Module: Mobile Application development (Android)

Session 21: Event Handling

## Event Handling

## 1 Understanding Android Events

## 2 Using the android:onClick Resource

## 3 Event Listeners and Callback Methods

## 4 An Event Handling Example

## 5 Designing the User Interface

## 6 The Event Listener and Callback Method

## 7 Consuming Events

## Understanding Android Events

Events in Android can take a variety of different forms, but are usually generated in response to an external action. The most common form of events, particularly for devices such as tablets and smartphones, involve some form of interaction with the touch screen. Such events fall into the category of input events.

The Android framework maintains an event queue into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis. In the case of an input event such as a touch on the screen, the event is passed to the view positioned at the location on the screen where the touch took place. In addition to the event notification, the view is also passed a range of information (depending on the event type) about the nature of the event such as the coordinates of the point of contact between the user’s fingertip and the screen.

In order to be able to handle the event that it has been passed, the view must have in place an event listener. The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method. In order to be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback. For example, if a button is to respond to a click event (the equivalent to the user touching and releasing the button view as though clicking on a physical button) it must both register the View.onClickListener event listener (via a call to the target view’s setOnClickListener() method) and implement the corresponding onClick() callback method. In the event that a “click” event is detected on the screen at the location of the button view, the Android framework will call the onClick() method of that view when that event is removed from the event queue. It is, of course, within the implementation of the onClick() callback method that any tasks should be performed or other methods called in response to the button click.

## Using the android:onClick Resource

Before exploring event listeners in more detail it is worth noting that a shortcut is available when all that is required is for a callback method to be called when a user “clicks” on a button view in the user interface. Consider a user interface layout containing a button view named button1 with the requirement that when the user touches the button, a method called buttonClick() declared in the activity class is called. All that is required to implement this behavior is to write the buttonClick() method (which takes as an argument a reference to the view that triggered the click event) and add a single line to the declaration of the button view in the XML file. For example:

<Button

android:id="@+id/button1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="buttonClick"

android:text="Click me" />

This provides a simple way to capture click events. It does not, however, provide the range of options offered by event handlers, which are the topic of the rest of this chapter.

## Event Listeners and Callback Methods

In the example activity outlined later in this chapter the steps involved in registering an event listener and implementing the callback method will be covered in detail. Before doing so, however, it is worth taking some time to outline the event listeners that are available in the Android framework and the callback methods associated with each one.

* **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the onClick() callback method which is passed a reference to the view that received the event as an argument.
* **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the onLongClick() callback method which is passed the view that received the event as an argument.
* **onTouchListener** – Used to detect any form of contact with the touch screen including individual or multiple touches and gesture motions. Corresponding with the onTouch() callback, this topic will be covered in greater detail in the chapter entitled Android Touch and Multi-touch Event Handling. The callback method is passed the view that received the event and a MotionEvent object as arguments.
* **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the onCreateContextMenu() callback method. The callback is passed the menu, the view that received the event and a menu context object.
* **onFocusChangeListener** – Detects when focus moves away from the current view as the result of interaction with a track-ball or navigation key. Corresponds to the onFocusChange() callback method which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.
* **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the onKey() callback method. Passed as arguments are the view that received the event, the KeyCode of the physical key that was pressed and a KeyEvent object.

## An Event Handling Example

In the remainder of this chapter, we will work through the creation of a simple application designed to demonstrate the implementation of an event listener and callback method to detect when the user has clicked on a button. The code within the callback method will update a text view to indicate that the event has been processed.

Launch Eclipse and create an Android Application Project named EventExample with the appropriate package name and SDK selections. As with previous examples, request the creation of a blank activity and the use of the default launcher icons. On the New Blank Activity screen of the New Android Application wizard, set the Activity Name to EventExampleActivity, the Layout Name to activity\_event\_example and the Fragment Layout to fragment\_event\_example.

## Designing the User Interface

The user interface layout for the EventExampleActivity class in this example is to consist of a RelativeLayout view, a Button and a TextView as illustrated in Figure 1.

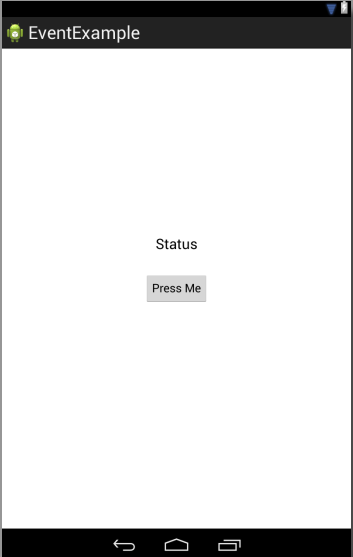
[](http://www.techotopia.com/index.php/Image:Android_event_handling_example_ui2.png)

Figure 1

|  |  |  |
| --- | --- | --- |
|  |  |  |

Locate and select the fragment\_event\_example.xml file created by Eclipse (located in the Package Explorer under EventExample -> res -> layout -> fragment\_event\_example.xml) and double click on it to load it into the editing panel. Switch from the Graphical Layout tool to the XML file using the tab at the bottom of the editing panel and delete the current content of the file. With a blank canvas, either use the Graphical Layout tool to design the user interface from Figure 17-1 (making sure to change the IDs of the Button and TextView objects to myButton and myTextView respectively), or directly enter the following XML into the editor:

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/myLayout"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent" >

<Button

android:id="@+id/myButton"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerHorizontal="true"

android:layout\_centerVertical="true"

android:text="@string/mybutton\_string" />

<TextView

android:id="@+id/myTextView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_above="@+id/myButton"

android:layout\_centerHorizontal="true"

android:layout\_marginBottom="41dp"

android:text="@string/mytextview\_string"

android:textAppearance="?android:attr/textAppearanceLarge" />

</RelativeLayout>

Within the Graphical Layout tool, right-click on the Button view and select the Edit Text… menu option. In the Resource Chooser dialog, click on the New String… button and in the Create New Android String dialog, enter Press Me into the String field and mybutton\_string into the New R.string field:

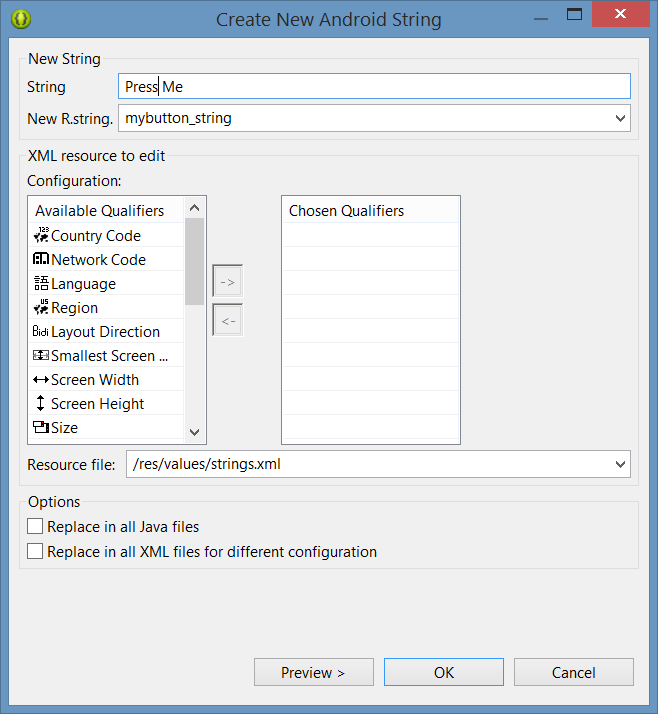
[](http://www.techotopia.com/index.php/Image:Android_44_create_string_resource.png)

Figure 2

Click on OK in the new android string dialog to create the new string resource and then again in the resource chooser dialog to assign the string to the button view.

Repeat these steps on the TextView object to create a new string resource named mytextview\_string with a string that reads “Status”. Be sure to save the file once the changes are complete.

With the user interface layout now completed, the next step is to register the event listener and callback method.

## The Event Listener and Callback Method

For the purposes of this example, an onClickListener needs to be registered for the myButton view. This is achieved by making a call to the setOnClickListener() method of the button view, passing through a new onClickListener object as an argument and implementing the onClick() callback method. Since this is a task that only needs to be performed when the activity is created, a good option is to override the onStart() lifecycle method of the EventExampleActivity class.

Within the Package Explorer panel, navigate to the EventExampleActivity.java file (src -> <package name> -> EventExampleActivity.java) and double click on it to load it into the code editor. Once loaded, locate implement the onStart() method to obtain a reference to the button view, register the event listener and implement the onClick() callback method:

package com.example.eventexample;

import android.app.Activity;

import android.app.ActionBar;

import android.app.Fragment;

import android.os.Bundle;

import android.view.LayoutInflater;

import android.view.Menu;

import android.view.MenuItem;

import android.view.View;

import android.view.ViewGroup;

import android.os.Build;

import android.widget.Button;

import android.widget.TextView;

public class EventExampleActivity extends Activity {

.

.

.

@Override

protected void onStart()

{

super.onStart();

Button button =

(Button)findViewById(R.id.myButton);

button.setOnClickListener(

new Button.OnClickListener() {

public void onClick(View v) {

}

}

);

}

.

.

.

}

The above code has now registered the event listener on the button and implemented the onClick() method. If the application were to be run at this point, however, there would be no indication that the event listener installed on the button was working since there is, as yet, no code implemented within the body of the onClick() callback method. The goal for the example is to have a message appear on the TextView when the button is clicked, so some further code changes need to be made:

@Override

protected void onStart()

{

super.onStart();

Button button =

(Button)findViewById(R.id.myButton);

button.setOnClickListener(

new Button.OnClickListener() {

public void onClick(View v) {

TextView myTextView =

(TextView) findViewById(R.id.myTextView);

myTextView.setText("Button clicked");

}

}

);

}

Complete this phase of the tutorial by compiling and running the application on either an AVD emulator or physical Android device. On touching and releasing the button view (otherwise known as “clicking”) the text view should change to display the “Button clicked” text.

## Consuming Events

The detection of standard clicks (as opposed to long clicks) on views is a very simple case of event handling. The example will now be extended to include the detection of long click events which occur when the user clicks and holds a view on the screen and, in doing so, cover the topic of event consumption.

Consider the code for the onClick() method in the above section of this chapter. The callback is declared as void and, as such, does not return a value to the Android framework after it has finished executing.

The onLongClick() callback method of the onLongClickListener interface, on the other hand, is required to return a Boolean value to the Android framework. The purpose of this return value is to indicate to the Android runtime whether the callback has consumed the event or not. If the callback returns a true value, the event is discarded by the framework. If, on the other hand, the callback returns a false value the Android framework will consider the event still to be active and will consequently pass it along to the next matching event listener that is registered on the same view.

As with many programming concepts this is, perhaps, best demonstrated with an example. The first step is to add an event listener and callback method for long clicks to the button view in the example activity:

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_event\_example);

Button button = (Button)findViewById(R.id.myButton);

button.setOnClickListener(

new Button.OnClickListener() {

public void onClick(View v) {

TextView myTextView =

(TextView)findViewById(R.id.myTextView);

myTextView.setText("Button clicked");

}

}

);

button.setOnLongClickListener(

new Button.OnLongClickListener() {

public boolean onLongClick(View v) {

TextView myTextView =

(TextView)findViewById(R.id.myTextView);

myTextView.setText("Long button click");

return true;

}

}

);

}

Clearly, when a long click is detected, the onLongClick() callback method will display “Long button click” on the text view. Note, however, that the callback method also returns a value of true to indicate that it has consumed the event. Compile and run the application and press and hold a fingertip over the button view until the “Long button click” text appears in the text view. On releasing the button, the text view continues to display the “Long button click” text indicating that the onClick() callback method was not called.

Next, modify the code such that the onLongClick() method now returns a false value:

button.setOnLongClickListener(

new Button.OnLongClickListener() {

public boolean onLongClick(View v) {

TextView myTextView = (TextView)findViewById(R.id.myTextView);

myTextView.setText("Long button click");

return false;

}

}

);

Once again, compile and run the application and perform a long click on the button until the long click message appears. Upon releasing the button this time, however, note that the onClick() callback is also triggered and the text changes to “Button click”. This is because the false value returned by the onLongClick() callback method indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In this case, the runtime ascertained that the onClickListener on the button was also interested in events of this type and subsequently called the onClick() callback method.

## Summary

A user interface is of little practical use if the views it contains do not do anything in response to user interaction. Android bridges the gap between the user interface and the back end code of the application through the concepts of event listeners and callback methods. The Android View class defines a set of event listeners, which can be registered on view objects. Each event listener also has associated with it a callback method.

When an event takes place on a view in a user interface, that event is placed into an event queue and handled on a first in, first out basis by the Android runtime. If the view on which the event took place has registered a listener that matches the type of event, the corresponding callback method is called. The callback method then performs any tasks required by the activity before returning. Some callback methods are required to return a Boolean value to indicate whether the event needs to be passed on to any other event listeners registered on the view or discarded by the system.