2.a. Views (widgets) and ViewGroups (layouts)

Mario Mata
mario.mata@gcu.ac.uk
M215a

GCU
Glasgow Caledonian University
University for the Common Good

`View` class

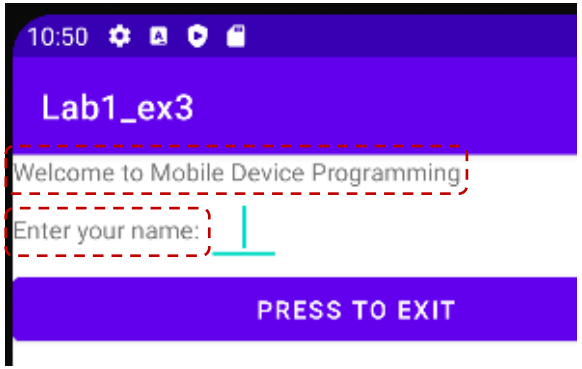
- Basic building block for GUIs. All components having a visual representation (**widgets**) inherit from `View`
  - We already know `TextView`, `EditText` and `Button`
  - We should do the same if we develop our own components

- A `View` occupies a rectangular area on the screen, and is responsible for **drawing** itself and producing **events**

- Each `View` has a number of **attributes** (aka *properties*) determining its appearance:
  - *Id, visibility, screen position, background colour, text colour, …*
  - Attributes can be set at design time (from the *`xml`* layout definition) or runtime (programmatically from Java code)

TextView class

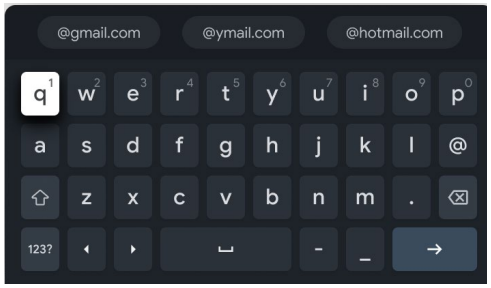
- The `TextView` widget is commonly used as a label, adding textual description next to other widgets
  - Can also be used to *output* short textual information



- Useful methods:
  - Setting the text content from a `String`:
    ```
    myTextView.setText("Length");
    ```
  - Getting the current text content:
    ```
    String a_string = myTextView.getText();
    ```
  - Setting the background colour:
    ```
    myTextView.setBackgroundColour(Color.rgb(255, 32, 0));
    ```
  - Setting foreground text colour:
    ```
    myTextView.setTextColour(Color.parseColor("#ff0200"));
    ```
  - Setting widget visibility:
    ```
    myTextView.setVisibility(INVISIBLE); //hide it
    ```

EditText class

- EditText is used to enter textual data for your app
  - Its default appearance includes a border (drawing a box)



- It opens an **on-screen keyboard**, allowing the user to enter text, numbers, email, passwords, etc.
- To retrieve the contents of a TextEdit: `getText()`
  - Returns an `Editable` object. It can be converted to `String`:

```
Editable e = myTextEdit.getText();
String a_string= e.toString();
```

  - Frequently shortened as:

```
String a_string= myTextEdit.getText().toString();
```

`EditText` class – on-screen keyboard

- The on-screen keyboard can be customized using property ***inputType***, i.e.: android:inputType = "textCapWords"

```
text
textCapCharacters
textCapWords
textCapSentences
textAutoCorrect
textAutoComplete
textMultiLine
textImeMultiLine
textNoSuggestions
textUri
textEmailAddress
textEmailSubject
textShortMessage
textLongMessage
textPersonName
textPostalAddress
textPassword
textVisiblePassword
textWebEditText
textFilter
textPhonetic
number
numberSigned
numberDecimal
phone
datetime
date
time
```

- This has a big impact on **usability**: a suitable keyboard (i.e. showing only numbers if we need a numerical value) **prevents input errors**, and makes typing easier
  - **NOTE**: the graphical layout editor IDE offers `EditText` views as *"Plain Text"*, *"Number"*, *"Number (Decimal)"*, *"E-mail"*, etc. depending on the *inputType* they use (it can still be changed)

Button class

- Button widgets are used to trigger an event on user interaction –usually `OnClick`
- Trapping this event allows to take action programmatically
  - Action is carried out in a "listener" method
  - Listener needs to be "attached" to object(s) of interest. This is implemented via an ***interface*** class in Java

- We'll discuss *events* in more detail shortly

- Default button appearance includes a border, and shows text inside it

Customizing widget borders

- In addition to widget colors, we can also set borders
- We can define the border in its own *xml* file (say *my_border.xml*), placed inside the ***res/drawable*** folder:

```
<shape
 xmlns:android="http://schemas.android.com/apk/res/android">
   <stroke  android:width="1dp"
            android:color="#55771B" />
   <padding android:left="17dp"
            android:top="13dp"
            android:right="17dp"
            android:bottom="17dp" />
   <corners android:radius="14dp" />
</shape>
```

my_border.xml

- Then we apply this border to the widget's *background*:

```
<TextView
    …
    android:background="@drawable/my_border"
 />
```

ViewGroup class - Layouts

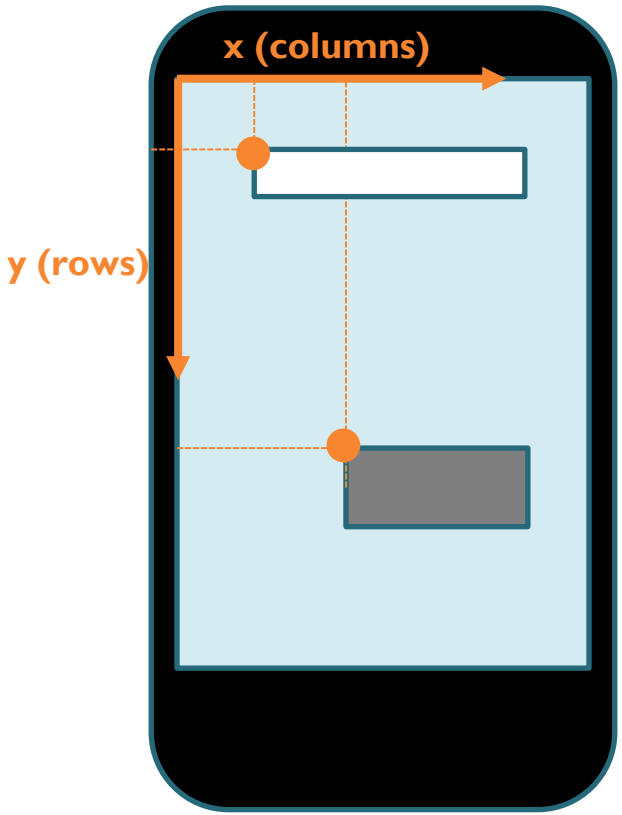
- `ViewGroup` inherits from `View`. It's parent class for **Layouts**

- Layouts are **containers** holding other *Views* (or other *ViewGroups*), and defining their placement in the screen
  - Layouts are **invisible** (unless we set their background colour)
  - A **parent-child** hierarchy is stablished between layout and contents
  - Any layout can have **another layout as a child**

- A range of layouts available to developers:
  - Absolute
  - Linear
  - Frame (and ScrollView)
  - Relative
  - Table / Grid
  - Constraint (default in Android Studio)

Layouts - Absolute

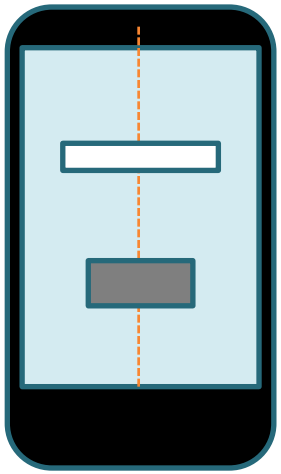
- Sets a **fixed** position for child views –not flexible!
  - In the screen / imaging world, the **top left-hand corner** of the screen is the origin of coordinates (0, 0)
    - The value for `layout_x` sets the widget column (left-right the screen)
    - The value for `layout_y` sets the rows (up-down the screen)



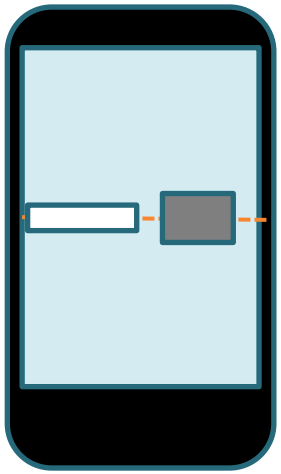

```
<AbsoluteLayout xmlns:android="http://..."
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<EditText
      android:id="@+id/enterInfo"
      android:layout_x="15px"
      android:layout_y="10px"
      … />
<Button
      android:id="@+id/navigateBack"
      android:layout_x="50px"
      android:layout_y="100px"
      … />
</Absolutelayout>
```

Layouts - Linear

- Linear Layout arranges the child items either *horizontally* (in a row) or *vertically* (in a column)
  - The `android:orientation` property controls this aspect



`android:orientation="vertical"`



`android:orientation="horizontal"`

```
<LinearLayout xmlns:android="http://..."
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
          android:text="Home Telephone"
          … />
    <Button
          android:id="@+id/navigateBack"
          android:text="Back"
          … />
</LinearLayout>
```

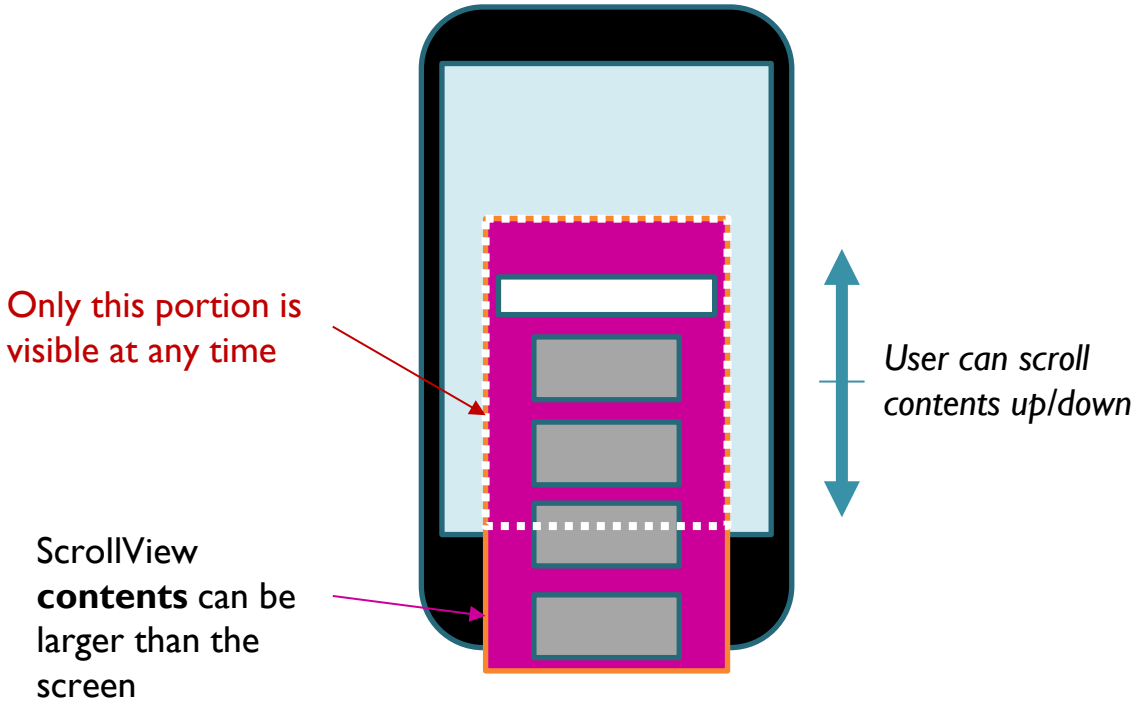
- As a layout can also contain other layouts, horizontal/vertical layouts are commonly combined

Layouts - Frame

- Frame Layout displays a **single** child View at a time
  - If you display multiple items they may overlap: the most recently added child will be shown on the top of others
  - You can use `android:layout_marginTop` and `android:layout_marginLeft` to separate the child items and avoid overlap
  - You can also use `setVisibility()` to programmatically show/hide widgets inside the Frame
    - I.e. a "splash screen" at the start of a game, showing on top of the main screen –its visibility can be changed to INVISIBLE after some time
- However, the Frame layout can contain another layout as its child -and this child layout can then contain several Views / further layouts
  - A Frame layout is often used to display a **Fragment** (more on this later on)

Layouts – ScrollView (I)

- ScrollView contains a view hierarchy that can be scrolled by the user
  - Its contents can be larger than the physical display. Only a portion is displayed, rest is hidden until scrolled into view
  - It extends FrameLayout, so only one child should be used (often a vertical LinearLayout, containing all other Views)



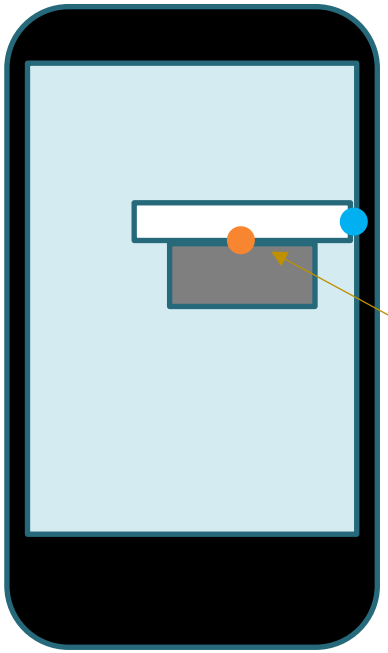
Layouts – ScrollView (II)

- Example xml definition for a ScrollView:

```
<ScrollView xmlns:android="http://..."
    xmlns:tools="http://schemas.android.com/tools"
    …
    tools:context=".ScrollTestActivity" >
    <LinearLayout //CHILD LAYOUT CONTAINING ALL ITEMS TO BE SCROLLED
        android:id="@+id/linearLayout2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >
        <TextView  //1st ITEM IN THE SCROLL
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/label1"  />
        <!– etc. -> //ALL OTHER ITEMS IN THE SCROLL
    </LinearLayout>
</ScrollView>
```

Layouts - Relative

- Relative layout sets the position of an item in relation to:
  - another **sibling** item (`above, below, toLeftOf, toRightOf`)
  - the **sides** of the parent (the relative layout)
  - Useful to place closely related components, i.e. a widget and its label, or a widget and a button doing something related



```
<RelativeLayout xmlns:android="http://..."
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/txtV1"
        android:text="Home Telephone“
        android:layout_alignParentRight="true"
        … />
    <Button
        android:id="@+id/navigateBack"
        android:layout_below="@id/txtV1"
        … />
</RelativeLayout>
```

Full list of properties:
http://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html

Layouts – Table / Grid

- Table layout arranges graphical components in a **grid** of rows and columns
  - The interface to some apps suits this i.e. a Dialer app

- Each `TableRow` entry is a child layout
  - Each item inside a row (**cell**) creates a column
  - Cells can contain any type of `View` (or Layout)
  - The table has as many columns as the row with the most cells

```
<TableLayout
 xmlns:android="http://...">
 <TableRow>
     <TextView
        …
       android:layout_column="1" />
     <EditText
        … />
 </TableRow>
 <TableRow>
      ……
 </TableRow>
</Tablelayout>
```

See the TableLayout reference documentation for more details

Layouts – Constraint (I)

- Constraint Layout arranges its child Views according to **relationships** between **sibling** Views and/or the **parent** layout (similar to relative layout)
  - This is the default layout that is used when you create a new project in Android Studio, but not the easier to start with!
- To define a View's position, you add **at least one** horizontal **and one** vertical constraint for the View
  - A View with no vertical constraints will appear on top
  - A View with no horizontal constraints will appear on the left
- You can use the graphical layout editor to create/edit constraints
- Or use the corresponding properties in the xml code

Layouts – Constraint (II)

- Constraints allow to specify:
  - **Position** within **parent** layout: constrain a side of the child view to an edge of the layout
  - **Position** to a **sibling** view: constrain a side of the child view to a side of another view
  - **Alignment** of a side of the view to the same side of another
- Related properties specify widget *sides*, with the form:

  `app:layout_constraintXXX_toYYYOf="id"`
  - **XXX** is a current view's *side* (`Start`, `End`, `Top` or `Bottom`)
  - **YYY** is a side of another View (or the parent layout)
  - **id** is either "parent" or the android id of another sibling view
- Margins are added to the constraint with:

  `android:layout_marginXXX="10dp"`

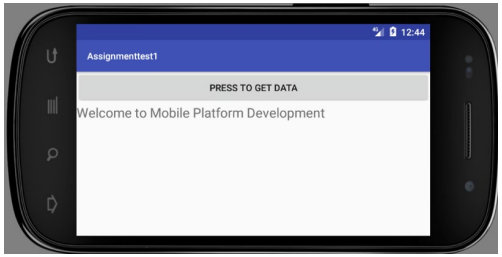
For details on all Constraint properties:
https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout

Device orientation: portrait/landscape

- Mobile devices can usually be rotated by the user into portrait (tall) or landscape (wide) orientations

Portrait:

Landscape:



- Portrait / landscape layouts are handled separately
  - Default orientation handling is usually not enough
  - We need **two** *activity_main.xml* files for the Portrait and the Landscape layouts. From the "Project File" view in AS:
    - **…/app/res/layout/activity_main.xml**
    - **…/app/res/layout-land/activity_main.xml**
  - **Check "CreatingPortraitAndLandscape.pdf" for details**