to create a preference activity to view and edit the preferences
- Quick
- Can be used to store any data with manipulation with the string type.

Cons

- Not suitable for storing very large amounts of data, as it will be slow in retrieving and sorting through the data.
- Use internal device memory (more limited than external)

Requires customization for any non-primitive data types.

```
SharedPreferences prefs = getSharedPreferences( "com.example.app",
MODE_PRIVATE);
```

Internal Memory

Uses the device's internal memory to store data in files within the devices internal memory.

Pros

- Many modes of permissions make this fairly versatile. You can have it usable only by your app, or shared to other applications as read-only or writable.
- Uses the quicker internal device memory
- Can use the cache folder to cache files (that will be removed by Android if it needs space)

- Can be used to store binary files.

Cons

- Uses the space limited internal device memory
- No real built-in framework, so lots of customization necessary.
- Not suitable for retrieving/sorting through large amounts of data.

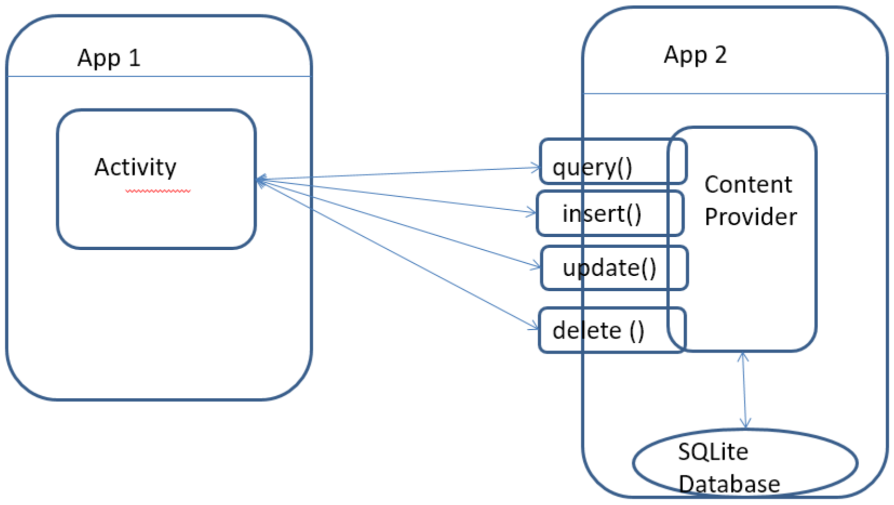
Writing

```
FileOutputStream fout = openFileOutput("filename", MODE_WORLD_READABLE);
OutputStreamWriter osw = new OutputStreamWriter(fout);
osw.write("data to write");
osw.flush();
osw.close();
```

Reading

```
FileInputputStream fin = openFileInput("filename");
InputStreamReader isr = new InputStreamReader(fin);
char[] buffer = nerw char[BLOCK_SIZE];
String s = "";
int charRead;
While((charRead = isr.read(buffer)) > 0 {
       Styring readString = String.copyValueOf(buffer,  0,  charRead);
                    s += readString();
                    buffer = new char[BLOCK_SIZE];
```

Content Providers



Basics

Access via ContentResolver, which provides basic CRUD functions.

```
Cursor cursor = getContentResolver();
```

```
Cursor.query(
   UserDictionary.Words.CONTENT_URI,
   projection,
   selectionClause,
   selectionArgs,
   sortOrder
);
```

Manifest

Permissions required specified in provider's manifest

e.g. User Dictionary requires android.permission.READ_USER_DICTIONARY

And

android.permission.WRITE_USER_DICTIONARY

<uses-permission android:name="android.permission.READ_USER_DICTIONARY">

Building a Provider

Necessary if:

- You want to offer data or files to other apps
- You want to allow users to copy complex data from your app into other apps.
- You want to provide custom search suggestions using the search framework.

- Define subclass of ContentProvider
- Define authority string (MUST BE UNIQUE) so use normal conventions
- Add the above to column names and define these as constants in a Contract class

Content Provider Methods:

- query()
- insert()
- update()
- delete()
- getType()
- onCreate()

Considerations

- All methods except onCreate() can be called by multiple threads, so synchronise code.
- Avoid lengthy operations in onCreate()
- Consider just returning 0 if you don't want to allow an operation.

Broadcast Intents & Broadcast Receivers

To inform user of system or application events

Android broadcasts intents extensively to announce battery charge levels, incoming calls, network connections, etc.

Broadcasting

- Construct an Intent
- Send using sendBroadcast()

```
public static final String WE_ARE_UNDER_ATTACK =
       “uk.ac.swansea.action.KLINGONS”
Intent intent = new Intent(WE_ARE_UNDER_ATTACK);
intent.putExtra(“under attack”, weComeInPeace);
sendBroadcast(intent);
```

Receiving

```
Extend BroadcastReceiver
Override onReceive()
import android.content.BroadcastReceiver;
import android.content.Context;
Import android.content.Intent;
public class MyReceiver extends BroadcastReceiver{
     public void onReceive(Context context, Intent intent){
          //React
     }
}
```

Registering a Broadcast Receiver

In manifest

- Add <receiver> tag within application node
- Include intent-filter tag with action code being listened for

```
 <receiver Android :name=“”.MyReceiver”>
   <intent-filter>
      <action android:name=“uk.ac.swansea.UNDER_ATTACK”/>
   </intent-filter>
</receiver>
```

- Receivers registered in code respond to Intents only when encapsulating app is running
- Useful for e.g. updating UI elements in Activity
- Register receiver dynamically using Context.registerReceiver()

In code

```
IntentFilter filter = new IntentFilter(UNDER_ATTACK);
MyReceiver receiver = new MyReceiver();
registerReceiver(r, filter); (Done in onResume())
```

Unregister receiver when done using application unRegisterReceiver()

```
unRegisterReceiver(receiver); (In onPause())
```

Services

- Services have little or no interaction = no GUI
- Used for updating Content Providers, triggering Notifications, firing Intents, etc.
- Perform ongoing or regular processing and handling events when app is invisible, inactive or finished

Lifecycle

- Higher priority than other background Activities.
- Less likely to be terminated by the system.
- If they are stopped by the system (usually) restarted automatically.
- Sometimes necessary to increase priority to the same as foreground Activity (e.g. music player)

onStartCommand()

- Called when Service started by call to startService()
- May be called several times during Service's lifetime.
- Tells the system how to handle restarts if Service killed by system

Flags

- START_STICKY onStartCommmand() called when Service restarted after system termination.
- START_NOT_STICKY used for Services launched to perform specific tasks. Such Services explicitly stopped when task completed. Only restarted if there are pending start calls.
- START_REDELIVER_INTENT used to ensure. Service completes the task. If service terminated by the system will be restarted if it has pending start calls OR process killed before calling stopSelf().

Registration

```
registerReceiver(r, filter); (
<service
   android:enabled="true"
   android:name=".MyService"
```

```
/>
```

- Use the requires-permission tag to demand that other apps need permission to use this service.
- Can set intent filters. Allow Service to be started implicitly. Otherwise must be named explicitly.
- Can be made private to this app by setting android: export to false

Stopping Services

- Services rarely killed off by the system, therefore should be explicitly stopped by calling Activity.
- stopService(), allows being restarted if other requests are pending.
- stopSelf(), forces stop. Typically called when service has finished its job.

Bound Services

- Binding a Service to an Activity creates a client(Activity)-server(Service) relationship between them.
- The binding Activity maintains a reference to the Service.
- Can interact with it, get feedback from it and use its public methods.

Implementing iBinder

1. Extend Binder. Do this if the Service is to be private to your Activity.
2. Use a messenger. Do this if your Service is to be available to multiple other clients running in their processes.

Two-Way Communication

Using a Messenger

- The Service implements a Handler which receives a callback for each call to the Service and queues them.
- The Handler is used to create a Messenger object.
- The Messenger creates an Ibinder which the Service returns to clients in onBind().
- The client uses the IBinder to create the Messenger, which the client then uses to send Message objects to the Service.
- The Service receives each Message in its Handler in handleMessage().

Communicating

Need to also create a Messenger in the client.

Then when the client receives the onServiceConnected() callback, it sends a Message to the service that includes the client's Messenger in the replyTo parameter of the send() method.

Multimedia

Playback via MediaPlayer or AudioTrack

- MediaPlayer audio playback: Data must be in a file or stream based
- AudioTrack: Can play raw audio in memory
- SoundPool: Play short pre-recorded sounds such as effects in games

Video Playback

Can only use MediaPlayer. Needs a Surface on which to display itself. Provided by VideoView.

```
VideoView videoView = (VideoView)findViewById(R.id.videoView);//assumes it is
described in layout file

if(videouri != null) videoView.setVideoUri(video_uri);
else videoView.setVideoView(video_path);
mediaController = new MediaController(this); //Gives us a nice interface
mediaController.setAnchorView(videoView);
if(videoView.canSeekForward())
       videoView.seekTo(videoView.getDuration()/2);
videoView.start();
```

Recording

States

- Initialise – The recording device is initialised
- Initialised – Ready to be used
- DataSource configured – The media source (where it is to be placed) is configured
- Prepared – Ready to record
- Recording- - recording is taking place
- Released – All resources are released

Using an Intent to record

```
Intent intent = new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);
startActivityForResult(intent, 1);
```

This launches the default recorder (assuming it exists) and starts to record. Once that returns, we can query the onActivityResult intent to see what happened and access our recording.

```
protected void onActivityResult(int requestCode, int resultCode, Intent
intent);
if(requestCode == 1){
     if(resultCode == RESULT_OK){
          Uri audioUri = intent.getData();
          playAudio(audioUri);
     }
}
```

Widgets

Canvas - Finding space on the screen: onMeasure(); Drawing in that space: onDraw();

onMeasure()

- Each parent in post-order (depth-first) asks its children how much space they want.

- The process can be repeated several times – function must be idempotent.
- Parent in absolute control. Doesn't have to grant request
- X = 3 – idempotent (never changes state).
- X = x + 3 – NOT idempotent (each call returns different result).

Options

Container gives three different choices to children when requesting space. It offers them dimensions and states whether this is:

- MeasureSpace.EXACTLY – widget has no choice but to accept space granted by parent.
- MeasureSpace.AT_MOST – Parent gives child maximum space, and child can ask for anything up to that.
- MeasureSpace.UNSPECIFIED- No restrictions. Child can ask for what it likes.

Drawing Tools

- Canvas – An easel which can bend, distort, twist, etc. NOT like a real canvas!
- Paint – The painting media, ink, colours, styles, etc.
- Bitmap – The paper on which we are drawing.
- Drawable – The thing we are drawing, a rectangle, circle, or more complex image.

Canvas Matrix

Can manipulate image infinitely.

- Rotate(angle);
- Translate(x, y);
- Scale(x, y);

Complex transformations possible – pre-concat and post-concat.

Location Based Services (LBS)

- Consumes lots of battery
- Getting location takes time
- Need to re-estimate every so often
- Location estimates may vary in accuracy

android.permission.ACCESS_FINE_LOCATION" /> //for GPS.

android.permission.ACCESS_COARSE_LOCATION //for cell phone or network.

Location API

- Location – class representing geographical location at a particular time
- LocationManager – provides access to system location services
- LocationListener – used for receiving notifications from Location Manager when location changes

Getting Location

- Start listening for location updates
  - locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,0,0,locationListener)
- Stop listening
  - locationManager.removeUpdates(this)

Trade-Offs

- power consumption
- longitude/latitude accuracy
- altitude accuracy
- speed
- direction information

Geocoding

- Forward – from address to longitude/latitude
- Backward – from long/lat to address (maybe only partial info)

Overlays

Markers in Arraylists

```
Uri audioUri
private ArrayList<OverlayItem> mOverlays = new ArrayList<OverlayItem>();

Define constructors:
public HelloItemizedOverlay(Drawable defaultMarker) {
       super(boundCenterBottom(defaultMarker));
}
```

Add Overlays to Arraylist

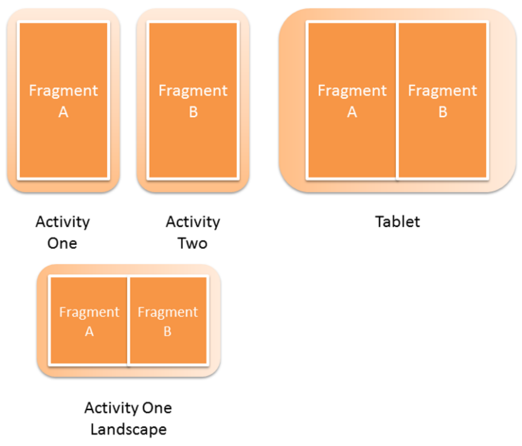
```
Uri audioUri
public void addOverlay(OverlayItem overlay) {
       mOverlays.add(overlay);
       populate();
}
//Must call populate() to draw overlay

Populate() calls createItem(int)
@Override
protected OverlayItem createItem(int i) {
       return mOverlays.get(i);
 }
//Must also override size();
@Override public int size() {
       return mOverlays.size();
}
```

Creating Overlay items

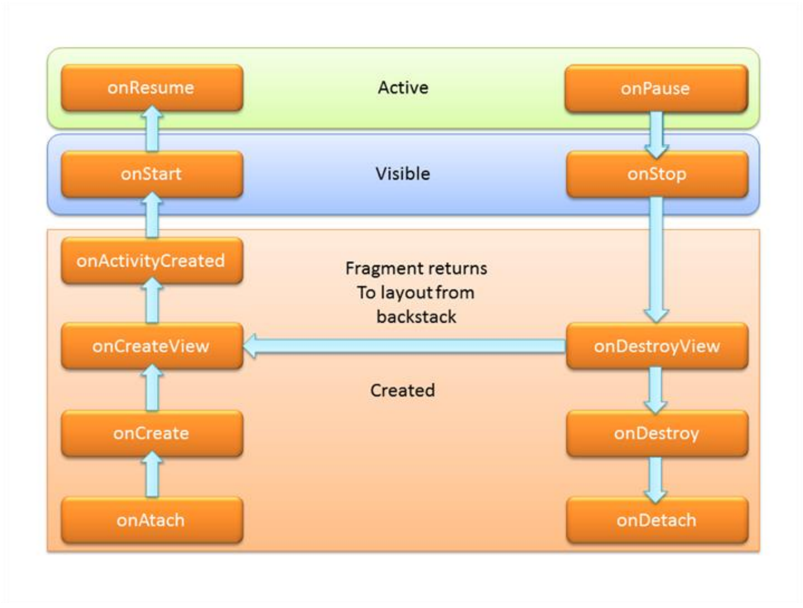
```
          Uri audioUri
//At the end of onCreate() add:
List<Overlay> mapOverlays = mapView.getOverlays(); Drawable drawable =
this.getResources().getDrawable(R.drawable.androidmarker);
HelloItemizedOverlay itemizedoverlay = new HelloItemizedOverlay(drawable,
this);
```

Fragments

- A Fragment is like a sub-Activity.
- Used for gaining extra control over different devices and UIs
- No interface
- Must be attached to Activity to exist



Lifecycle



Fragment Handlers

Attaching & Detaching

onAttach() - called when the fragment is attached to its parent activity. Happens before User Interface has been created and before the fragment's parent Activity has finished initializing.

onDetach() - called when the fragment has been detached from its parent activity.

Creating & Destroying

onCreate() -used to initialise fragment. Create class objects here. Called once in the lifetime of the fragment.

onDestroy() - can be used to release resources for use elsewhere.

onCreateView() - used to initialize the fragment. This is where you inflate the User Interface and get references to its views.

onDestroyView() -called when the fragment's view has been detached from its parent activity. Use it to release any resources used by the view.

Fragment Manager

- Used in parent Activity
- The Fragment Manager has methods which can be used to access the currently embedded fragments.
- Add, replace and remove fragments while the activity is running.
- Fragments can be attached statically or dynamically

Static

- Easiest way to embed fragments in an Activity is to define the fragments in the Activity's XML layout file.
- The layout is then inflated in the Activity, and the fragment becomes a View Group, within Activity's layout.
- Fragment manages its own User Interface within the Activity.
- Great for static layouts for various screen sizes.

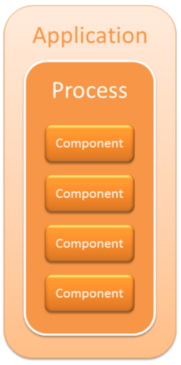
Dynamic

- Want to dynamically modify layouts by adding, removing and replacing fragments while the Activity is running?

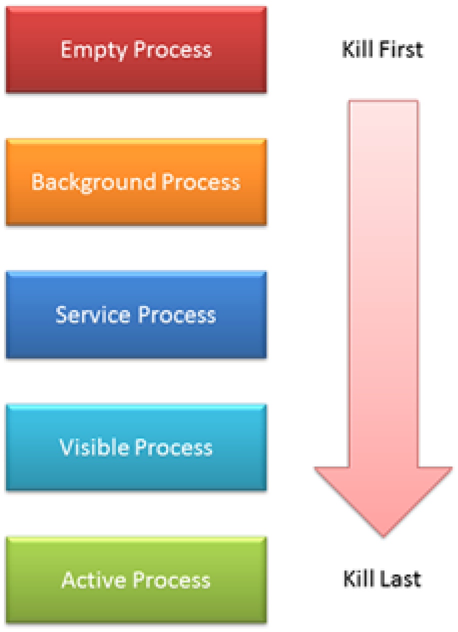
- Use container views in the Activity’s layout.
- Then at runtime, create the fragments in the Activity’s onCreate() method and place them in the containers.

Processes and Threads

Isolation



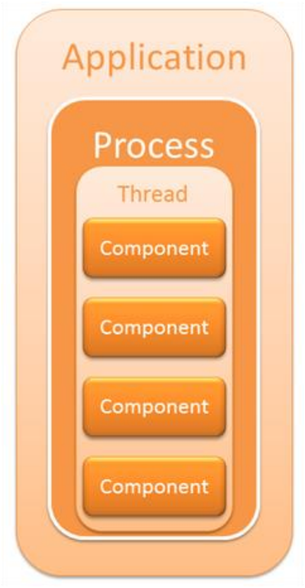
Killer List



Two Rules of the Road

- Don’t block the Main thread
- Don’t try and access the UI Toolkit (the elements that the user interacts with) directly from the worker thread

Main Thread



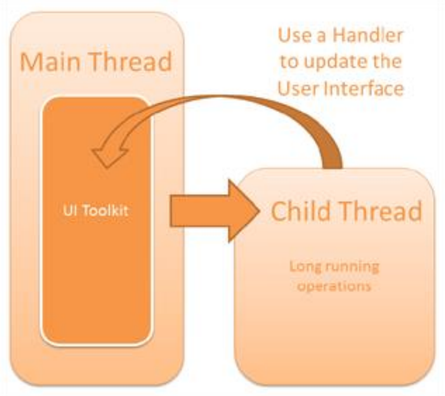
What Triggers "Application Not Responding" (ANR)?

- If an app can't respond to user input within 5 seconds
- If Broadcast receivers haven't finished executing within 10 seconds

Alternatives

You can use your own Threads and use the Handler class to notify the main thread when the work is done (This lecture)

- You can use the AsyncTask class to do the work in the background.
- Use a service for longer running operations.
- Use Loaders for asynchronous data loading.
- IntentServices are the best practice for background services.
- You can use Broadcast receivers to do small amounts of work in the background
- Instead of using threads for intensive tasks, use an IntentService


Main Thread
Use a Handler to update the User Interface
UI Toolkit
Child Thread
Long running operations