- Much has been covered in the previous chapters relating to the design of user interfaces for Android applications.

- An area that has yet to be covered, however, involves the way in which a user's interaction with the user interface triggers the underlying activity to perform a task.

- In other words, we know from the previous chapters how to create a user interface containing a button view, but not how to make something happen within the application when it is touched by the user.

- The primary objective of this chapter, therefore, is to provide an overview of event handling in Android applications together with an Android Studio based example project.

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.

- As will be outlined in later chapters, the onClick property also has limitations in layouts involving fragments.

- When working within Android Studio Layout Editor, the onClick property can be found and configured in the Attributes panel when a suitable view type is selected in the device screen layout.

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 as an argument the view that received the event.

- 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 later in this chapter.

  The callback method is passed as arguments the view that received the event and a MotionEvent object.

- 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 an Android Studio project designed to demonstrate the implementation of an event listener and corresponding 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.

- Select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.

- Enter EventExample into the Name field and specify edu.niu.your_Z-ID.eventexample as the package name.

- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Designing the User Interface

- The user interface layout for the MainActivity class in this example is to consist of a ConstraintLayout, a Button and a TextView as illustrated below in Figure 11-1.

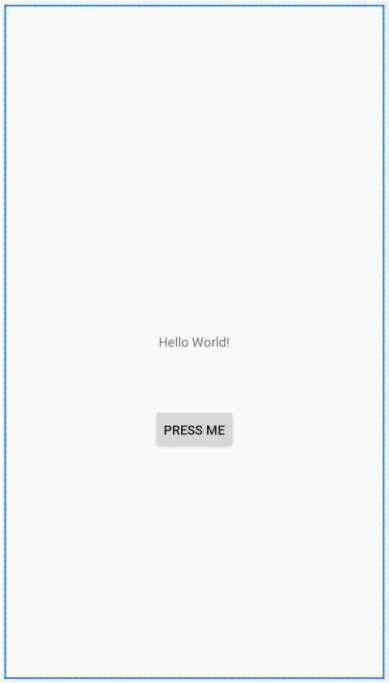

Figure 11-1

- Locate and select the activity_main.xml file created by Android Studio (located in the Project tool window under app > res > layouts) and double-click on it to load it into the Layout Editor tool.

- Make sure that Autoconnect is enabled, then drag a Button widget from the palette and move it so that it is positioned in the horizontal center of the layout and beneath the existing TextView widget.

- When correctly positioned, drop the widget into place so that appropriate constraints are added by the autoconnect system.

- Select the “Hello World!” TextView widget and use the Attributes panel to set the ID to statusText.

- Repeat this step to change the ID of the Button widget to myButton.

- Add any missing constraints by clicking on the Infer Constraints button in the layout editor toolbar.

- 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 (Figure 11-2) display the warnings list and click on the Fix button to extract the text string on the button to a resource named press_me:

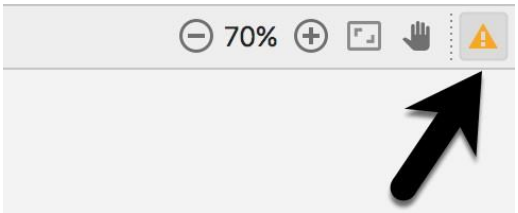

Figure 11-2

- 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 location is the onCreate() method of the MainActivity class.
- If the MainActivity.java file is already open within an editor session, select it by clicking on the tab in the editor panel.
- Alternatively locate it within the Project tool window by navigating to (**app** > **java** > **edu.niu.your_Z-ID. eventexample** > **MainActivity**) and double-click on it to load it into the code editor.
- Once loaded, locate the template onCreate() method and modify it to obtain a reference to the button view, register the event listener and implement the onClick() callback method:

```
package edu.niu.your_Z-ID.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)
          {
          }
        }
      );
    }
    .
    .
}
```

- 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 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");
      }
    }
  );
}
```

- 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 code assigned to the onLongClickListener, 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 or not the callback has consumed the event.

- 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 = 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;
```

```
        }
      }
    );
}
```

- 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.

- Next, modify the code so that the onLongClick listener now returns a false value:

```
button.setOnLongClickListener(
  new Button.OnLongClickListener()
  {
    public boolean onLongClick(View v)
    {
      TextView myTextView = 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 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.

- Most Android based devices use a touch screen as the primary interface between user and device.

- The previous chapter introduced the mechanism by which a touch on the screen translates into an action within a running Android application.

- There is, however, much more to touch event handling than responding to a single finger tap on a view object.

- Most Android devices can, for example, detect more than one touch at a time.

- Nor are touches limited to a single point on the device display.

- Touches can, of course, be dynamic as the user slides one or more points of contact across the surface of the screen.

- Touches can also be interpreted by an application as a gesture.

- Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook, or how a pinching motion can be used to zoom in and out of an image displayed on the screen.

- This chapter will explain the handling of touches that involve motion and explore the concept of intercepting multiple concurrent touches.

- The topic of identifying distinct gestures will be covered in the next chapter.

Intercepting Touch Events

- Touch events can be intercepted by a view object through the registration of an onTouchListener event listener and the implementation of the corresponding onTouch() callback method.

- The following code, for example, ensures that any touches on a ConstraintLayout view instance named myLayout result in a call to the onTouch() method:

```
myLayout.setOnTouchListener(
  new ConstraintLayout.OnTouchListener()
  {
    public boolean onTouch(View v, MotionEvent m)
    {
      // Perform tasks here
      return true;
    }
  }
);
```

- As indicated in the code example, the onTouch() callback is required to return a Boolean value indicating to the Android runtime system whether or not the event should be passed on to other event listeners registered on the same view or discarded.

- The method is passed both a reference to the view on which the event was triggered and an object of type MotionEvent.

The MotionEvent Object

- The MotionEvent object passed through to the onTouch() callback method is the key to obtaining information about the event.

- Information contained within the object includes the location of the touch within the view and the type of action performed.

- The MotionEvent object is also the key to handling multiple touches.

Understanding Touch Actions

- An important aspect of touch event handling involves being able to identify the type of action performed by the user.

- The type of action associated with an event can be obtained by making a call to the getActionMasked() method of the MotionEvent object which was passed through to the onTouch() callback method.

- When the first touch on a view occurs, the MotionEvent object will contain an action type of ACTION_DOWN together with the coordinates of the touch.

- When that touch is lifted from the screen, an ACTION_UP event is generated.

- Any motion of the touch between the ACTION_DOWN and ACTION_UP events will be represented by ACTION_ MOVE events.

- When more than one touch is performed simultaneously on a view, the touches are referred to as pointers.

- In a multi-touch scenario, pointers begin and end with event actions of type ACTION_POINTER_DOWN and ACTION_POINTER_UP respectively.

- In order to identify the index of the pointer that triggered the event, the getActionIndex() callback method of the MotionEvent object must be called.

Handling Multiple Touches

- Earlier in this chapter, we began exploring event handling within the narrow context of a single touch event.

- In practice, most Android devices possess the ability to respond to multiple consecutive touches (though it is important to note that the number of simultaneous touches that can be detected varies depending on the device).

- As previously discussed, each touch in a multi-touch situation is considered by the Android framework to be a pointer.

- Each pointer, in turn, is referenced by an index value and assigned an ID.

- The current number of pointers can be obtained via a call to the getPointerCount() method of the current MotionEvent object.

- The ID for a pointer at a particular index in the list of current pointers may be obtained via a call to the MotionEvent getPointerId() method.

- For example, the following code excerpt obtains a count of pointers and the ID of the pointer at index 0:

```
public boolean onTouch(View v, MotionEvent m)
{
  int pointerCount = m.getPointerCount();
  int pointerId = m.getPointerId(0);
  return true;
}
```

- Note that the pointer count will always be greater than or equal to 1 when the onTouch listener is triggered (since at least one touch must have occurred for the callback to be triggered).

- A touch on a view, particularly one involving motion across the screen, will generate a stream of events before the point of contact with the screen is lifted.

- As such, it is likely that an application will need to track individual touches over multiple touch events.

- While the ID of a specific touch gesture will not change from one event to the next, it is important to keep in mind that the index value will change as other touch events come and go.

- When working with a touch gesture over multiple events, therefore, it is essential that the ID value be used as the touch reference in order to make sure the same touch is being tracked.

- When calling methods that require an index value, this should be obtained by converting the ID for a touch to the corresponding index value via a call to the findPointerIndex() method of the MotionEvent object.

An Example Multi-Touch Application

- The example application created in the remainder of this chapter will track up to two touch gestures as they move across a layout view.

- As the events for each touch are triggered, the coordinates, index and ID for each touch will be displayed on the screen.

- Select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.

- Enter MotionEvent into the Name field and specify edu.niu.your_Z-ID.motionevent as the package name.

- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

Designing the Activity User Interface

- The user interface for the application's sole activity is to consist of a ConstraintLayout view containing two TextView objects.

- Within the Project tool window, navigate to app > res > layout and double-click on the activity_main.xml layout resource file to load it into the Android Studio Layout Editor tool.

- Select and delete the default "Hello World!" TextView widget and then, with autoconnect enabled, drag and drop a new TextView widget so that it is centered horizontally and positioned at the 16dp margin line on the top edge of the layout:

Figure 11-3

- Drag a second TextView widget and position and constrain it so that it is distanced by a 32dp margin from the bottom of the first widget:

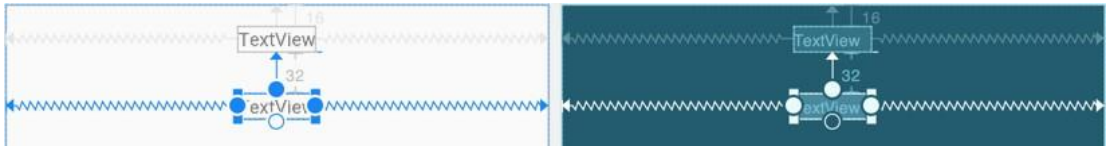

Figure 11-4

- Using the Attributes tool window, change the IDs for the TextView widgets to textView1 and textView2 respectively.

- Change the text displayed on the widgets to read "Touch One Status" and "Touch Two Status" and extract the strings to resources using the warning button in the top right-hand corner of the Layout Editor.

- Select the ConstraintLayout entry in the Component Tree and use the Attributes panel to change the ID to activity_main.

Implementing the Touch Event Listener

- In order to receive touch event notifications it will be necessary to register a touch listener on the layout view within the onCreate() method of the MainActivity activity class.

- Select the MainActivity.java tab from the Android Studio editor panel to display the source code.

- Within the onCreate() method, add code to identify the ConstraintLayout view object, register the touch listener and implement code which, in this case, is going to call a second method named handleTouch() to which is passed the MotionEvent object:

```
package edu.niu.your_Z-ID.motionevent;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import androidx.constraintlayout.widget.ConstraintLayout;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity
{
  @Override
```

```
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ConstraintLayout myLayout = findViewById(R.id.activity_main);

    myLayout.setOnTouchListener(
      new ConstraintLayout.OnTouchListener()
      {
        public boolean onTouch(View v, MotionEvent m)
        {
          handleTouch(m);
          return true;
        }
      }
    );
  }
  .
  .
}
```

- The final task before testing the application is to implement the handleTouch() method called by the listener. The code for this method reads as follows:

```
void handleTouch(MotionEvent m)
{
  TextView textView1 = findViewById(R.id.textView1);
  TextView textView2 = findViewById(R.id.textView2);

  int pointerCount = m.getPointerCount();

  for (int i = 0; i < pointerCount; i++)
  {
    int x = (int) m.getX(i);
    int y = (int) m.getY(i);
    int id = m.getPointerId(i);
    int action =  m.getActionMasked();
    int actionIndex = m.getActionIndex();
    String actionString;

    switch (action)
    {
      case MotionEvent.ACTION_DOWN:
        actionString = "DOWN";
        break;
      case MotionEvent.ACTION_UP:
        actionString = "UP";
        break;
      case MotionEvent.ACTION_POINTER_DOWN:
        actionString = "PNTR DOWN";
```

```
          break;
        case MotionEvent.ACTION_POINTER_UP:
          actionString = "PNTR UP";
          break;
        case MotionEvent.ACTION_MOVE:
          actionString = "MOVE";
          break;
        default:
          actionString = "";
      }

      String touchStatus = "Action: " + actionString + " Index: " +
        actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

      if (id == 0)
        textView1.setText(touchStatus);
      else
        textView2.setText(touchStatus);
    }
}
```

- Before compiling and running the application, it is worth taking the time to walk through this code systematically to highlight the tasks that are being performed.

- The code begins by obtaining references to the two TextView objects in the user interface and identifying how many pointers are currently active on the view:

```
TextView textView1 = findViewById(R.id.textView1);
TextView textView2 = findViewById(R.id.textView2);

int pointerCount = m.getPointerCount();
```

- Next, the pointerCount variable is used to initiate a for loop which performs a set of tasks for each active pointer.

- The first few lines of the loop obtain the X and Y coordinates of the touch together with the corresponding event ID, action type and action index. Lastly, a string variable is declared:

```
for (int i = 0; i < pointerCount; i++)
{
  int x = (int) m.getX(i);
  int y = (int) m.getY(i);
  int id = m.getPointerId(i);
  int action =  m.getActionMasked();
  int actionIndex = m.getActionIndex();
  String actionString;
```

- Since action types equate to integer values, a switch statement is used to convert the action type to a more meaningful string value, which is stored in the previously declared actionString variable:

```
switch (action)
{
  case MotionEvent.ACTION_DOWN:
    actionString = "DOWN";
    break;
  case MotionEvent.ACTION_UP:
    actionString = "UP";
    break;
  case MotionEvent.ACTION_POINTER_DOWN:
    actionString = "PNTR DOWN";
    break;
  case MotionEvent.ACTION_POINTER_UP:
    actionString = "PNTR UP";
    break;
  case MotionEvent.ACTION_MOVE:
    actionString = "MOVE";
    break;
  default:
    actionString = "";
}
```

- Finally, the string message is constructed using the actionString value, the action index, touch ID and X and Y coordinates.

- The ID value is then used to decide whether the string should be displayed on the first or second TextView object:

```
String touchStatus = "Action: " + actionString + " Index: "
  + actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

if (id == 0)
  textView1.setText(touchStatus);
else
  textView2.setText(touchStatus);
```

Running the Example Application

- Compile and run the application and, once launched, experiment with single and multiple touches on the screen and note that the text views update to reflect the events as illustrated in Figure 11-5.

- When running on an emulator, multiple touches may be simulated by holding down the Ctrl (Cmd on macOS) key while clicking the mouse button:

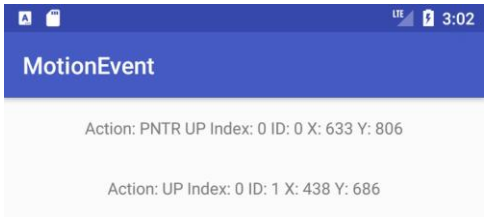

Figure 11-5

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.
- This code 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.
- Having covered the basics of event handling, the next chapter will explore in some depth the topic of touch events with a particular emphasis on handling multiple touches.
- Activities receive notifications of touch events by registering an onTouchListener event listener and implementing the onTouch() callback method which, in turn, is passed a MotionEvent object when called by the Android runtime.
- This object contains information about the touch such as the type of touch event, the coordinates of the touch and a count of the number of touches currently in contact with the view.
- When multiple touches are involved, each point of contact is referred to as a pointer with each assigned an index and an ID.
- While the index of a touch can change from one event to another, the ID will remain unchanged until the touch ends.
- This chapter has worked through the creation of an example Android application designed to display the coordinates and action type of up to two simultaneous touches on a device display.
- The next chapter will look further at touch screen event handling through the implementation of gesture recognition.