



The
University
Of
Sheffield.



COM4510/6510

Software Development for Mobile Devices

Lab 2: Layouts

Dr Po Yang

The University of Sheffield

po.yang@sheffield.ac.uk



The
University
Of
Sheffield.

Layouts

<https://developer.android.com/guide/topics/ui/declaring-layout>

- A layout defines the visual structure for a user interface, such as the UI for an activity or app widget.
- You can declare a layout in two ways:
 - Declare UI elements in XML via View classes and subclasses for widgets and layouts
- Instantiate layout elements at runtime
 - You can create **View** and **ViewGroup** objects (and manipulate their properties) programmatically

For example, you could **declare your application's default layouts in XML**, including the screen elements that will appear in them and their properties.

You could then **add code** in your application that would modify the state of the screen objects, including those declared in XML, at run time.

XML is best

- **Declaring your UI in XML enables**
 - to separate the presentation of your application from the code that controls its behaviour.
- **Your UI descriptions are external to your application code,**
 - which means that you can modify or adapt it without having to modify your source code and recompile
- **XML makes it easier**
 - to visualise the structure of your UI via Android Studio, so it's easier to debug problems

One XML Root

- Each layout file must contain exactly one root element,
 - which must be a **View** or **ViewGroup** object
- Additional layout objects or widgets are child elements
 - to gradually build a **View hierarchy** that defines your layout



look under res/layout/

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto"
4   android:layout_width="match_parent"
5   android:layout_height="match_parent"
6   app:layout_behavior="com.google.android.material.appbar.AppBarLayout$Scrolli...">
7
8   <fragment
9     android:id="@+id/nav_host_fragment_content_main"
10    android:name="androidx.navigation.fragment.NavHostFragment"
11    android:layout_width="0dp"
12    android:layout_height="0dp"
13    app:defaultNavHost="true"
14    app:layout_constraintBottom_toBottomOf="parent"
15    app:layout_constraintLeft_toLeftOf="parent"
16    app:layout_constraintRight_toRightOf="parent"
17    app:layout_constraintTop_toTopOf="parent"
18    app:navGraph="@navigation/nav_graph" />
19 </androidx.constraintlayout.widget.ConstraintLayout>
```


Layout for your Activities

When you compile your application, each XML layout file is compiled into a [View](#) resource.

You should load the layout resource from your application code, in

```
class MainActivity : AppCompatActivity() {

    private lateinit var appBarConfiguration: AppBarConfiguration
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setSupportActionBar(binding.toolbar)

        val navController = findNavController(R.id.nav_host_fragment_content_main)
        appBarConfiguration = AppBarConfiguration(navController.graph)
        setupActionBarWithNavController(navController, appBarConfiguration)
    }
}
```

_name.

ity

nder

the main directory and omits the .xml suffix

So **R.layout.activity_main** is equivalent to
/layout/main_layout.xml

Attributes

- **Views** and **ViewGroups** support a variety of XML attributes
- Some attributes are specific to a View object
 - for example, TextView supports the textSize attribute
- Attributes are inherited from parent elements
- Most of them define the layout parameters (e.g. size)

ID Attribute

ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the application is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the `View` class) and you will use it very often. The syntax for an ID, inside an XML tag is:

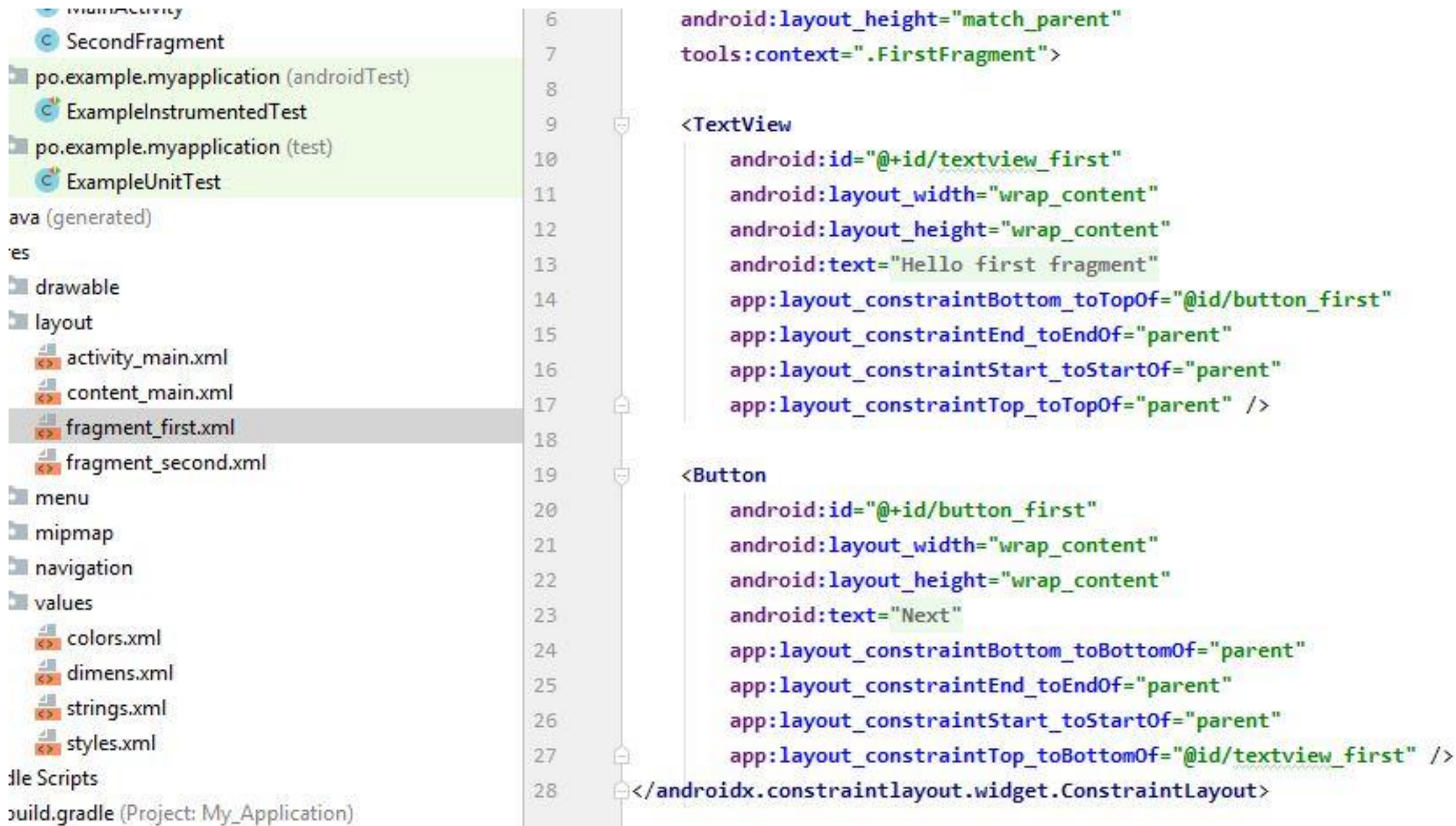
```
android:id="@+id/my_button"
```

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the `android` package namespace, like so:

```
android:id="@android:id/empty"
```

Why the ID attribute?

- To reference the element in the code



```
6 android:layout_height="match_parent"
7 tools:context=".FirstFragment">
8
9 <TextView
10     android:id="@+id/textview_first"
11     android:layout_width="wrap_content"
12     android:layout_height="wrap_content"
13     android:text="Hello first fragment"
14     app:layout_constraintBottom_toTopOf="@id/button_first"
15     app:layout_constraintEnd_toEndOf="parent"
16     app:layout_constraintStart_toStartOf="parent"
17     app:layout_constraintTop_toTopOf="parent" />
18
19 <Button
20     android:id="@+id/button_first"
21     android:layout_width="wrap_content"
22     android:layout_height="wrap_content"
23     android:text="Next"
24     app:layout_constraintBottom_toBottomOf="parent"
25     app:layout_constraintEnd_toEndOf="parent"
26     app:layout_constraintStart_toStartOf="parent"
27     app:layout_constraintTop_toBottomOf="@id/textview_first" />
28 </androidx.constraintlayout.widget.ConstraintLayout>
```

Why the ID attribute?

- So, e.g. we can capture a click event

```
class FirstFragment : Fragment() {  
  
    private var _binding: FragmentFirstBinding? = null  
  
    // This property is only valid between onCreateView and  
    // onDestroyView.  
    private val binding get() = _binding!!  
  
    override fun onCreateView(  
        inflater: LayoutInflater, container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
  
        _binding = FragmentFirstBinding.inflate(inflater, container, attachToParent: false)  
        return binding.root  
    }  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        binding.buttonFirst.setOnClickListener { it: View!  
            findNavController().navigate(R.id.action_FirstFragment_to_SecondFragment)  
        }  
    }  
  
    override fun onDestroyView() {  
        super.onDestroyView()  
        _binding = null  
    }  
}
```

Common Layouts

Linear Layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

Web View



Displays web pages.

Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses [AdapterView](#) to populate the layout with views at runtime. A subclass of the [AdapterView](#) class uses an [Adapter](#) to bind data to its layout. The [Adapter](#) behaves as a middleman between the data source and the [AdapterView](#) layout—the [Adapter](#) retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the [AdapterView](#) layout.

Common layouts backed by an adapter include:

List View



Displays a scrolling single column list.

Grid View



Displays a scrolling grid of columns and rows.

Linear Layout

[LinearLayout](#) is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the [android:orientation](#) attribute.

In this document

- > [Layout Weight](#)
- > [Example](#)

Key classes

- > [LinearLayout](#)
- > [LinearLayout.LayoutParams](#)



All children of a [LinearLayout](#) are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). A [LinearLayout](#) respects *margins* between children and the *gravity* (right, center, or left alignment) of each child.

Layout Weight

`LinearLayout` also supports assigning a *weight* to individual children with the `android:layout_weight` attribute. This attribute assigns an "importance" value to a view in terms of how much space it should occupy on the screen. A larger weight value allows it to expand to fill any remaining space in the parent view. Child views can specify a weight value, and then any remaining space in the view group is assigned to children in the proportion of their declared weight. Default weight is zero.

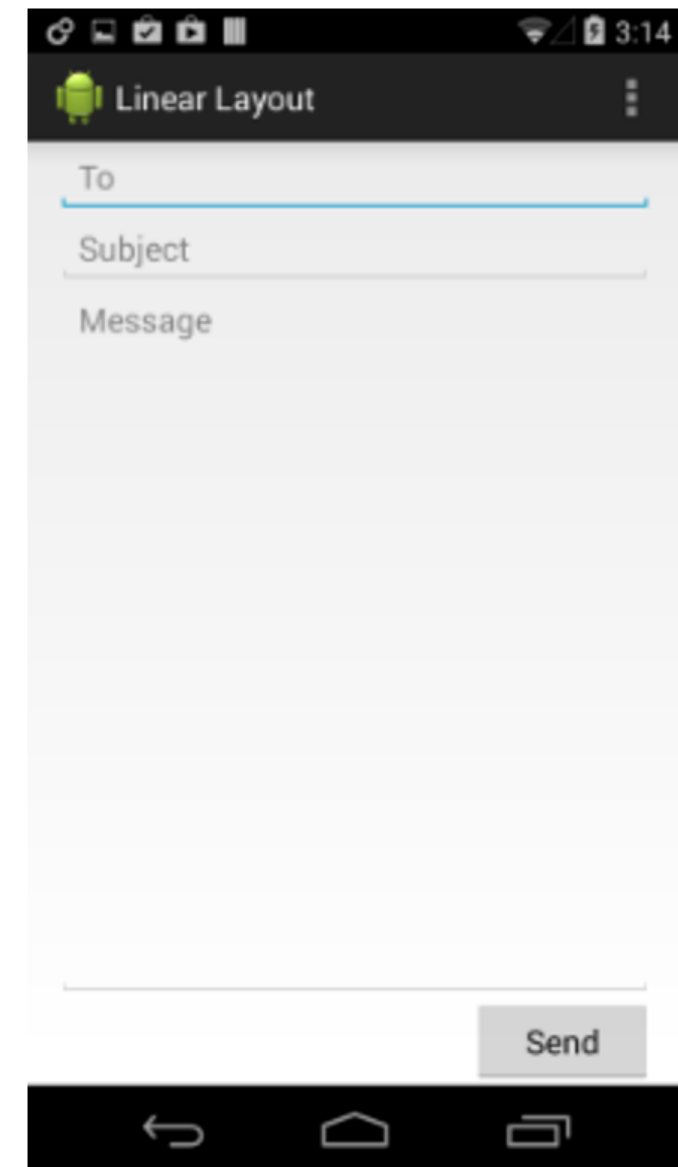
For example, if there are three text fields and two of them declare a weight of 1, while the other is given no weight, the third text field without weight will not grow and will only occupy the area required by its content. The other two will expand equally to fill the space remaining after all three fields are measured. If the third field is then given a weight of 2 (instead of 0), then it is now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

Equally weighted children

To create a linear layout in which each child uses the same amount of space on the screen, set the `android:layout_height` of each view to `"0dp"` (for a vertical layout) or the `android:layout_width` of each view to `"0dp"` (for a horizontal layout). Then set the `android:layout_weight` of each view to `"1"`.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```



Please look at the rest

Layouts

Linear Layout

Relative Layout

Recycler View

List View

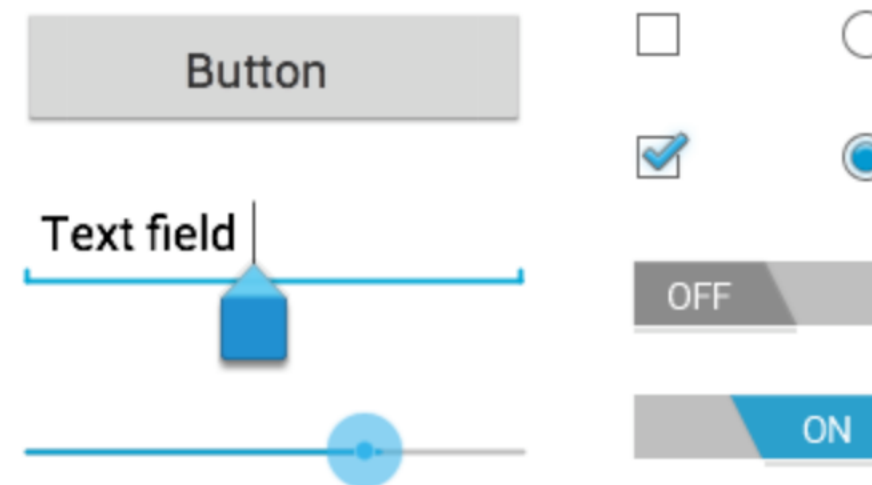
Grid View

<https://developer.android.com/guide/topics/ui/layout/linear.html>

Input Controls

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, checkboxes, zoom buttons, toggle buttons, and many more.

Adding an input control to your UI is as simple as adding an XML element to your [XML layout](#). For example, here's a layout with a text field and button:



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button android:id="@+id/button_send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send"
        android:onClick="sendMessage" />
</LinearLayout>
```

Each input control supports a specific set of input events so you can handle events such as when the user enters text or touches a button.

Controls

Common Controls <https://developer.android.com/guide/topics/ui/controls.html>

Here's a list of some common controls that you can use in your app. Follow the links to learn more about using each one.

Note: Android provides several more controls than are listed here. Browse the [android.widget](#) package to discover more. If your app requires a specific kind of input control, you can build your own [custom components](#).

Control Type	Description	Related Classes
Button	A push-button that can be pressed, or clicked, by the user to perform an action.	Button
Text field	An editable text field. You can use the AutoCompleteTextView widget to create a text entry widget that provides auto-complete suggestions	EditText , AutoCompleteTextView
Checkbox	An on/off switch that can be toggled by the user. You should use checkboxes when presenting users with a group of selectable options that are not mutually exclusive.	CheckBox
Radio button	Similar to checkboxes, except that only one option can be selected in the group.	RadioGroup RadioButton
Toggle button	An on/off button with a light indicator.	ToggleButton
Spinner	A drop-down list that allows users to select one value from a set.	Spinner
Pickers	A dialog for users to select a single value for a set by using up/down buttons or via a swipe gesture. Use a DatePicker widget to enter the values for the date (month, day, year) or a TimePicker widget to enter the values for a time (hour, minute, AM/PM), which will be formatted automatically for the user's locale.	DatePicker , TimePicker

Event Listeners

An event listener is an interface in the [View](#) 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).

Using an OnClickListener

You can also declare the click event handler programmatically rather than in an XML layout. This might be necessary if you instantiate the `Button` at runtime or you need to declare the click behavior in a `Fragment` subclass.

To declare the event handler programmatically, create an `View.OnClickListener` object and assign it to the button by calling `setOnClickListener(View.OnClickListener)`. For example:

```
Button button = (Button) findViewById(R.id.button_send);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Do something in response to button click
    }
});
```



The
University
Of
Sheffield.

Exercise



Create and run

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/to"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:hint="To" />

    <EditText
        android:id="@+id/subject"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:hint="Subject" />

    <EditText
        android:id="@+id/message"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:layout_gravity="top"
        android:hint="Message" />

    <Button
        android:id="@+id/send"
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:layout_margin="20dp"
        android:hint="Send" />

</LinearLayout>
```

9:03

lab1kt

To

Subject

Message

Send

in text

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/to"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:hint="To" />

    <EditText
        android:id="@+id/subject"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:hint="Subject" />

    <EditText
        android:id="@+id/message"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:layout_gravity="top"
        android:hint="Message" />

    <Button
        android:id="@+id/send"
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:layout_margin="20dp"
        android:hint="Send" />

</LinearLayout>
```

android:hint="To"
Should really be written as "@string/to"
where to is defined as

```
<string name="to">To</string>
```

in values.strings.xml

- Show the layout on the emulator
 - by running the app

2. When button is pressed

- Modify the Kotlin code so that:
 - When the button is pressed
 - The values of the fields are displayed in message
 - Steps:
 - assign ids to the elements in the XML file
 - set an `onClick` listener for the button in Kotlin
 - get the fields values Open **MainActivity.kt** file and
 - set *OnClickListener* for the button to get the user input from `EditText`
 - show the input as Toast message.
- See next slide for some hints
- `EditText` is used to get input from the user. `EditText` is commonly used in forms and login or registration screens.

Getting EditText Value

Here is a generic example on how to access the text in EditText at the press of a button to print the value

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        // Finding the edit text "To"  
        val m_To: EditText = findViewById(R.id.to)  
  
        // Finding the button "send"  
        val m_Button: Button = findViewById(R.id.send)  
  
        // Finding the edit text "subject"  
        val m_Subject: EditText = findViewById(R.id.subject)  
  
        // Finding the edit text "message"  
        val m_Message: EditText = findViewById(R.id.message)  
  
        // Setting On Click Listener  
        m_Button.setOnClickListener() { it: View!  
  
            //Getting the user input  
            val m_To_text = m_To.text  
            val m_Subject_text = m_Subject.text  
            val m_Message_text = m_Message.text  
  
            // Showing the user input  
            Toast.makeText(context: this, m_To_text, Toast.LENGTH_SHORT).show()  
        }  
    }  
}
```

Use the Debugger

- **You can also set a break point on something like**

```
Log.v("EditText", mEdit.getText().toString());
```

- and inspect the variables
 - you can hover over the code or look into the bottom window

Android EditText with Examples

Step 1: Open **activity_main.xml** file and create an EditText using id **to**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/to"
        android:inputType="text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:hint="To" />
```


Android EditText with Examples

Step 2: In `activity_main.xml` add code to show a button.

```
<Button
    android:id="@+id/send"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:layout_gravity="end"
    android:layout_margin="20dp"
    android:hint="Send" />
```

```
</LinearLayout>
```

Android EditText with Examples

Step 3: Open **MainActivity.kt** file and get the reference to **Button** and **EditText** defined in the Layout file.

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        // Finding the edit text "To"  
        val mTo: EditText = findViewById(R.id.to)  
  
        // Setting On Click Listener  
        m_Button.setOnClickListener() { it: View!  
  
            //Getting the user input  
            val m_To_text = m_To.text  
            val m_Subject_text = m_Subject.text  
            val m_Message_text = m_Message.text  
  
            // Showing the user input  
            Toast.makeText(context: this, m_To_text, Toast.LENGTH_SHORT).show()  
        }  
    }  
}
```

Step

Android EditText with Examples

Step 4: Setting the on click listener to the button.

```
// Setting On Click Listener
m_Button.setOnClickListener() { it: View!

    //Getting the user input
    val m_To_text = m_To.text
    val m_Subject_text = m_Subject.text
    val m_Message_text = m_Message.text

    // Showing the user input
    Toast.makeText(context: this, m_To_text, Toast.LENGTH_SHORT).show()
}
```



The
University
Of
Sheffield.

Android EditText with Examples

10:08

lab1kt

Po

COM4510

Lecture 1

Send