ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 6 — January 18, 2013

Patrick Lam *version 1*

Today's Plan. Fridays are going to be Android days. Today, we'll talk about XML; inversion of control; handlers and runnables (setting up timers); and about the Android `Activity` class, which is where you'll be putting code for this course.

XML

You need to know a bit about XML to build Android applications, so here's a quick description. For further reading, you can consult many resources on the Web, like `http://www.w3schools.com/xml/default.asp`.

XML in one line.

XML is a *structured document format*.

All XML documents therefore have the same format. However, XML has no intrinsic meaning. It just separates content from structure.

Let's look at an example of an Android manifest. It's an XML format document.

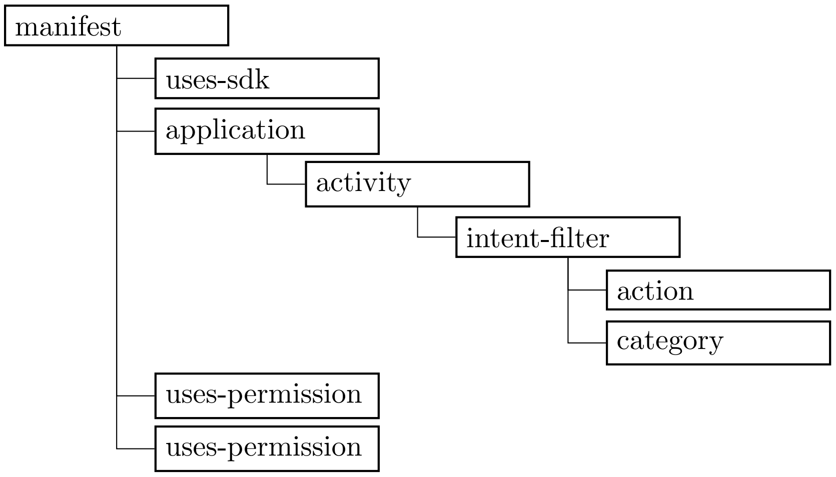
```
<?xml version="1.0" encoding="utf-8"?>
<manifest   ——→ root node
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="ca.uwaterloo.Lab1_plam"    ——→ attribute, name is “package”, value is “. . . Lab1_plam”.
  android:versionCode="1"   ————————→ must quote all values, e.g. “1”
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="10"
    android:targetSdkVersion="16" />   —→ self-closing tag

  <application  —→ application tag is nested within manifest tag
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity   ————→ more nesting
      android:name="ca.uwaterloo.Lab1_plam.MainActivity"
      android:label="@string/app_name" >
      <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_CONTACTS" />
</manifest>
```

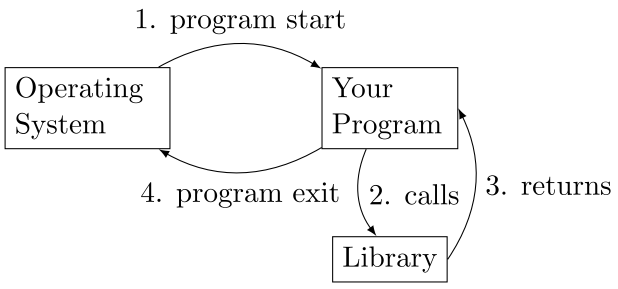
Structure of XML files. XML is always tree-structured: at the top level, there is a *root* element. (Ignore the XML declaration.) So we can convert the textual form above into a tree. Tags must be well-nested: you can't open a tag `<a>` and then close a tag `</b>` without closing `</a>` first.



Inversion of Control

As we talked about last time, Android programming is event-driven programming, which changes the basic structure of a program from what you saw in ECE150.

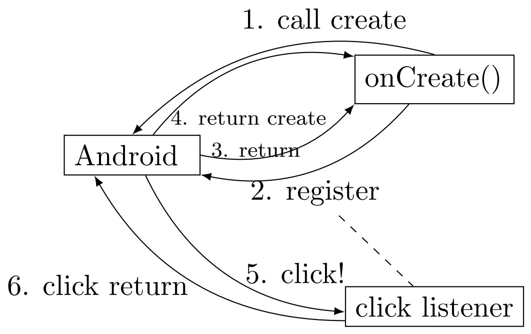
Here's a representation of how a program works, as you saw it in ECE150:



The main idea is that when the user chooses to launch your program, you have all the control until you exit. If you want some input, you'll ask the library/operating system for the input (using a system call), and it'll return it to you when it returns.

Note that calling the library does not lead, by itself, to an inversion of control.

New Event-Driven Paradigm. Here's an example of the new paradigm.



Now, Android takes all of the responsibility for calling your code when appropriate. It'll start your Activity by calling the `onCreate()` method. In that method, you may choose to register event listeners, or programmatically add widgets. But you only do setup work there. Don't do any real computation.

Next, when something happens, Android is going to call you and let you know about it. It's doing something that looks like this:

```
while (!done) {
 r <- fetch Runnable from Queue
 dispatch r
}
```

This is like a tight polling loop, but it goes to sleep between events (in the fetch).

Timers in Android

Remember from last time: we want Android to send us an event in the future. (The actual guarantee is that the event will occur no sooner than a given time from now). Here's how:

1. Create a `Handler` object, call it `h`.
2. Create a `Runnable` object `r` using an inner class; code the behaviour that you want to happen.
3. Call `h.postDelayed()` with the `Runnable` and the requested delay.

Handler objects. Each Android thread has a message queue (remember, inversion of control). A `Handler` object allows you to manipulate the message queue for a thread. You can enqueue events for later execution.

Code example: `Handler h = new Handler();`

Runnable objects. A `Runnable` object represents a task. You get to say what's in the task. You do this using an inner class (from Lecture 4) and implementing the `run()` method.

Note that you don't run the task explicitly; you ask Android to run it for you.

Here's an example of creating a `Runnable`:

```
Runnable r = new Runnable() {
  public void run() {
    // execute the task
  }
}
```

Putting It All Together. You have a `Runnable`, called `r`, and a `Handler`, called `h`. To put them together, just call the `postDelayed()` method, like so:

```
h.postDelayed(r, delayInMS);
```

Android Activity class

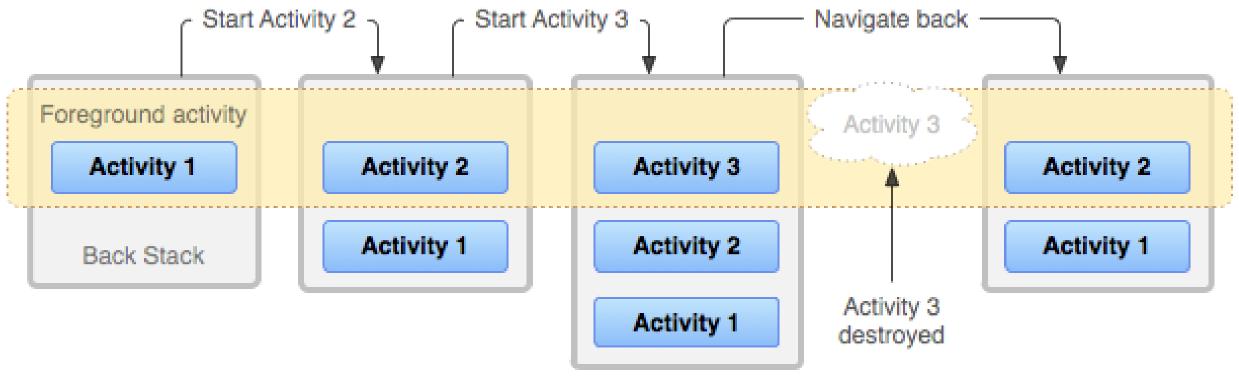
"An activity is a single, focused thing that the user can do."

Usually, an Activity corresponds to a full-screen window which the user may interact with. For instance, the user may:

- set up a timer;
- read off sensor values; or
- make a phone call.

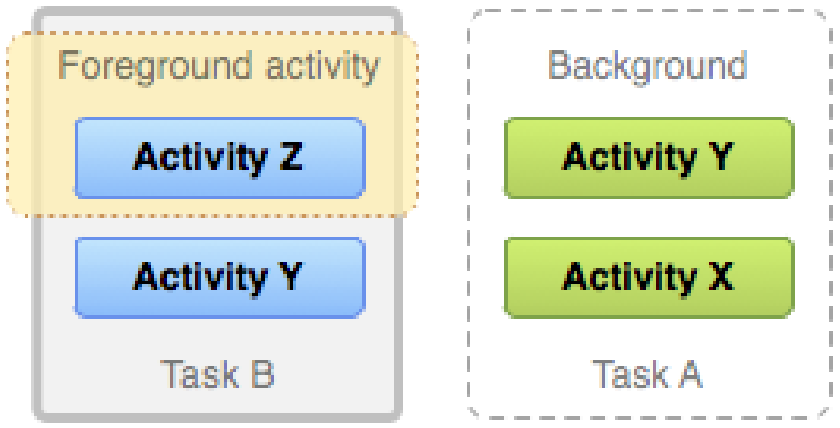
Applications may contain multiple activities, each of which corresponds to a thing that the user wants to do. Android organizes activities into tasks. A task consists of a last-in, first-out stack of activities, possibly from different applications.

Task Navigation: the Back button. The Back button pops the topmost activity off the stack and gets rid of it.



(from `http://developer.android.com/images/fundamentals/diagram_backstack.png`, retrieved January 18, 2013)

Task Navigation: switching tasks. It is also possible to switch between tasks. Switching tasks puts a different activity and its stack in the foreground, and puts the old activity in the background.



(from `http://developer.android.com/images/fundamentals/diagram_multitasking.png`, retrieved January 18, 2013)

Doing Something in Your Activity

The most useful activity method, where you'll be writing a lot of code, is `onCreate()`. It gets executed when the activity starts. Typically, it will set up the user interface, namely:

- creating widgets;
- setting up event listeners;

Note that you must call `super.onCreate()`; this is taken care of for you in the autogenerated boilerplate code.

Retrieving Widgets. If you've declared widgets in the XML file, you can use the `findViewById()` method to get a hold of them. Note:

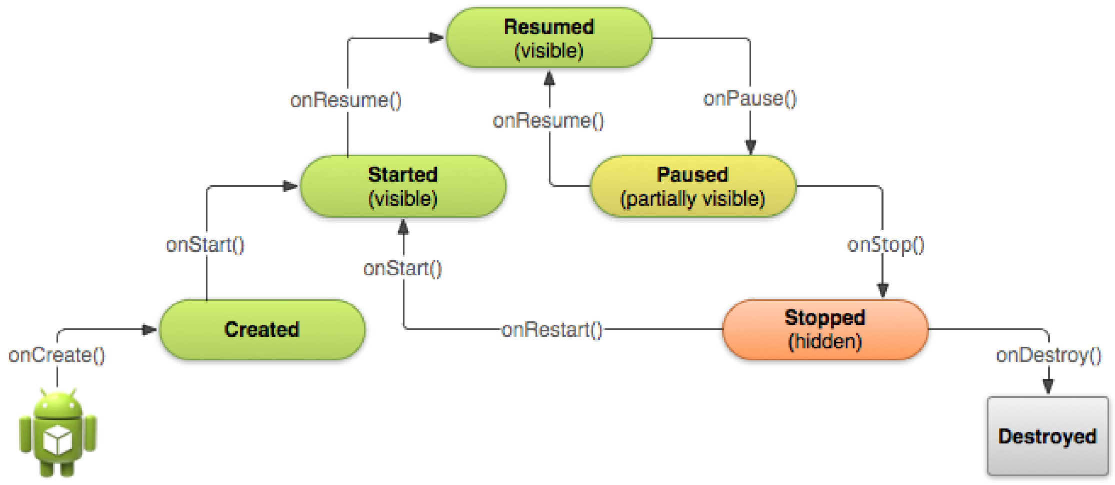
- you need to cast the return value, e.g.
  `tv = (TextView) findViewById(R.id.t);`
- you must save the XML file to get the right ids on the `R` object.

Programmatically Adding Widgets. We ask you to add widgets programmatically in Lab 1. There are two steps:

1. Create the widget: `tv = new TextView(getApplicationContext());`
2. Add it to the Activity: `addView(tv);`

Activity Lifecycle

Although we've only talked about `onCreate()`, there are numerous other methods on `Activity` which Android calls at various times (inversion of control!)



(from `http://developer.android.com/images/training/basics/basic-lifecycle.png`, retrieved January 18, 2013)

Eclipse demo

I did an Eclipse demo as follows:

- Create a new Android project.
- Add an `EditText` to the main `Activity`.
- Use `addTextChangedListener` to watch for changes in the text.
- Use Quick Fix to get method stubs in the `TextWatcher` inner class.
- Add code to the `afterTextChanged` method.