Input Events

IN THIS DOCUMENT

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so.

Within the various View classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the `onTouchEvent()` method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every View object in order to handle such an event would not be practical. This is why the View class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called event listeners (#EventListeners), are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the `Button` (/reference/android/widget/Button.html) class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class event handlers (#EventHandlers).

Event Listeners

An event listener is an interface in the `View` (/reference/android/view/View.html) class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.

Included in the event listener interfaces are the following callback methods:

`onClick()`
: From `View.OnClickListener`. This is called when the user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.

`onLongClick()`
: From `View.OnLongClickListener`. This is called when the user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).

`onFocusChange()`
: From `View.OnFocusChangeListener`. This is called when the user navigates onto or away from the item, using the navigation-keys or trackball.

`onKey()`
: From `View.OnKeyListener`. This is called when the user is focused on the item and presses or releases a hardware key on the device.

`onTouch()`
: From `View.OnTouchListener`. This is called when the user performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen (within the bounds of the item).

`onCreateContextMenu()`
: From `View.OnCreateContextMenuListener`. This is called when a Context Menu is being built (as the result of a sustained "long click"). See the discussion on context menus in the Menus developer guide.

These methods are the sole inhabitants of their respective interface. To define one of these methods and handle your events, implement the nested interface in your Activity or define it as an anonymous class. Then, pass an instance of your implementation to the respective `View.set...Listener()` method. (E.g., call `setOnClickListener()` (/reference/android/view/View.html#setOnClickListener(android.view.View.OnClickListener)) and pass it your implementation of the `OnClickListener` (/reference/android/view/View.OnClickListener.html).)

The example below shows how to register an on-click listener for a Button.

```
// Create an anonymous implementation of OnClickListener
private OnClickListener mCorkyListener = new OnClickListener() {
    public void onClick(View v) {
      // do something when the button is clicked
    }
};

protected void onCreate(Bundle savedValues) {
    ...
    // Capture our button from layout
    Button button = (Button)findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(mCorkyListener);
    ...
}
```

You may also find it more convenient to implement OnClickListener as a part of your Activity. This will avoid the extra class load and object allocation. For example:

```
public class ExampleActivity extends Activity implements OnClickListener {
    protected void onCreate(Bundle savedValues) {
        ...
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implement the OnClickListener callback
    public void onClick(View v) {
      // do something when the button is clicked
    }
    ...
}
```

Notice that the `onClick()` callback in the above example has no return value, but some other event listener methods must return a boolean. The reason depends on the event. For the few that do, here's why:

- `onLongClick()` - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return *true* to indicate that you have handled the event and it should stop here; return *false* if you have not handled it and/or the event should continue to any other on-click listeners.

- `onKey()` - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return *true* to indicate that you have handled the event and it should stop here; return *false* if you have not handled it and/or the event should continue to any other on-key listeners.
- `onTouch()` - This returns a boolean to indicate whether your listener consumes this event. The important thing is that this event can have multiple actions that follow each other. So, if you return *false* when the down action event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event. Thus, you will not be called for any other actions within the event, such as a finger gesture, or the eventual up action event.

Remember that hardware key events are always delivered to the View currently in focus. They are dispatched starting from the top of the View hierarchy, and then down, until they reach the appropriate destination. If your View (or a child of your View) currently has focus, then you can see the event travel through the `dispatchKeyEvent()` (/reference/android/view/View.html#dispatchKeyEvent(android.view.KeyEvent)) method. As an alternative to capturing key events through your View, you can also receive all of the events inside your Activity with `onKeyDown()` (/reference/android/app/Activity.html#onKeyDown(int, android.view.KeyEvent)) and `onKeyUp()` (/reference/android/app/Activity.html#onKeyUp(int, android.view.KeyEvent)).

Also, when thinking about text input for your application, remember that many devices only have software input methods. Such methods are not required to be key-based; some may use voice input, handwriting, and so on. Even if an input method presents a keyboard-like interface, it will generally **not** trigger the `onKeyDown()` (/reference/android/app/Activity.html#onKeyDown(int, android.view.KeyEvent)) family of events. You should never build a UI that requires specific key presses to be controlled unless you want to limit your application to devices with a hardware keyboard. In particular, do not rely on these methods to validate input when the user presses the return key; instead, use actions like `IME_ACTION_DONE` (/reference/android/view/inputmethod/EditorInfo.html#IME_ACTION_DONE) to signal the input method how your application expects to react, so it may change its UI in a meaningful way. Avoid assumptions about how a software input method should work and just trust it to supply already formatted text to your application.

> **Note:** Android will call event handlers first and then the appropriate default handlers from the class definition second. As such, returning *true* from these event listeners will stop the propagation of the event to other event listeners and will also block the callback to the default event handler in the View. So be certain that you want to terminate the event when you return *true*.

Event Handlers

If you're building a custom component from View, then you'll be able to define several callback methods used as default event handlers. In the document about Custom Components (/guide/topics/ui/custom-components.html), you'll learn see some of the common callbacks used for event handling, including:

- `onKeyDown(int, KeyEvent)` - Called when a new key event occurs.
- `onKeyUp(int, KeyEvent)` - Called when a key up event occurs.
- `onTrackballEvent(MotionEvent)` - Called when a trackball motion event occurs.
- `onTouchEvent(MotionEvent)` - Called when a touch screen motion event occurs.
- `onFocusChanged(boolean, int, Rect)` - Called when the view gains or loses focus.

There are some other methods that you should be aware of, which are not part of the View class, but can directly impact the way you're able to handle events. So, when managing more complex events inside a layout, consider these other methods:

- `Activity.dispatchTouchEvent(MotionEvent)` - This allows your `Activity` to intercept all touch events before they are dispatched to the window.

- ViewGroup.onInterceptTouchEvent(MotionEvent) - This allows a ViewGroup to watch events as they are dispatched to child Views.
- ViewParent.requestDisallowInterceptTouchEvent(boolean) - Call this upon a parent View to indicate that it should not intercept touch events with onInterceptTouchEvent(MotionEvent).

Touch Mode

When a user is navigating a user interface with directional keys or a trackball, it is necessary to give focus to actionable items (like buttons) so the user can see what will accept input. If the device has touch capabilities, however, and the user begins interacting with the interface by touching it, then it is no longer necessary to highlight items, or give focus to a particular View. Thus, there is a mode for interaction named "touch mode."

For a touch-capable device, once the user touches the screen, the device will enter touch mode. From this point onward, only Views for which isFocusableInTouchMode() (/reference/android/view/View.html#isFocusableInTouchMode()) is true will be focusable, such as text editing widgets. Other Views that are touchable, like buttons, will not take focus when touched; they will simply fire their on-click listeners when pressed.

Any time a user hits a directional key or scrolls with a trackball, the device will exit touch mode, and find a view to take focus. Now, the user may resume interacting with the user interface without touching the screen.

The touch mode state is maintained throughout the entire system (all windows and activities). To query the current state, you can call isInTouchMode() (/reference/android/view/View.html#isInTouchMode()) to see whether the device is currently in touch mode.

Handling Focus

The framework will handle routine focus movement in response to user input. This includes changing the focus as Views are removed or hidden, or as new Views become available. Views indicate their willingness to take focus through the isFocusable() (/reference/android/view/View.html#isFocusable()) method. To change whether a View can take focus, call setFocusable() (/reference/android/view/View.html#setFocusable(boolean)). When in touch mode, you may query whether a View allows focus with isFocusableInTouchMode() (/reference/android/view/View.html#isFocusableInTouchMode()). You can change this with setFocusableInTouchMode() (/reference/android/view/View.html#setFocusableInTouchMode(boolean)).

Focus movement is based on an algorithm which finds the nearest neighbor in a given direction. In rare cases, the default algorithm may not match the intended behavior of the developer. In these situations, you can provide explicit overrides with the following XML attributes in the layout file: *nextFocusDown*, *nextFocusLeft*, *nextFocusRight*, and *nextFocusUp*. Add one of these attributes to the View *from* which the focus is leaving. Define the value of the attribute to be the id of the View *to* which focus should be given. For example:

```
<LinearLayout
    android:orientation="vertical"
    ... >
  <Button android:id="@+id/top"
          android:nextFocusUp="@+id/bottom"
          ... />
  <Button android:id="@+id/bottom"
          android:nextFocusDown="@+id/top"
          ... />
</LinearLayout>
```

Ordinarily, in this vertical layout, navigating up from the first Button would not go anywhere, nor would navigating down from the second Button. Now that the top Button has defined the bottom one as the *nextFocusUp* (and vice versa), the navigation focus will cycle from top-to-bottom and bottom-to-top.

If you'd like to declare a View as focusable in your UI (when it is traditionally not), add the `android:focusable` XML attribute to the View, in your layout declaration. Set the value *true*. You can also declare a View as focusable while in Touch Mode with `android:focusableInTouchMode`.

To request a particular View to take focus, call `requestFocus()` (/reference/android/view/View.html#requestFocus()).

To listen for focus events (be notified when a View receives or looses focus), use `onFocusChange()` (/reference/android/view/View.OnFocusChangeListener.html#onFocusChange(android.view.View, boolean)), as discussed in the Event Listeners (#EventListeners) section, above.