User Interface: Layout

Asst. Prof. Dr. Kanda Runapongsa Saikaew
Computer Engineering
Khon Kaen University
http://twitter.com/krunapon

Agenda

- User Interface
- Declaring Layout
- Common Layouts

User Interface

- View Hierarchy
- Layout
- Widgets
- UI Events
- Menus

Objects in User Interface

- In an Android application, the user interface is built using View and ViewGroup objects
- There are many types of views and view groups, each of which is a descendant of the View class
- View objects are the basic units of user interface expression on the Android platform
- The View class serves as the base for subclasses called "widgets," which offer fully implemented UI objects
- The ViewGroup class serves as the base for subclasses called "layouts"

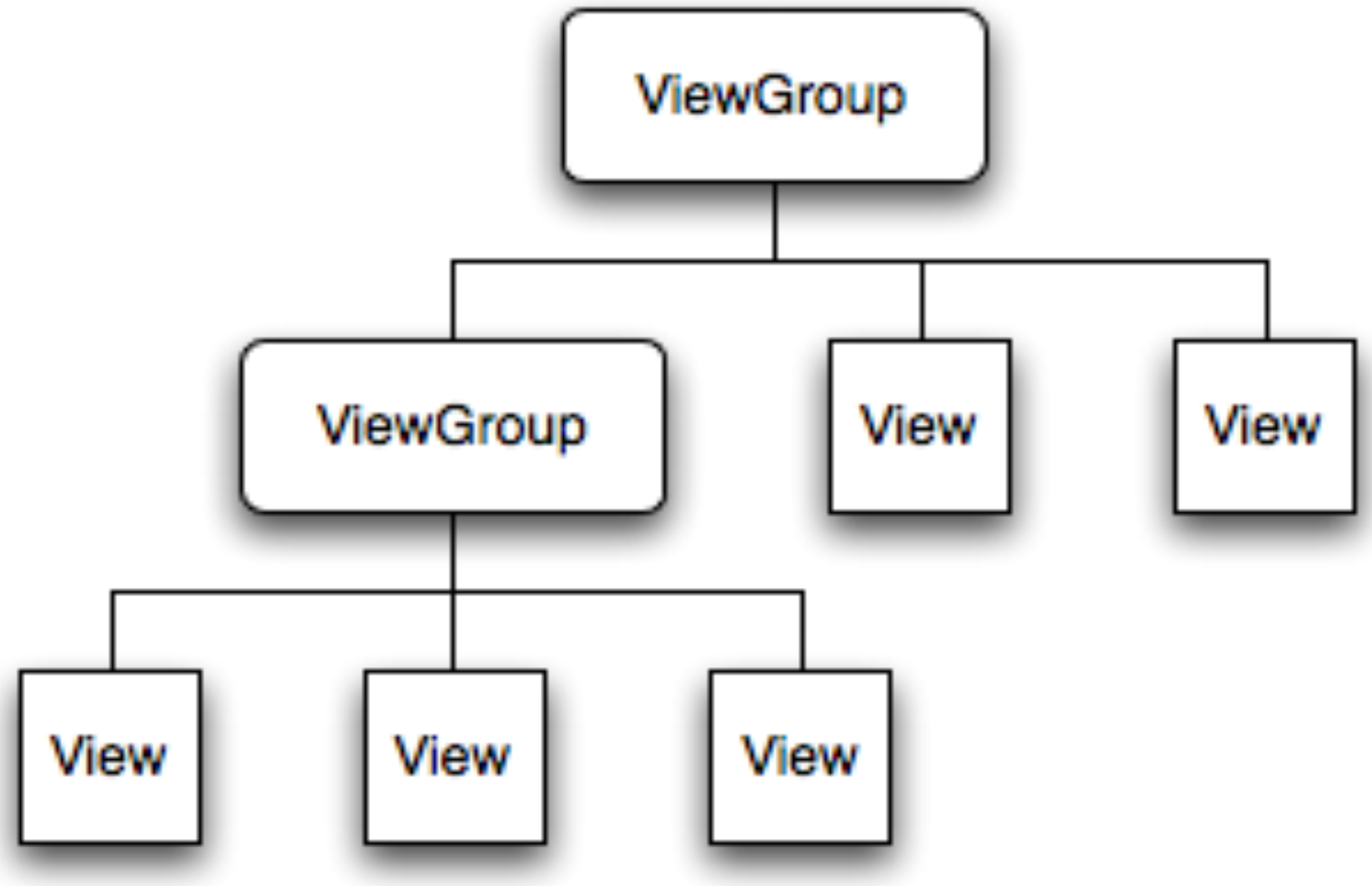
A View Object

- A View object is a data structure whose properties store the layout parameters and content for a specific rectangular area of the screen
- A View object handles its own measurement, layout, drawing, focus change, scrolling, and key/gesture interactions for the rectangular area of the screen in which it resides
- As an object in the user interface, a View is also a point of interaction for the user and the receiver of the interaction events

View Hierarchy

- On the Android platform, you define an Activity's UI using a hierarchy of View and ViewGroup nodes, as shown in the diagram below
- This hierarchy tree can be as simple or complex as you need it to be
  - You can build it up using Android's set of predefined widgets and layouts, or with custom Views that you create yourself
- In order to attach the view hierarchy tree to the screen for rendering, your Activity must call the setContentView()method and pass a reference to the root node object

View Hierarchy Sample



Drawing Process in View Hierarchy

- The Android system receives the reference to the root node object and uses it to invalidate, measure, and draw the tree
- The root node of the hierarchy requests that its child nodes draw themselves — in turn, each view group node is responsible for calling upon each of its own child views to draw themselves
- The children may request a size and location within the parent, but the parent object has the final decision on where how big each child can be

Elements Parsing Order

- Android parses the elements of your layout in-order (from the top of the hierarchy tree), instantiating the Views and adding them to their parent(s)
- The tree is traversed in-order, this means that parents will be drawn before (i.e., behind) their children, with siblings drawn in the order they appear in the tree
- Because these are drawn in-order, if there are elements that overlap positions, the last one to be drawn will lie on top of others previously drawn to that space

Drawing the Layout

- Drawing the layout is a two pass process: a measure pass and a layout pass
- The measuring pass is implemented in measure(int, int) and is a top-down traversal of the View tree
- Each View pushes dimension specifications down the tree during the recursion
- The second pass happens in layout(int, int, int, int) and is also top-down.
- During this pass each parent is responsible for positioning all of its children using the sizes computed in the measure pass.

The Measure Pass

- The measure pass uses two classes to communicate dimensions
    - The View.MeasureSpec class is used by Views to tell their parents how they want to be measured and positioned
    - The base LayoutParams class just describes how big the View wants to be for both width and height

View.MeasureSpec

MeasureSpecs are used to push requirements down the tree from parent to child. A MeasureSpec can be in one of three modes:

- UNSPECIFIED: This is used by a parent to determine the desired dimension of a child View
- EXACTLY: This is used by the parent to impose an exact size on the child. The child must use this size, and guarantee that all of its descendants will fit within this size
- AT_MOST: This is used by the parent to impose a maximum size on the child. The child must guarantee that it and all of its descendants will fit within this size

LayoutParams

For each dimension, LayoutParams can specify one of:

- An exact number
- FILL_PARENT, which means the View wants to be as big as its parent (minus padding)
- WRAP_CONTENT, which means that the View wants to be just big enough to enclose its content (plus padding)

Supporting Multiple Screens

- Android runs on a variety of devices that offer different screen sizes and densities
- For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed
- The quantity of pixels within a physical area of the screen; usually referred to as dots per inch (dpi)
- The density-independent pixel (dp) is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen.

The Relationship between dp, px, and dpi

- At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use
- The conversion of dp units to screen pixels is simple: px = dp * (dpi / 160)
- For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels
- You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

Screen Characteristic: Density

- ldpi Resources for low-density (ldpi) screens (~120dpi).
- mdpi Resources for medium-density (mdpi) screens (~160dpi). (This is the baseline density.)
- hdpi Resources for high-density (hdpi) screens (~240dpi).
- xhdpi Resources for extra high-density (xhdpi) screens (~320dpi).
- nodpi Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.

Layout

- The most common way to define your layout and express the view hierarchy is with an XML layout file
    - XML offers a human-readable structure for the layout, much like HTML
- Each element in XML is either a View or ViewGroup object (or descendant thereof)
- View objects are leaves in the tree
- ViewGroup objects are branches in the tree (see the View Hierarchy figure above)

XML Layout Elements

- The name of an XML element is respective to the Java class that it represents
- A <TextView> element creates a TextView in your UI
- A <LinearLayout> element creates a LinearLayout view group
- When you load a layout resource, the Android system initializes these run-time objects, corresponding to the elements in your layout
- Some pre-defined view groups offered by Android (called layouts) include LinearLayout, RelativeLayout, TableLayout, GridLayout and others

Widget

- A widget is a View object that serves as an interface for interaction with the user
- Android provides a set of fully implemented widgets, like buttons, checkboxes, and text-entry fields, so you can quickly build your UI
- Some widgets provided by Android are more complex, like a date picker, a clock, and zoom controls
- But you're not limited to the kinds of widgets provided by the Android platform
    - If you'd like to do something more customized and create your own actionable elements, you can, by defining your own View object or by extending and combining existing widgets.

UI Events

- Once you've added some Views/widgets to the UI, you probably want to know about the user's interaction with them, so you can perform actions
- To be informed of UI events, you need to do one of two things
  - **Define an event listener and register it with the View**
  - **Override an existing callback method for the View**

Defining an Event Listener

- More often than not, this is how you'll listen for events
- The View class contains a collection of nested interfaces named On<*something*>Listener, each with a callback method called On<*something*>()
- If you want your View to be notified when it is "clicked" (such as when a button is selected)
  - Implement OnClickListener and define its onClick() callback method (where you perform the action upon click)
  - Register it to the View with setOnClickListener()

Overriding an Existing Callback Method

- This is what you should do when you've implemented your own View class and want to listen for specific events that occur within it
- Example events you can handle include when the screen is touched (onTouchEvent()), when the trackball is moved (onTrackballEvent()), or when a key on the device is pressed (onKeyDown())
- This allows you to define the default behavior for each event inside your custom View and determine whether the event should be passed on to some other child View
- Again, these are callbacks to the View class, so your only chance to define them is when you build a custom component

Menus

- Application menus are another important part of an application's UI
- Menus offers a reliable interface that reveals application functions and settings
- The most common application menu is revealed by pressing the MENU key on the device
- However, you can also add Context Menus, which may be revealed when the user presses and holds down on an item

Implementing Menus

- Menus are also structured using a View hierarchy, but you don't define this structure yourself
- Instead, you define the onCreateOptionsMenu() or onCreateContextMenu() callback methods for your Activity and declare the items that you want to include in your menu
- Menus also handle their own events, so there's no need to register event listeners on the items in your menu
- When an item in your menu is selected, the onOptionsItemSelected() or onContextItemSelected() method will be called by the framework

Declaring Layout

- Write the XML
- Load the XML Resource
- Attributes
- Position
- Size, Padding, and Margins

Ways to Declare Layout

- Your layout is the architecture for the user interface in an Activity
- It defines the layout structure and holds all the elements that appear to the user
- You can declare your layout in two ways
  - Declare UI elements in XML
  - Instantiate layout elements at runtime
- The Android framework gives you the flexibility to use either or both of these methods for declaring and managing your application's UI

Declaring UI Elements in XML

- Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts
- Advantages
    - Enable you to better separate the presentation of application from the code that controls its behavior
    - Make it easier to visualize the structure of your UI, so it's easier to debug problems
    - You can create XML layouts for different screen orientations, different device screen sizes, and different languages
    - Be able to modify or adapt UI without having to modify your source code and recompile

Instantiating Layout Elements at Runtime

- Your application can create View and ViewGroup objects (and manipulate their properties) programmatically
- If you're interested in instantiating View objects at runtime, refer to the ViewGroup and View class references
- View
  - TextView
  - ViewGroup
    - LinearLayout
      - TableLayout
      - TabWidget
    - RelativeLayout
    - TabHost
    - GridView
    - ListView

Writing the XML

- Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements
- Each layout file must contain exactly one root element, which must be a View or ViewGroup object
- You can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout

Load the XML Resource

- 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 your Activity.onCreate() callback implementation
- Do so by calling setContentView(), passing it the reference to your layout resource in the form of:R.layout.*layout_file_name*
- For example, if your XML layout is saved as main_layout.xml, you would load it for your Activity like so:

-
  ```
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView.(R.layout.main_layout);
  }
  ```

Attributes

- Every View and ViewGroup object supports their own variety of XML attributes
- Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class
- Some are common to all View objects, because they are inherited from the root View class (like the id attribute).

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.

New Attribute ID

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)

IDs of Existing Resources

- 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"
```

- With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.

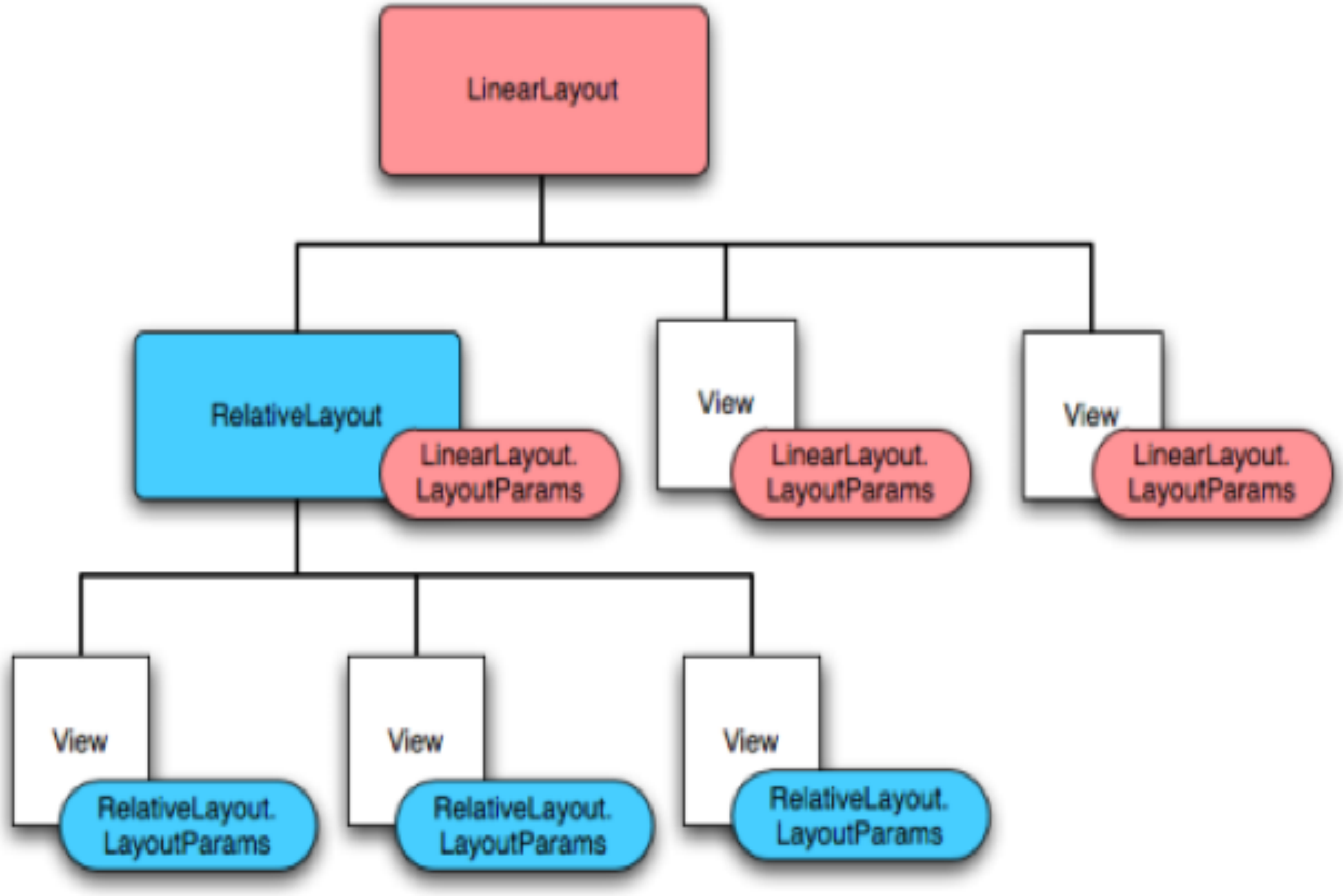
Creating and Referencing Views

1. Define a view/widget in the layout file and assign it a unique ID:`<Button android:id="@+id/my_button" ...>`
2. Then create an instance of the view object and capture it from the layout (typically in the onCreate() method):`Button myButton = (Button) findViewById(R.id.my_button);`

Layout Parameters

- XML layout attributes named layout_*something* define layout parameters for the View that are appropriate for the ViewGroup in which it resides
- Every ViewGroup class implements a nested class that extends [ViewGroup.LayoutParams]()
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group
- The parent view group defines layout parameters for each child view (including the child view group)

Layout Parameters



Setting Width and Height

- In general, specifying a layout width and height using absolute units such as pixels is not recommended
- Instead, using relative measurements such as density-independent pixel units (dp), wrap_content, or fill_parent, is a better approach
  - Because it helps ensure that your application will display properly across a variety of device screen sizes

Layout Position

- The geometry of a view is that of a rectangle
- A view has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and a height
- The unit for location and dimensions is the pixel
- It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop()
- These methods both return the location of the view relative to its parent
- In addition, several convenience methods are offered to avoid unnecessary computations, namely getRight() andgetBottom().

View Size

- The size of a view is expressed with a width and a height. A view actually possess two pairs of width and height values
- The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() andgetMeasuredHeight()
- The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout

View Padding

- To measure its dimensions, a view takes into account its padding
- The padding is expressed in pixels for the left, top, right and bottom parts of the view
- Padding can be used to offset the content of the view by a specific amount of pixels
- For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge
- Padding can be set using the setPadding(int, int, int, int) method and queried by callinggetPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

Constants for Setting Width and Height

- Often you will use one of these constants to set the width and height
  - wrap_content tells your view to size itself to the dimensions required by its content
  - fill_parent (renamed match_parent in API Level 8) tells your view to become as big as its parent view group will allow

Common Layouts

- LinearLayout
- TableLayout
- RelativeLayout

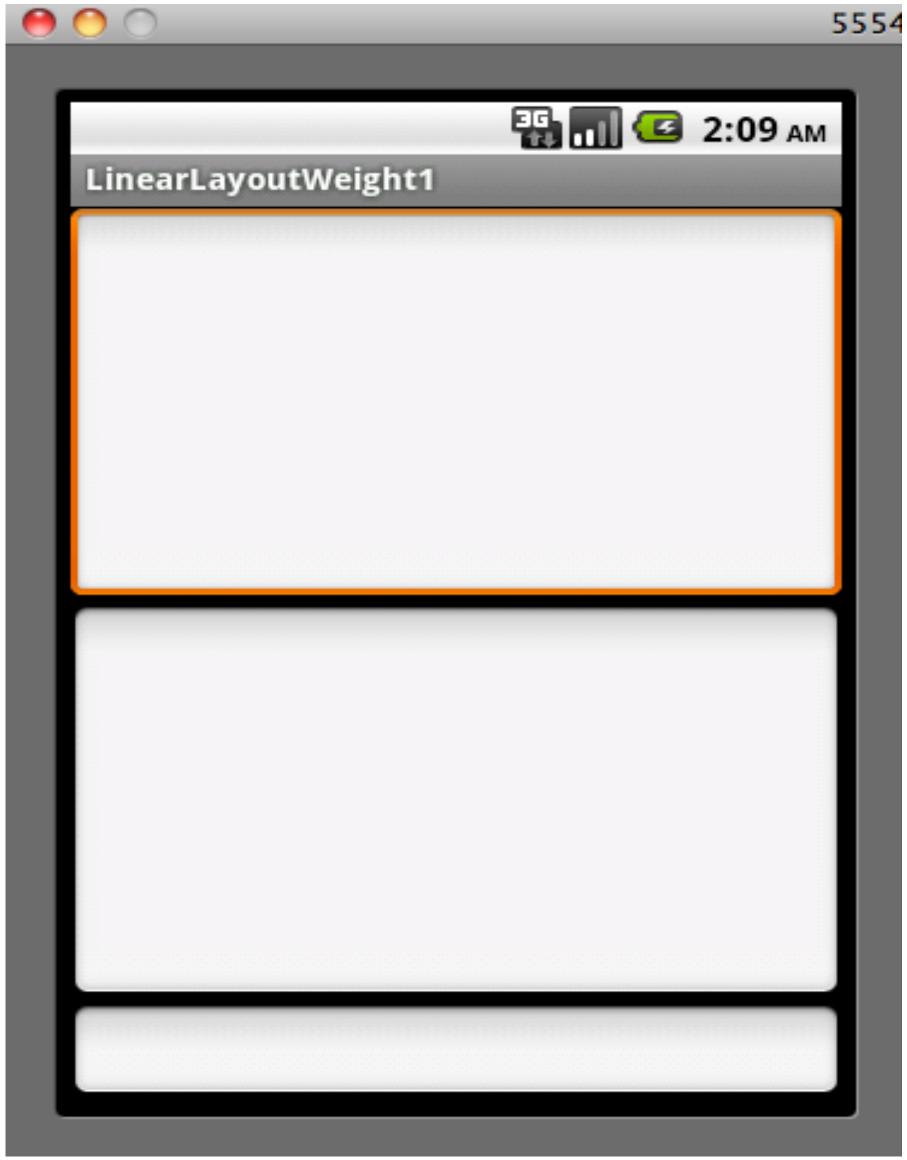
Linear Layout

- LinearLayout aligns all children in a single direction — vertically or horizontally, depending on how you define the orientation attribute
- All children 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)
- LinearLayout also supports assigning a *weight* to individual children
  - This attribute assigns an "importance" value to a view, and allows it to expand to fill any remaining space in the parent view
  - Default weight is zero

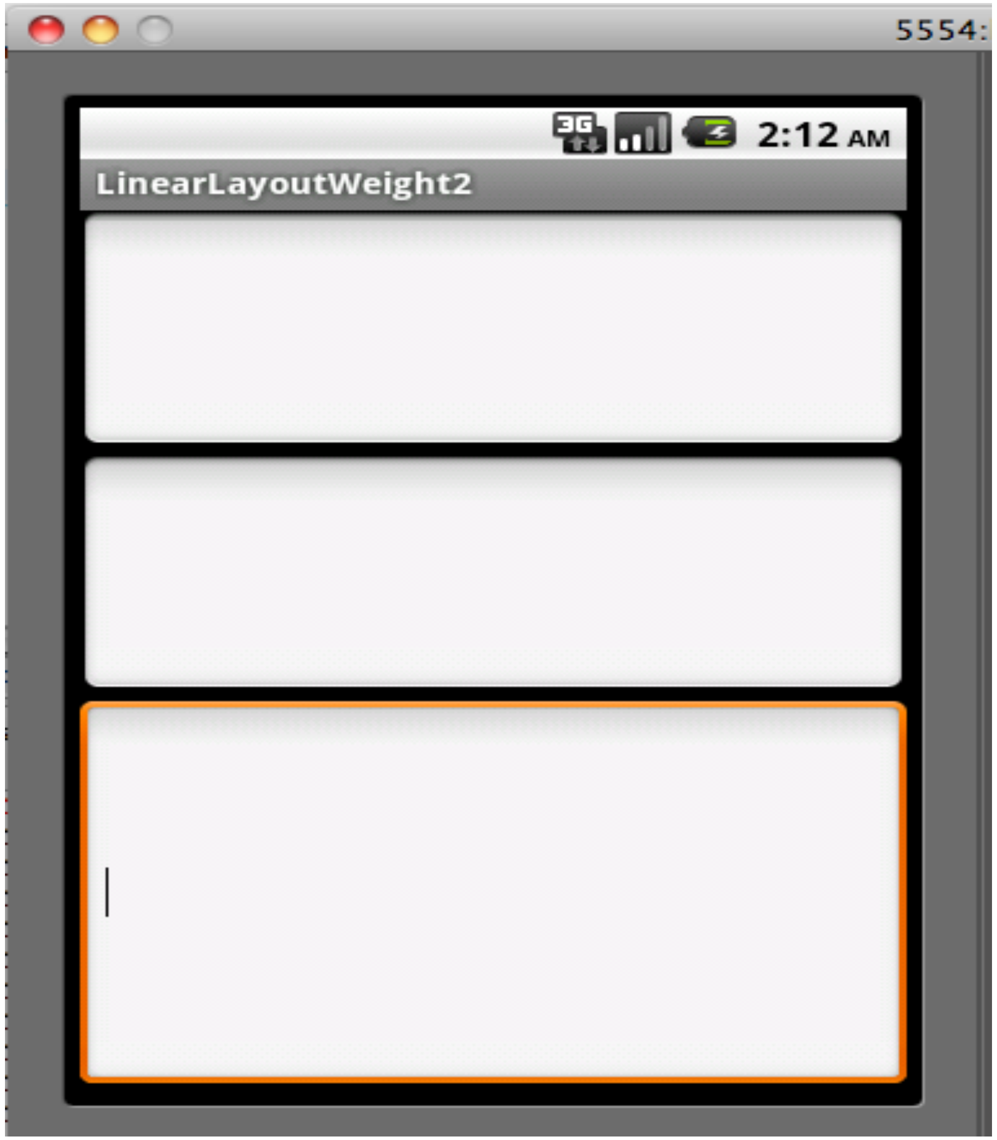
Weight Examples

- If there are three text boxes and two of them declare a weight of 1, while the other is given no weight (0)
  - The third text box 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 boxes are measured.
- If there are three text boxes and two of them declared a weight of 1 and the third box is given a weight of 2
  - The third text box 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.

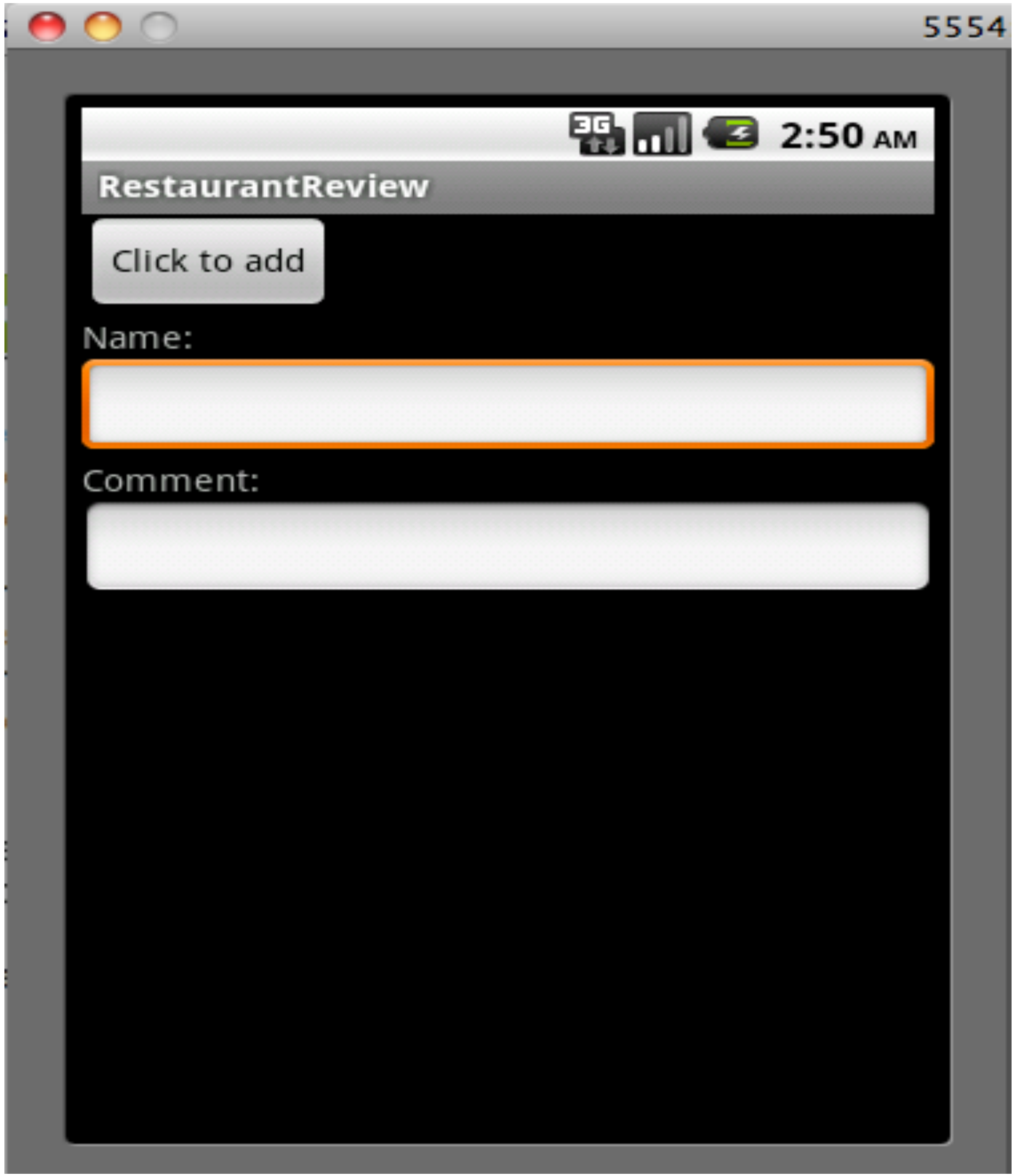
LinearLayout Weight Example1



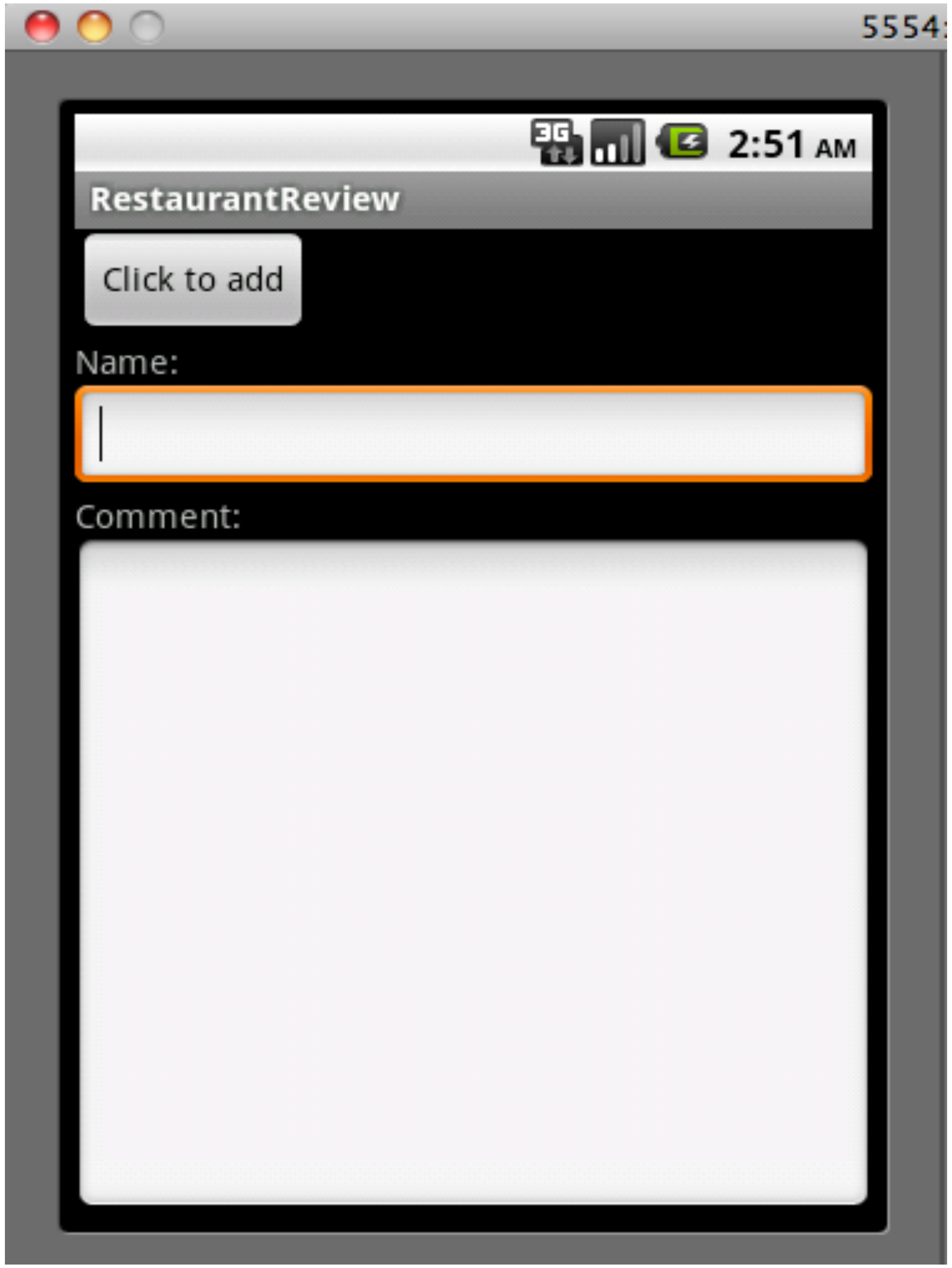
LinearLayout Weight Example2



Equal Weight Text Boxes



Unequal Weight Text Boxes



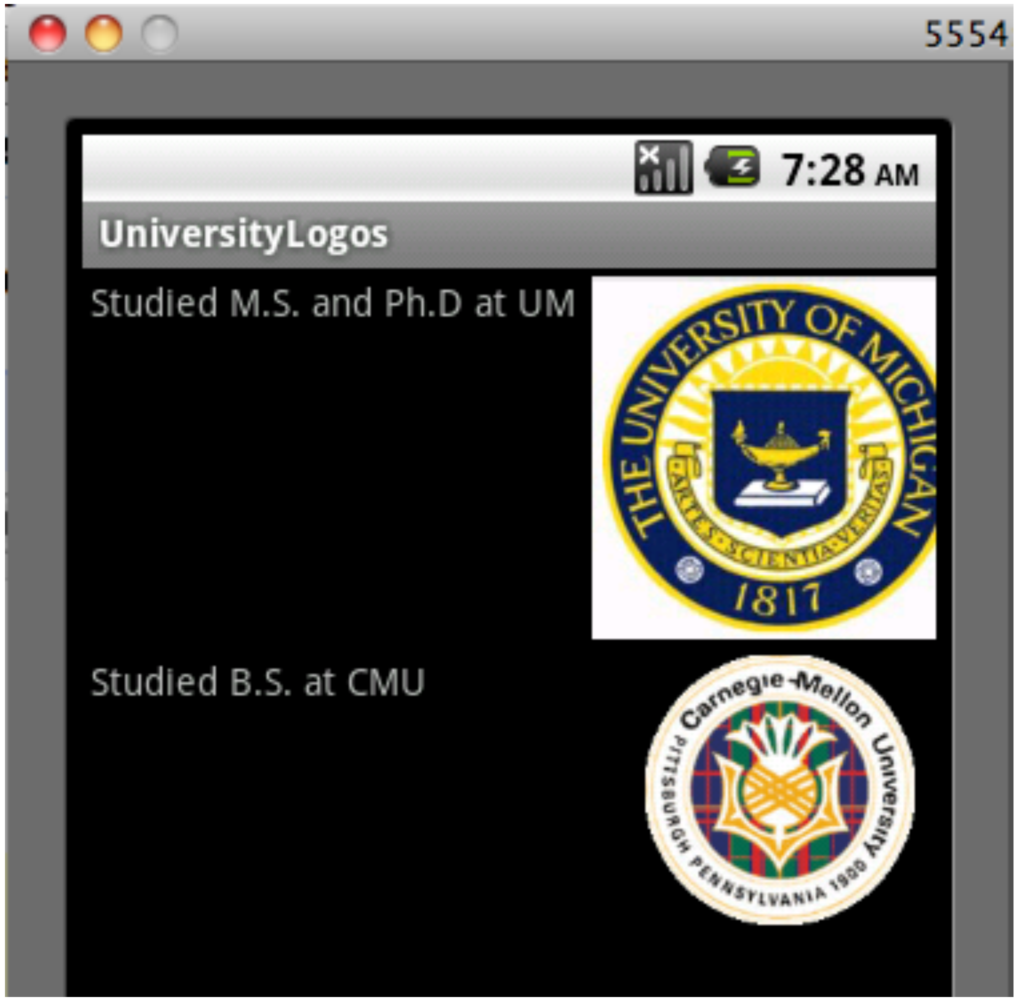
TableLayout

- TableLayout positions its children into rows and columns
- TableLayout containers do not display border lines for their rows, columns, or cells
- The table will have as many columns as the row with the most cells
- A table can leave cells empty, but cells cannot span columns, as they can in HTML

TableRow

- TableRow objects are the child views of a TableLayout
- Each TableRow defines a single row in the table
- Each row has zero or more cells, each of which is defined by any kind of other View
- The cells of a row may be composed of a variety of View objects, like ImageView or TextView objects

TableRow with ImageView & TextView



UniversityLogos

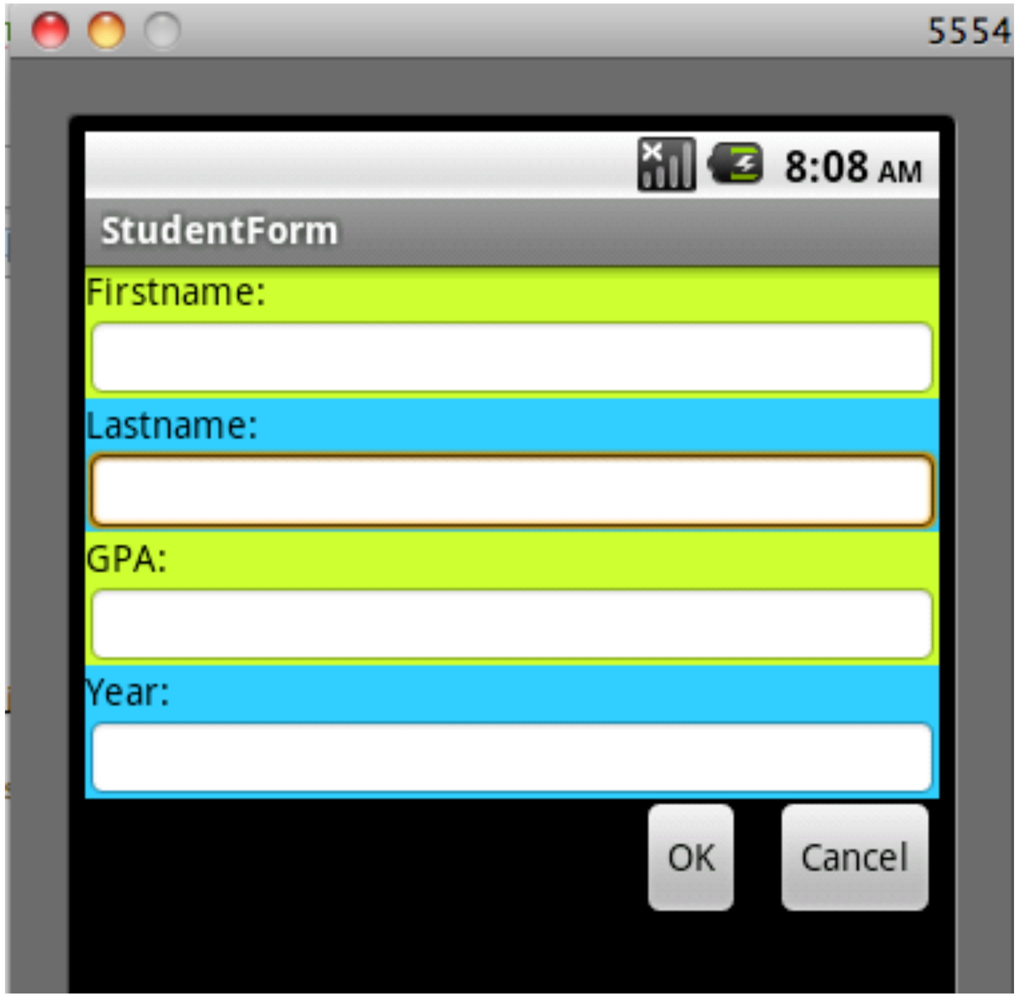
Studied M.S. and Ph.D at UM

Studied B.S. at CMU

RelativeLayout

- RelativeLayout is a ViewGroup that displays child View elements in relative positions
- The position of a View can be specified as
  - Relative to sibling elements (such as to the left-of or below a given element) or
  - In positions relative to the RelativeLayout area (such as aligned to the bottom, left of center)
- Elements are rendered in the order given
  - The element that you will reference (in order to position other view objects) must be listed in the XML file before you refer to it from the other views via its reference ID

StudentForm



References

- http://developer.android.com/guide/topics/ui/index.html
- http://developer.android.com/guide/topics/ui/declaring-layout.html
- http://developer.android.com/guide/topics/ui/layout-objects.html