Android Development - Manipulating GUI Objects, Listeners, Event Handlers and a note on Debugging

Prepared by Richard Foley

"Connecting" source code objects to Class Instances

Some GUI screen "objects" are "decorative" and so never need to be manipulated by code (e.g. a label field, via the **TextView** class in the Android SDK, which never changes its text value). However, other GUI screen "objects" need to be manipulated by code (e.g. an editable text box, via the **EditText** class in the Android SDK, into which you can, amongst many other things, enter text from the screen and so you may want to get its text value dynamically at some point during program execution).

Dynamically obtaining a programmable "pointer" to a physical GUI object

With this (i.e. the latter **EditText** example from above) you will have to declare a class instance variable for the **EditText** "object" in your project's **Activity** class and then initialise that to the physical "object" described in the (main.)xml file for the screen it is part of. Thus the class instance should be declared "globally" (normally) as a **private** class instance variable and in the **onCreate()** method you need to initialise it using the **findViewById** method, giving that method the appropriate class instance variable from the **R.java** code. E.g. presume that we have an **EditText** box in the **main.xml** file called **myEditText**. This will be shown in the xml as the following attribute of the **<EditText** tag: **android:id=*"@+id/myEditText"***. Thus in your Activity class if you have the following:

```
private EditText myEditText;

@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    myEditText = (EditText) findViewById(R.id.myEditText);

}
```

Then the class instance variable **myEditText** will be initialised to, and be able to manipulate, the physical screen object. In the above code fragment I have used the same name for both the physical resource and the program object, but that is just my convention, there is no requirement for that. The **findViewById** method just returns a "generic" object pointer and so you should note that you have to type cast it to your class instance type (in this case **EditText**). The same structure would be used for other GUI "objects" (such as a **Button**).

The Overall Picture for Handling Events on a GUI

With any GUI based development environment the program code has to "link" the handling of events on an "object" on the interface with an "object" in the source code. E.g. when a button is pressed on the interface this generally has to result in a specific method being called in the source code. That way you can code functionality in that method to execute the tasks you want to occur when the button is pressed. In java (and Android) this is done via Listeners (which are Interfaces) and Handlers (which are methods). Essentially you have to "register"

(or rather set) a graphical component (such as a Button or Textbox) as having the Listener for a particular type of event and then you have to write a particular handler method for that event which is "attached" to the Listener which has been set for that GUI object. There are several (in fact 3) ways of implementing this mechanism. Which one you decide upon very much depends upon the simplicity or complexity of the interface requirements. For example, in a simple object (such as a Button) this "registration and attachment" can be done very simply via the XML editor. However with more complex objects (such as an EditView or a SeekBar) it has to be/is better done dynamically via code as a two stage process of first creating a listener class for the given object (this listener class will have methods for each of the events which are associated with the object) and then setting this listener class as the listener for the programmable object. This can also be done in one of two ways, either by implementing the listener directly as part of the Activity class or by having a separate listener class.

Simple Event Handling via the XML Editor

In the XML graphical editor for the screen, simply right click on the button and select **Other Properties** -> **Inherited from View** -> **OnClick ...**, this will bring up a dialog box into which you should enter the name of the method you wish to be the handler to be called if the button is physically "clicked". You can check the source xml and you will see a tag of the form **android:onClick=*"myButtonClickHandler"***. Then you have to "manually" add a method of this name as follows to your activity class:

```
public void myButtonClickHandler(View v)
{
	//put your code to handle the functionality
	//for the event here
}
```

Event Handling by Implementing the Listener as part of the Activity Class

This is a “two stage” process which involves implementing the listener’s abstract class interface as methods in the Activity class and setting the “objects” listener to the **Activity** class itself. Thus the following code fragment implements the same as above:

```
public class myProgActivity extends Activity implements OnClickListener
{
    /** Called when the activity is first created. */
      private Button myButton;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ...
        myButton = (Button) findViewById(R.id.myButton);
        myButton.setOnClickListener(this);
        ...
    }

@Override
      public void onClick(View v)
      {
            // TODO Auto-generated method stub[1]
            //put your code to handle the functionality
            //for the event here
      }
} // End of class
```

You should recall from your use of the Eclipse IDE for Android that there is a popup helper dialog which appears when you "hover over" the red squiggly line of any error. For example you will get this when you first type in the use of one of the Android classes for the GUI, but haven't included the corresponding import statement at the start of your source code file. This is a useful piece of IDE help since you can automatically select the suggestion to add (say) the import to your code. Thus when manually you add the **implements OnClickListener** to the first line to give you the full line:

```
public class myProgActivity extends Activity implements OnClickListener
{

      // Code to be executed when Listener called is entered here

} // End of Listener
```

then you will get a red squiggly line on the word **OnClickListener** (as above). On "hovering over it" you can select the **import 'OnClickListener' (android.view.View)** suggestion. However, when you do this you then get a red squiggly line under the name of your **Activity** class (i.e. **myProgActivity**). The "help when you hover" over it now indicates that you **must implement the inherited abstract method View.OnClickListener.onClick(View)**, and so you should accept its option to automatically add/generate this unimplemented method for you (which will added at the end of the source code file)

[1] You should note automated comment here and the associated mechanism for automatically implementing the listener's callback method (onClick) which is explained below

Event Handing with an Explicit Listener Class

This is also "two-stage" mechanism which generally involves creating an anonymous class implementation for the listener/handler and adding/setting that to the object. For example, if we take the clicking of a button, the following code fragment implements the same as above:

```
private Button myButton;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ...
        ...
        myButton = (Button) findViewById(R.id.myButton);
        myButton.setOnClickListener(myButtonListener);

    }

    private OnClickListener myButtonListener = new OnClickListener(){

            @Override
            public void onClick(View v)
            {
                  // TODO Auto-generated method stub[2]
                  //put your code to handle the functionality
                  //for the event here
            }
    };
```

Note: since this is an anonymous "in-line" class implementation it is in a "single" declaration statement. Thus it ends with a semicolon (**;**) after the last curly brace (**}**). Don't forget this.

The **Button OnClickListener** is an **Interface** which you have implemented for the functionality of your code. It only has a single abstract method **onClick**. Essentially when you create your own (anonymous) implementation of this interface, then you implement (by overriding) your own "version" of the **onClick** method of the interface. Thus, when the Android OS framework traps the clicking of the button and automatically calls the **onClick** method of the **OnClickListener**, then it will be your implemented version which will actually be called.

In a similar way to the previous method of implementing the listener, when you are creating listeners using the above method, then when you are typing it in Eclipse and accepting the various import/help suggestions as you get the red squiggly lines as you type, then (normally) when you get to:

```
    private OnClickListener myButton = new OnClickListener (){
```

and then press the return key, it will automatically put in the ending curly bracket after a blank line, you should then put in the semicolon (**;**) after that curly bracket. After all of this, it will probably also show a red squiggly line as above on the first occurrence of

[2] You should note the mechanism for automatically implementing the listener's callback method (onClick) which is explained below

OnClickListener. When you hover with the mouse over this, the first suggestion (which you should accept) is to **import 'OnClickListener' (android.view.View)**. Then you will get another red squiggly underneath the second (i.e. the constructor) occurrence of **OnClickListener**, e.g.

```
private OnClickListener myButton = new OnClickListener (){
```

For this the "help when you hover" over it indicates that this type **must implement the inherited abstract method View.OnClickListener.onClick(View)**, and so you should accept its option to automatically add/generate this unimplemented method for you.

Which type of Listener implementation should you use?

The first method can only be used with some very simple objects/callback methods, essentially the **onClick** method for an object. Thus it is not very flexible. Which of the other two you might use, would be a matter of personal taste as well as being dependant on the complexity of the interface. For example, you might have a screen which has more than one button. However, if you use the mechanism of implementing the listener as part of the complete Activity class (the second approach), then whichever button you press it will always call the same single callback (**onClick**) method. Then you would have to put code in that method to first check which button the event occurred on and then take different action appropriately. This might make the code for that single method less readable/understandable. In this case you might want to use the third method and you would create separate listeners for each button's functionality and set the different listener to the different button. That way the code in each individual handler would be "cleaner" and potentially more understandable.

Listeners with more complex Event Handler Callback Methods

The Button is a very simple "Object" since it essentially has only one event which your generally handle.s A number of objects have more complex event handling. For example the "object" pictured below is an Android **SeekBar** (some other languages/platforms vie this as a type of Scrollbar).

In the Android development reference documentation
http://developer.android.com/reference/android/widget/SeekBar.OnSeekBarChangeListener.html
then you will see that it has three callback handler methods:

- **onProgressChanged**: Notification that the progress level has changed.
- **onStartTrackingTouch**: Notification that the user has started a touch gesture.
- **onStopTrackingTouch**: Notification that the user has finished a touch gesture.

When you implement your listener, you need to remember that you have to implement all three methods, even if you don't intend to trap/use all three events in your code. This again where the "help when you hover" pop-up is handy since it will put in the skeleton for all (three) callback methods and you can just leave the one's you don't use "empty".

Simple Debugging in Android when using the Emulator

The simplest (and often the most effective) way to debug is to place **println** statements within your source code to "animate" its execution. Often the key to debugging in a mobile/GUI app is to check to see that you are in the correct event handler, and/or within some code you can indicate that you are at (or after) a particular statement/execution point and also output the value of some variable you expect to have a particular value. This is most effectively done (particularly for a beginner learning a new platform/language who will be developing comparatively simple programs) not be using the (often complex) features of the inbuilt "professional" debugger of the IDE, but by simply putting in appropriate **println** statements at appropriate points. In java this is of the form:

System.out.println(<*your string value*>);

In Android, **System.out** messages are output to the **LogCat** tab of the IDE (the one with the "sideways" android robot symbol beside the **Console** tab at the bottom of the main IDE window). The Android system itself also outputs numerous log messages of its own to this tab as part of its installation and launch procedure for apps from the IDE. Thus you might find it useful to add a filter to the LogCat tab. This is easily done by selecting the tab and then the red plus sign. In the subsequent dialog enter any name you like for the **Filter Name:** field, and enter the text **System.out** in the **By Log Tag:** field. Then when you run your app, you can go to the **LogCat** tag and select the **System.out** filter and only those outputs will be shown on the window. Then if you have the emulator running but place it so that you can also see the **LogCat** tab output, then you can see your debug output as the interact with the app.

LogCat keeps the log entries from all of the runs in any session and so you might find it useful to clear the log screen before each new running of a project. The **Clear Log** symbol is the small "text page" icon with the red cross on it to the top right of the **LogCat** pane.