LAB 4 – Android Event Handling

At the end of this lab, students should be able to:

- Understand Android Events
- Apply event handling in Android applications
- Detect common gestures by using the AndroidGestureDetector class

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*.

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.

Event Listeners and Callback Methods

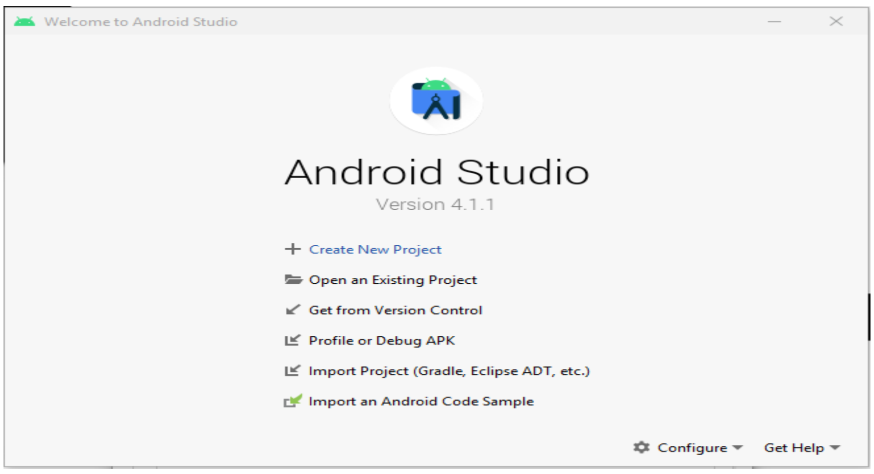
There are variety of event listeners that are available in the Android framework and the callback methods associated with each one. Below are two examples of event Listeners.

Event Listener	Description
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 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 as arguments the view that received the event and a MotionEvent object.

An Event Handling Example

a) Creating the Empty Activity Project

i. Go to File -> New Project

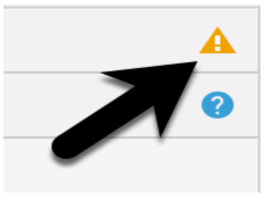


ii. Choose **Empty activity** template, then click Next

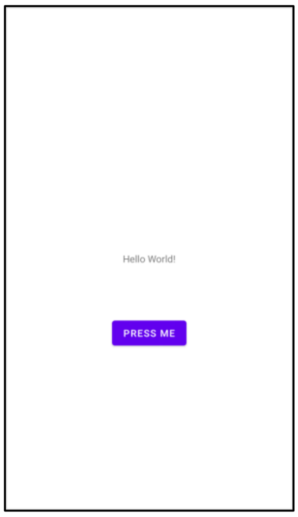
iii. Enter EventExample into the Name field and specify com.ebookfrenzy.eventexample as the package name.
Change the Minimum API level setting to API26: Android 8.0(Oreo) and the Language menu to Java.
Click Finish.

b) Designing the User Interface

i. Open activity_main.xml. Turn on the Autoconnect button if the Autoconnect button is off.
ii. Drag a button widget from the palette and move it at the bottom part of the existing TextView widget.
iii. Select the “Hello World” TextView widget ad use the Attributes panel to set the ID to *statusText.* Repeat this step to change the ID of the Button widget to *myButton*.
iv. Add any missing constraints by clicking the *Infer Constraints* button in the layout editor toolbar.
v. With the Button widget selected, use the Attributes panel to set the text property to Press Me. Using the yellow warning button located in the top right-hand corner of the Layout Editor (as Figure below), display the warnings list and click on the Fix button to extract the text string on the button to a resource named *press_me*:



vi. The user interface layout now should look like below figure.



c) The Event Listener and Callback Method

i. 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 location is the *onCreate()* method of the MainActivity class.

ii. Open MainActivity.java file, modify the code by adding the highlight code below:

```
package com.ebookfrenzy.eventexample;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_event_example);
Button button = findViewById(R.id.myButton);
button.setOnClickListener(
new Button.OnClickListener() {
public void onClick(View v) {
        }
}
);
 }
..}
```

iii. 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 as highlighted below:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_event_example);
Button button = findViewById(R.id.myButton);
button.setOnClickListener(
new Button.OnClickListener() {
public void onClick(View v) {
TextView statusText =
findViewById(R.id.statusText);
statusText.setText("Button clicked");
}
}
);
}
```

iv. 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.

d) 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.

i. The first step is to add an event listener and callback method for long clicks to the button view in the lab activity.

```
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_event_example);
Button button = findViewById(R.id.myButton);
button.setOnClickListener(
new Button.OnClickListener() {
public void onClick(View v) {
TextView statusText =
findViewById(R.id.statusText);
statusText.setText("Button clicked");
}});
button.setOnLongClickListener(
new Button.OnLongClickListener() {
public boolean onLongClick(View v) {
TextView statusText =
findViewById(R.id.statusText);
statusText.setText("Long button click");
return true;
}
}
);
}
}
```

ii. 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. Run the application and press and hold 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 listener code was not called.

iii. Next, modify the code so that the onLongClick listener now returns a *false* value.

iv. 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 listener is also triggered and the text changes to "Button click". This is because the false value returned by the *onLongClick* listener code 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* listener code.

e) Implementing Common Gesture Detection

When a user interacts with the display of an Android device, the *onTouchEvent()* method of the currently active application is called by the system and passed MotionEvent objects containing data about the user's contact with the screen. This data can be interpreted to identify if the motion on the screen matches a common gesture such as a tap or a swipe.

i. Edit MainActivity.java file by adding highlighted text below:

```
.
.
import android.view.GestureDetector;
import android.view.MotionEvent;

public class MainActivity extends AppCompatActivity
implements GestureDetector.OnGestureListener,
GestureDetector.OnDoubleTapListener
{
        private TextView gestureText;

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

setContentView(R.layout.activity_main);
gestureText = findViewById(R.id.statusText);
}
                ..}
```

ii. Add on the corresponding methods as below:

```
@Override
public boolean onFling(MotionEvent event1, MotionEvent event2,
float velocityX, float velocityY) {
gestureText.setText("onFling");
return true;
}
@Override
public void onLongPress(MotionEvent event) {
gestureText.setText("onLongPress");
}
@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
float distanceX, float distanceY) {
gestureText.setText("onScroll");
return true;
                }

@Override
public boolean onSingleTapConfirmed(MotionEvent event) {
gestureText.setText("onSingleTapConfirmed");
return true;
                }
@Override
public boolean onDoubleTap(MotionEvent event) {
gestureText.setText("onDoubleTap");
return true;
                }

@Override
public boolean onDown(MotionEvent event) {
gestureText.setText ("onDown");
return true;
}

@Override
public void onShowPress(MotionEvent event) {
gestureText.setText("onShowPress");
}

@Override
public boolean onDoubleTapEvent(MotionEvent event) {
gestureText.setText("onDoubleTapEvent");
return true;
}

@Override
public boolean onSingleTapUp(MotionEvent event) {
gestureText.setText("onSingleTapUp");
return true;
}
```

f) Creating the GestureDetectorCompat Instance

i. With the activity class now updated to implement the listener interfaces, the next step is to create an instance of the GestureDetectorCompat class. Since we also want to detect double taps, the code also needs to call the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance:

```
.
.
import android.view.MotionEvent;
import androidx.core.view.GestureDetectorCompat;
.
.
private TextView gestureText;
private GestureDetectorCompat gDetector;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_common_gestures);
gestureText = findViewById(R.id.gestureStatusText);
this.gDetector = new GestureDetectorCompat(this,this);
gDetector.setOnDoubleTapListener(this);
}
..}
```

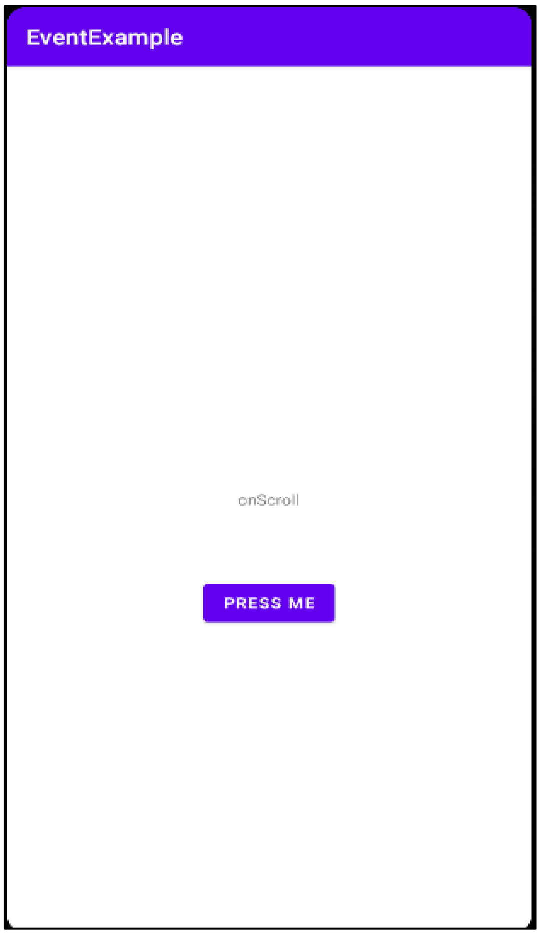
g) Implementing the onTouchEvent() Method

i. If the application were to be compiled and run at this point, nothing would happen if gestures were performed on the device display. This is because no code has been added to intercept touch events and to pass them through to the GestureDetectorCompat instance. In order to achieve this, it is necessary to override the *onTouchEvent()* method within the activity class and implement it such that it calls the *onTouchEvent()* method of the GestureDetectorCompat instance. Remaining in the *MainActivity.java* file, therefore, implement this method so that it reads as follows:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
this.gDetector.onTouchEvent(event);
// Be sure to call the superclass implementation
return super.onTouchEvent(event);
}
```

h) Testing the Application

i. Compile and run the application on either a physical Android device or an AVD emulator. Once launched, experiment with swipes, presses, scrolling motions and double and single taps. Note that the text view updates to reflect the events as illustrated in Figure below:


EventExample
onScroll
PRESS ME