IN721 2019 Practical 2.1 – Event Handlers

In this practical you will explore the Android model for attaching event handler code to controls. As discussed in lecture, the general process is:

* Write an inner class that implements an appropriate interface.
* Create an instance of that class.
* Get a reference to the control.
* Invoke one of the control’s event setting methods, passing in the object instance.

## Task 1: Multiple Button Click Events (Can be done on a phone)

In lecture, we looked carefully at how to bind code to a button’s click event. We created an inner class that implemented the interface *OnClickListener*, and we passed an instance of that class to a button’s *setOnClickListener* method.

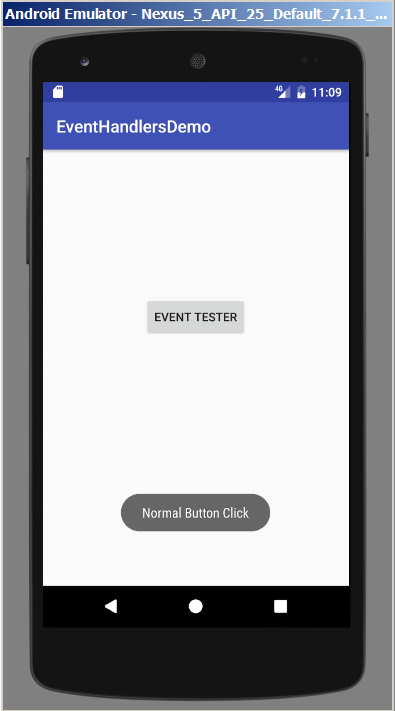
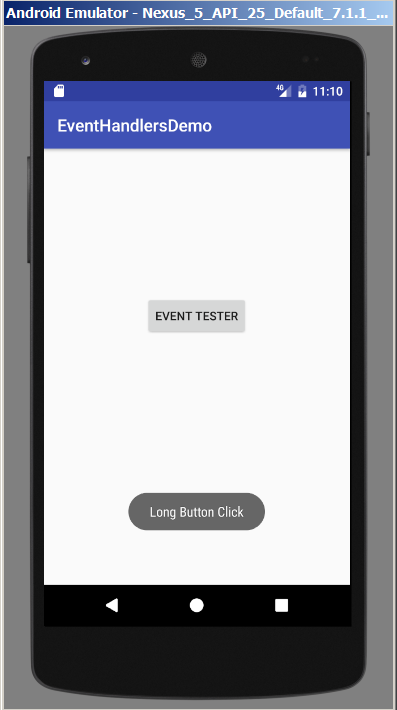
Android buttons actually have two distinct click events – the normal one that we have seen, and a second event for “long clicks”. The long click event is raised when the user clicks on a button and holds the mouse down for several seconds, or, when using a touch screen, presses and holds their finger on a button for several seconds.

We can compare the two click events:

|  |  |  |
| --- | --- | --- |
|  | **Normal Click** | **Long Click** |
| **Interface** | View.OnClickListener | View.OnLongClickListener |
| **Associated Interface method** | void onClick(View v) | boolean onLongClick(View v) |
| **Event setter method of the button** | setOnClickListener | setOnLongClickListener |

***Note that while a normal click handler returns void, a long click handler returns a boolean.*** This is common; many Android event handlers return boolean. The convention is that if the handler has ***completely*** processed the event it should return true, and if the handler has ***not completely*** processed the event, it should return false. If a handler returns false, the system will pass the event up the computation hierarchy to see if some other object wants to deal with it further. We will see this process in action when we start listening for specific keystrokes in the second task of this practical. Our handlers will field all the keyboard input, but when they detect some keystroke they can’t completely deal with, they will return false, allowing the keystroke to continue up the application hierarchy. In the first task, however, your onLongClick method will completely process the click event, so it should always return true.

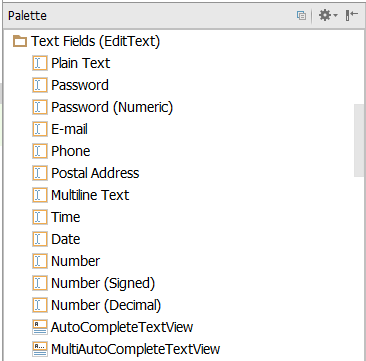
Build an Android app with a single button that responds differently to normal clicks and long clicks. Use Toast to distinguish between the two events. Two screen shots of my solution are:

On click and release On click and hold

## Task 2: Using an EditText

We frequently want users to enter some text into our application. The control to use for this with Android is the EditText (equivalent to a TextBox in .NET). In Design view, multiple versions of the EditText control are shown in the palette (see below). The different versions each have a different preset value for their ***inputType*** property, which determines the type of soft keyboard the device will display when the user selects the field and also, in some cases, the way the entered text is displayed. For example, if the inputType is textEmailAddress, the soft keyboard will include the @ key. If the inputType is phone, the keypad will be presented. If the inputType is textPassword, the entered text characters will be displayed as dots. You can find detailed information about the various flavours of EditText object in the Android documentation.



EditText controls support a number of events. As with buttons, to bind code to an EditText event, you gain a reference to the control and call an appropriate event setter, passing in an instance of an object that implements an appropriate interface. In the current task, we want to respond to user input ***from the hard keyboard[[1]](#footnote-1)*** (the computer keyboard when working in the emulator). The relevant components are:

|  |  |
| --- | --- |
| **Interface** | View.OnKeyListener |
| **Interface method[[2]](#footnote-2)** | boolean onKey(View view, int keyCode, KeyEvent keyEvent) |
| **Event setter method of the EditText control** | setOnKeyListener |

The three arguments of the onKey method are passed in by the system when the event is raised, and you access them as per normal within your handler code. We discuss each argument in detail:

|  |  |
| --- | --- |
| **View view** | * This is the control that raised the event. That is, the EditText into which your user is typing. * You can get the resource id of this control by calling its getId() method. That is, *view.getId()*. * Can you think of a reason that you are frequently going to need to get the resource id of the receiving control? (Hint: what data do you pass to findViewById?) |
| **int keyCode** | * A numeric code for the key that has been pressed. * To specify a particular key's code it is considered best practice to use the symbolic constants defined in the KeyEvent class. * For example, to check if the user has hit the enter key, you can say:      * For the full list of keyCode constants, see http://developer.android.com/reference/android/view/KeyEvent.html |
| **KeyEvent keyEvent** | * This is a data bundle that contains information about the event (equivalent to EventArgs in .NET). The most commonly used piece of data in the KeyEvent is the ***action***. A key press is actually composed of a several parts, each of which is an action. For example, the key down is an action and the key up is a separate action. Both will raise the onKey event. To determine which action has actually caused your onKey handler to fire, you can query the KeyEvent argument via its *getAction()* method. For example, if you wished to pop up a different Toast for key down and key up, you could write something like this: |

## Task 2.1 (emulator required)

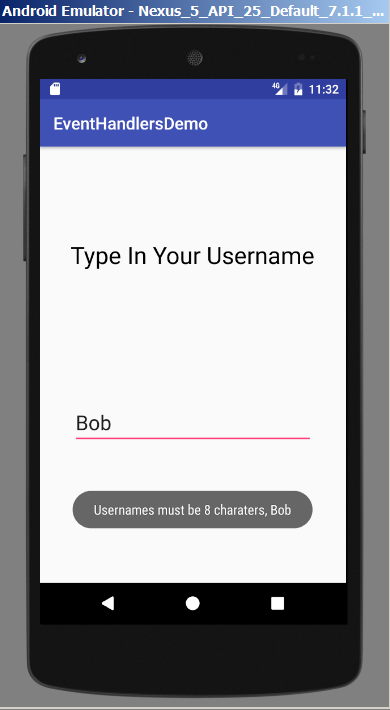
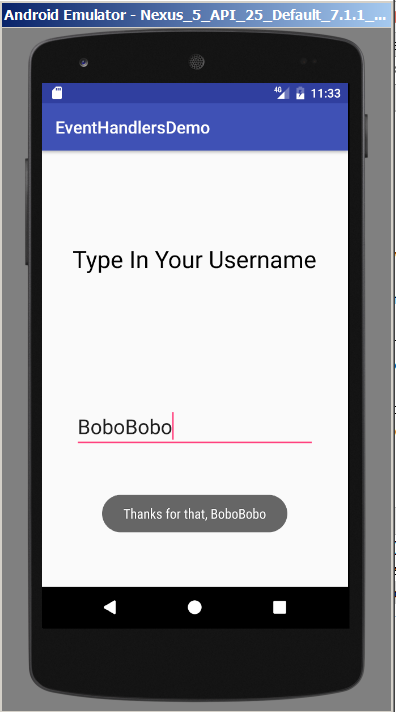
Build an app with an EditText. The user types into the control. When they type @ (i.e. an at-sign), pop up a Toast that says “Don’t Type @”.



You may find that your Toast actually pops up twice when the user types @. Why do you think that might be? How could you fix it? Modify your code to solve this problem.

## Task 2.2 (can be done on a phone)

Write an app that asks your user to specify a username. They should type their chosen username into an EditText, followed by the Enter key. If the username is not exactly 8 characters long, pop up a Toast saying that usernames must be 8 characters long. If the username is exactly 8 characters, say thank you.

Two things to think about before starting this task:

1. After using findViewById to get a reference to your EditText control, you can call its getText() method to retrieve its contents. Note however that getText() returns an object of type ***Editable***, not of type ***String***. An Editable is simply a string whose value can be changed (unlike traditional String variables, which are immutable). If you need to, you can convert an Editable into a String via ToString(). However, you can use the Editable directly in concatenation with literal strings and/or as an argument to Toast.makeText.
2. In this exercise, pay careful attention to the impact of returning true or returning false from the onKey method that you write. Remember that when you return true, you state that you have ***completed*** ***all necessary*** ***processing*** of the key that was pressed. This means it will not be passed further up the computation hierarchy. For example, it will not be passed to the object which is responsible for displaying it on the app screen. In contrast, when you return false, you state that you have ***not completed all the necessary processing*** of the key press and the system will continue to work with it. If you find the length of the username doesn’t seem to be behaving, check that you are not inadvertently allowing the system to write the Enter key into the EditText (effectively adding the Enter key to the username).

1. In some situations, the soft keyboard is more complicated. We will deal with this later. [↑](#footnote-ref-1)
2. Depending on which version of the API you are using, the variable names inserted by AS may be different, but the data types will be the same. [↑](#footnote-ref-2)