# Mobile UI Development in Android

Lecture 7

# Goal for today

- Understand how to manage views with layout managers
- How to inflate custom UI
- Custom adapters for list views
- Declarative versus programmatic UI

# Content

- Alternatives of UI definition
- Views and view groups
- Layouts and inflaters
- ViewHolder pattern

# Introduction

Any app consists of windows:

- Which contain several visual components called controls, views, widgets or viewing elements.
- GUI is based on single-threaded implementations

  Advantages/ disadvantages?
- The user's actions are transformed by the OS in events which are transmitted to the application.
- The functions dealing with these events are called handler functions/methods.

# Android UI architecture

→Single threaded UI – main app thread.

→Handles user input, events, drawing, parsing and creation of layout controls.

→Event-driven

→Library of nested components

Avoid blocking of UI thread

→Background threads, asynchronous tasks

UI design patterns

- Separation of the visualizing components and the data structures (view and model)

# Android MVC UI

→Model-View-Controller

Offers a clear separation between model (data), view (display) and controller (treats events which affect the model or view)

# Android MVC UI

→Model

- Contains the data structures and objects that keep the application specific state and data.
- Serves as the source for data for the display windows.
- Can be modified through the controller following a user action or an internal operation.
- Notifies display windows to redraw themselves in case its state has changed.

# Android MVC UI

→View

- Implements the component for viewing the model.
- The application component which draws the user interface for the model.
- Implemented as a tree made from objects derived from the Android View class.
- From a graphical point of view, each object represents a rectangular area on the screen, included in the parent area of the tree.
- Displaying on the screen is done by traversing the tree in preorder and drawing each object

# Android MVC UI

→Controller

- The application component which handles external events: tap, keystroke, phone call, etc.
- Implemented as an event queue.
- Each external action is represented as a unique event.
- Events are taken from the queue and distributed towards execution to the corresponding handler methods.
- Single-threaded:

  Each event is processed completely before initiating the next one.

# Android MVC UI

→Controller

- Asynchronous Callback
- The operations that last longer shouldn't block the UI
- How can you avoid blocking the UI at the execution of long lasting operations?

  It is recommended to implement long-lasting operations in different threads.

  The end of operations can be communicated to the UI through callback functions or messages.

# Android GUI basics

## View

→Represents the base block for UI element construction

→Occupies a rectangular area on the screen and is responsible with drawing content and handling events

## ViewGroup

→Subclass of View

→Invisible container which contains other Views or ViewGroups

→Base class for layout

# Android GUI basics

## Layout

→Represents the UI design for an activity

→Establishes the UI elements placement structure in the window and keeps those elements

→There are two ways to create it:

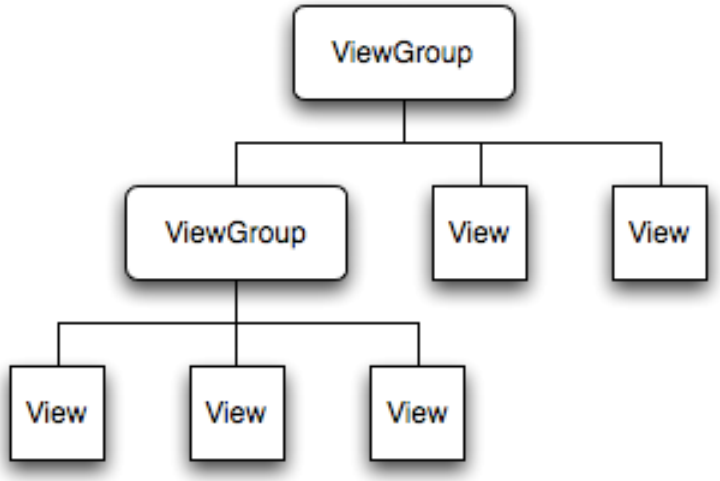
Declaring UI elements in an XML resource file

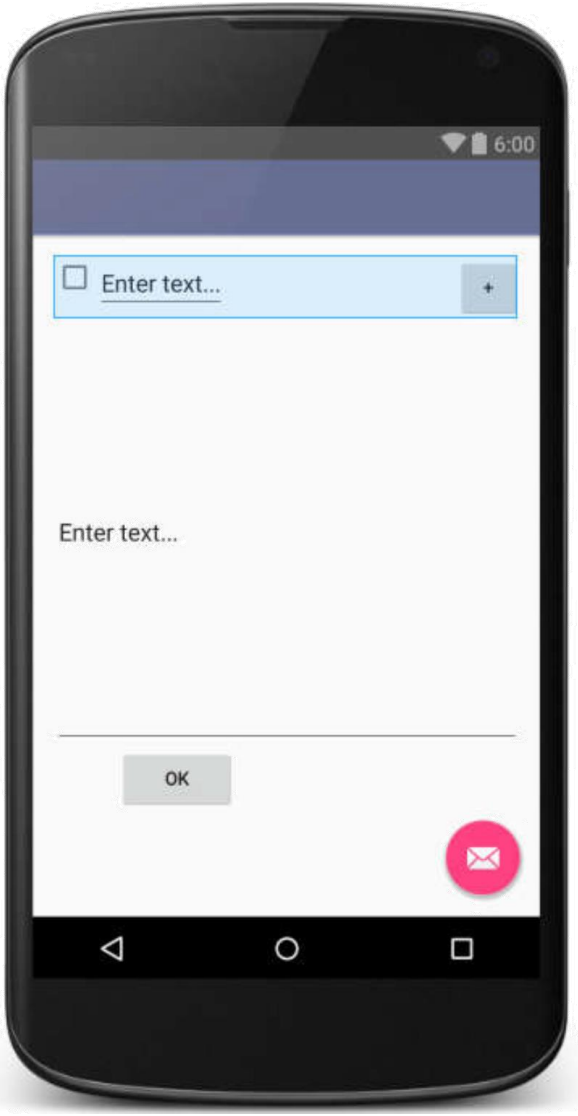
Programmatic creation of UI elements at runtime

# Android GUI hierarchy

UI element hierarchy in an activity:

- ViewGroup – layouts
- View: button, text, edit, check box



# Android GUI elements

Controls (View / Widget)

- Leaf nodes of the UI tree
- Render a specific control on the screen

Containers (ViewGroup, Layout)

- Internal nodes of the UI tree
- Manage child objects on the screen
- Manage changes of UI according with device configuration changes
- Nested containers decrease UI performance

# Attributes - id

- **id** – unique in its view tree (subtree where it is searched)

```
android:id="@+id/button1"
```

**@** means the XML parser has to extend the *id* (fully qualified name)

**+** means a new view has to be created and added to the R.java resources file

You may also reference a view from the *android.R* resources file:

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

# Attributes - id

- Define a new view in the layout xml

```
<Button android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/someText"/>
```

Referring to the view in *onCreate*:

```
Button button1 = (Button) findViewById(R.id.button1);
```

```
Button button1 = (Button)someLayout.
findViewById(R.id.button1);
```

# Attributes – layout params

Specifying width and height is mandatory for all views

- wrap_content – resize to smallest possible dimensions to fit content
- match_parent – resize to fit parent viewgroup's dimension (on one axis only)

Do not use pixels (px) to specify size

Use density independent pixels (dp) instead

```
android:layout_width="150dp"
```

# Linking Activity with XML

**setContentView**(parentContainer)

- The root view is called to render itself.
- The root view then calls its children to render themselves.
- The child controls call their children recursively until the entire UI is rendered.

# Retrieve views from an XML

**findViewById**(R.id.view)

- Finds a view that was identified by the android:id XML attribute that was processed in *onCreate*.
- The resulting view should be cast to the appropriate type.

```
ImageView imageView=(ImageView)findViewById(R.id.imageView);
```

Will only work after setContentView (in onCreate)

Cannot be called to initialize a class member, e.g.:

```
private Button btn1 = (Button)findViewById(R.id.btn1);
```

btn1 will be null!

# Android layouts

- LinearLayout – a container which arranges the internal UI elements in a linear arrangement: horizontal or vertical.
- RelativeLayout – a container which arranges the UI elements relative to the parent and each other.
- TableLayout – a container which arranges the UI elements as a table, with rows and columns.
- FrameLayout – a container in which UI elements are placed overlapping in the top left corner.
- AbsoluteLayout – a container in which the absolute placement (on screen) of the UI elements can be specified.

# Layouts

- Web View – a view that displays web pages.
- List View – a view group which displays a list of scrollable items.
- Grid View – a view group which displays items in a two dimensional scrollable grid.

# LinearLayout

Characterized by:

→Orientation (vertical or horizontal)

→Fill model (sizing mode – match, wrap, fixed size)

→Weight (relative sizing of children)

→Gravity (alignment mode)

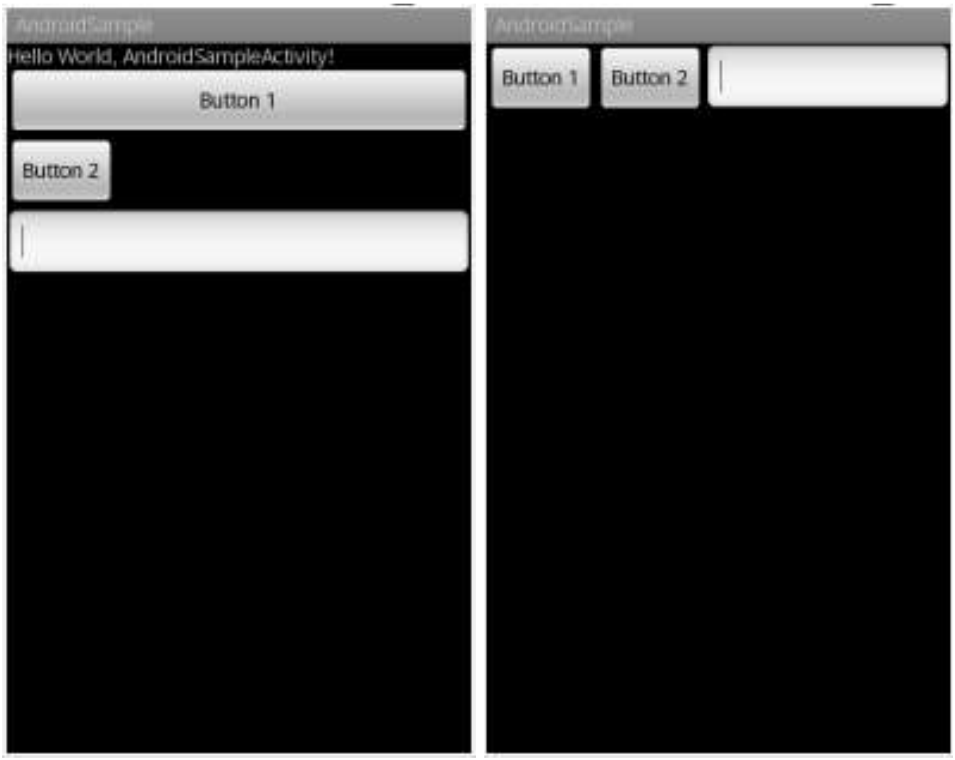
→Padding (relative content spacing from the edges inside the view)

→Margin (relative spacing around the edges of the parent, outside the view)

# LinearLayout

Layout orientation →

- horizontal vs vertical



# LinearLayout

Weight →two variants of specifying it:

- sum_weight = X in parent (=100%) and layout_weight of all children should sum up X.
- Define child's layout_weight **(<1)** and sum up to **1.0**. E.g., 0.3 → 30% of parent size

Example: weights 1,2,1 →

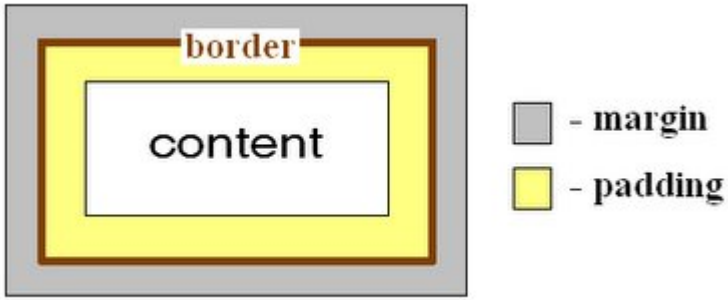
# LinearLayout

Gravity → specifies the alignment mode for the contained elements.

# LinearLayout

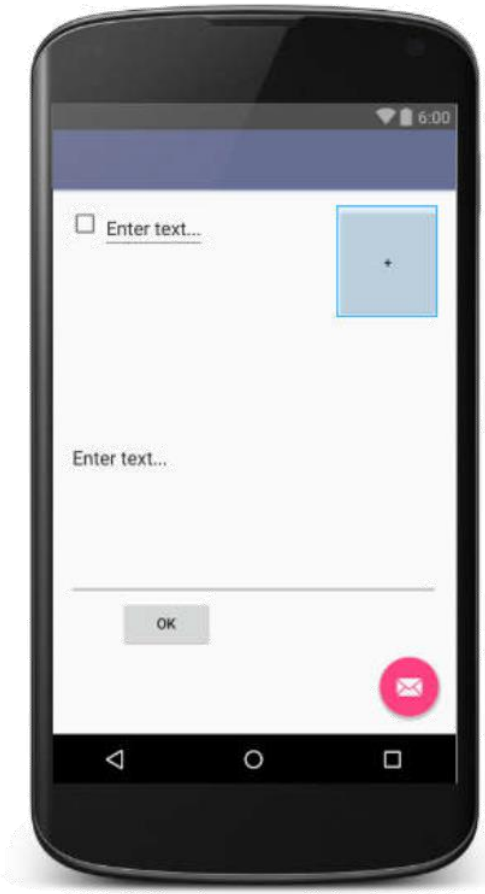
Margin versus padding

(Outside versus inside)



# LinearLayout – Padding example

```
android:text="@string/default_text"
android:focusable="false"
android:layout_toRightOf="@id/checkBox"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_weight="1"/>

<Button
    style="?android:attr/buttonStyleSmall"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="+"
    android:layout_alignParentRight ="true"
    android:padding="20pt"
    android:id="@+id/button2"
    android:layout_gravity="right" />

RelativeLayout>


ditText
  android:id="@+id/text2"
  android:text="@string/default_text"
  android:focusable="false"
  android:layout_width="fill_parent"
```



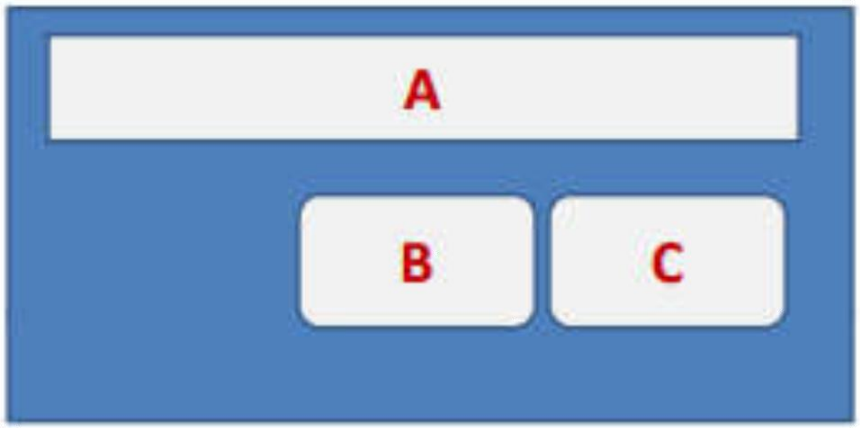
# LinearLayout – Margin example

```
<EditText
    android:id="@+id/text1"
    android:text="@string/default_text"
    android:focusable="false"
    android:layout_toRightOf="@id/checkBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"/>

<Button
    style="?android:attr/buttonStyleSmall"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="+"
    android:layout_alignParentRight ="true"
    android:layout_margin="20pt"
    android:id="@+id/button2"
    android:layout_gravity="right" />

RelativeLayout>


ditText
  android:id="@+id/text2"
  android:text="@string/default_text"
```

# RelativeLayout

Places the contained elements relative to the container or one another.

- Each contained element must be identified by a unique name using an ID (e.g. @+id/viewName)
- Referring other elements is done using the ID.



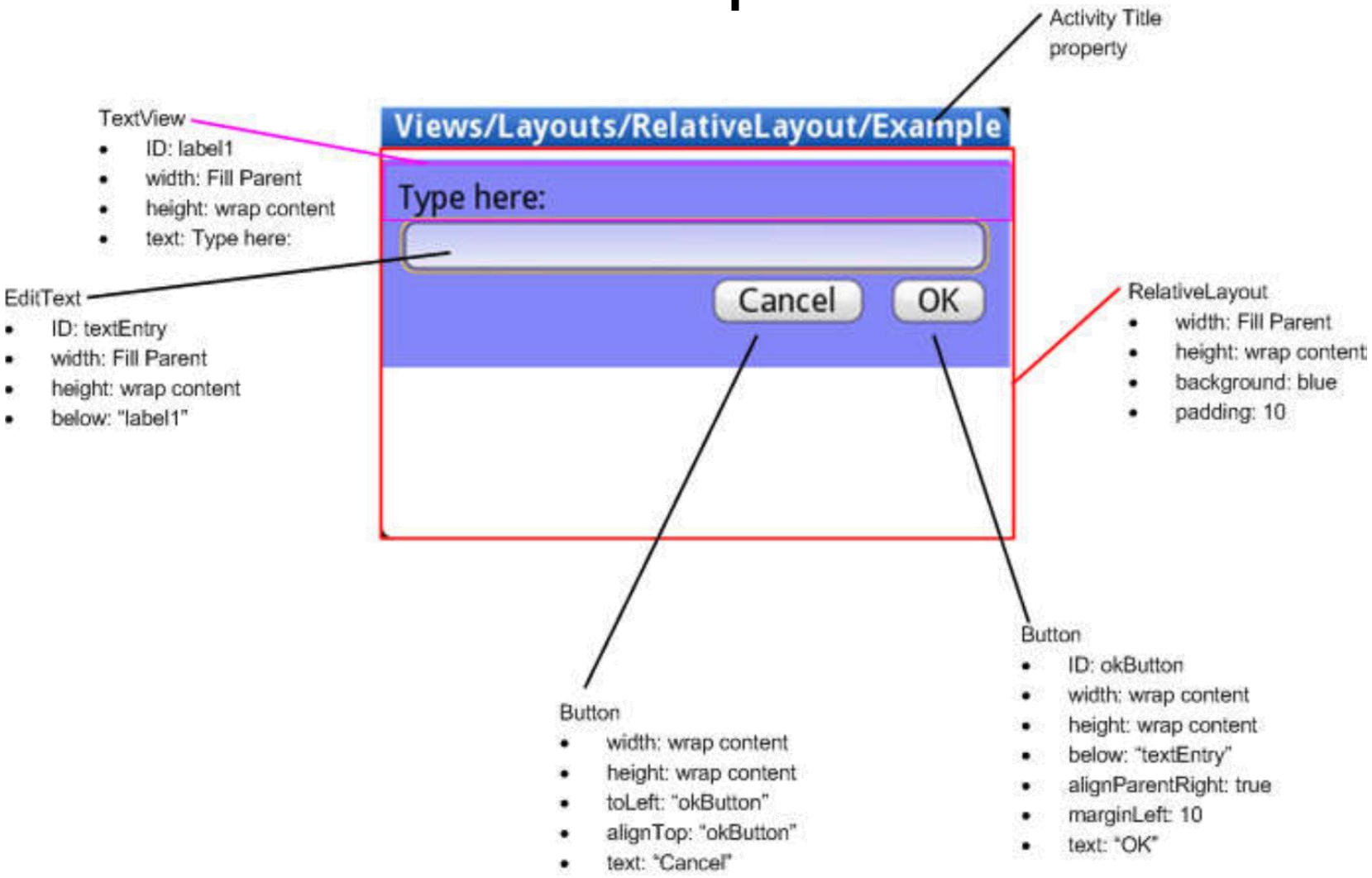

**Example:**
A is by the parent's top
C is below A, to its right
B is below A, to the left of C

# RelativeLayout

Layout rules:

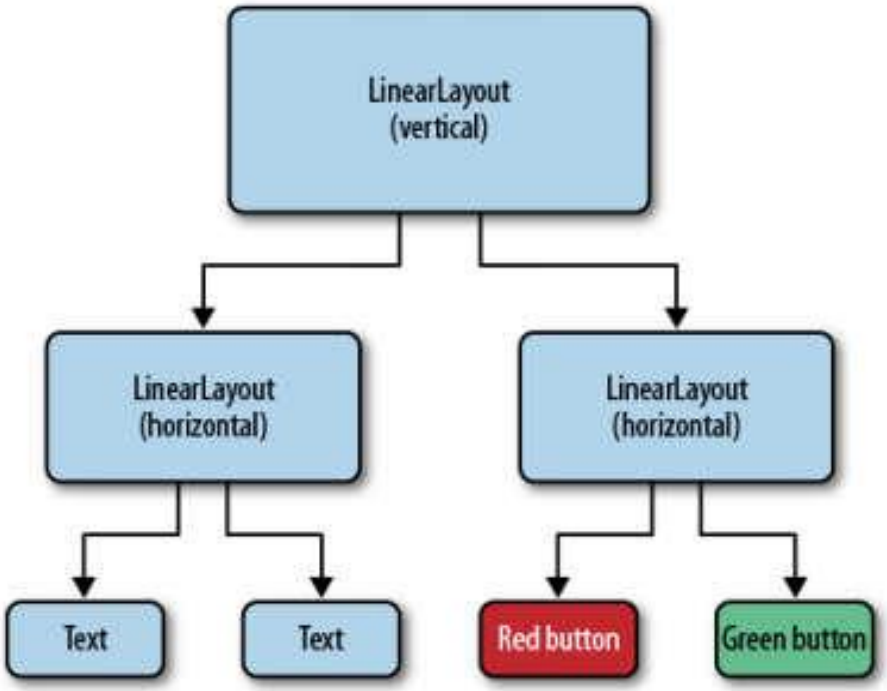
- layout_alignParentTop, Bottom, Left, Right
- layout_centerHorizontal, Vertical, InParent
- layout_above, below
- layout_toLeftOf, toRightOf
- layout_alignTop, bottom, left, right, baseline

# Android GUI exam example



# Android GUI hierarchy

Another example.


This is not the only solution.

E.g. vertical in horizontal linear layouts; relative in relative; linear in relative; just one relative layout etc.

# TableLayout

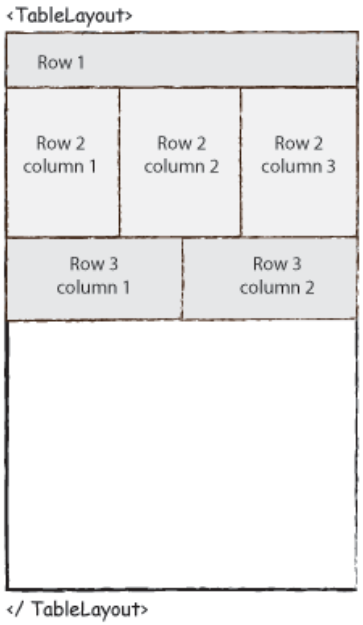
Allows to position the controls in a table of identifiable rows and columns.

- Columns might shrink or stretch according with their content.
- TableRow →

<TableRow> is used to build a row in the table.

Each row has zero or more cells;

Each cell can hold one View object.



# Web View

A View that displays web pages.

Basis for own web browser or displaying online content.

→Uses the WebKit rendering engine, and includes methods to navigate forward and backward through a history, zoom in and out, perform text searches and more.

→Needs `<uses-permission android:name="android.permission.INTERNET" />`

```
WebView webview = new WebView(this);
setContentView(webview);
```

# Web View

Does not enable JavaScript and web page errors are ignored by default.

```
// Simplest usage: note that an exception will NOT be thrown if there is an error loading this page (see below).
 webview.loadUrl("https://example.com/");


 // OR, you can also load from an HTML string:
 String summary = "<html><body>You scored <b>192</b> points.</body></html>";
 webview.loadData(summary, "text/html", null);
```

# Web View

May handle only specific URLs by intercepting them via *shouldOverrideUrlLoading*() – generic hybrid app.

Otherwise, pass content to a browser application:

```
Uri uri = Uri.parse("https://www.example.com");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);;
```

# List View

List items (objects) are inserted to the list using an *Adapter* that pulls content from a source such as an array (or database query).

Adapters convert each item into a view that is placed in the list.

→Use ArrayAdapter or a custom adapter to populate list.

→ArrayAdapter handles strings by invoking toString()

# Example of populating a list view

## Model class:

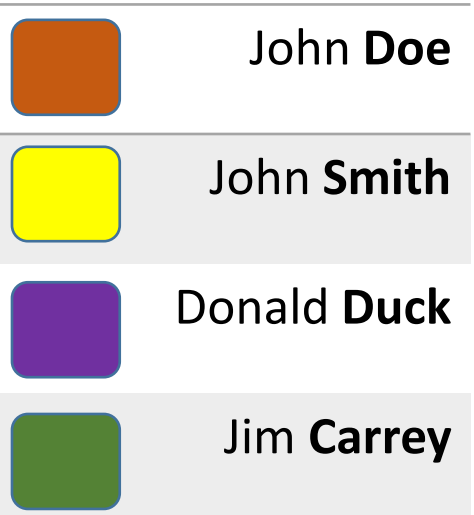
```
public class Person {
    private String name;
    private String surname;
    private Bitmap image;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
    public String toString() {
        return name + " " + surname;
    }
}
```

# Example of populating a list view

List 1 (array adapter):

| |
|---|
| John Doe |
| John Smith |
| Donald Duck |
| Jim Carrey |

List 2 (custom adapter):



# List View with ArrayAdapter

```
public class Person {
    public toString() {
        return name + " " + surname;
    }
}
```

In onCreate:

```
final ArrayAdapter<Person> pAdapter = new ArrayAdapter<>(this,
android.R.layout.simple_spinner_item, persons);

pListView.setAdapter(pAdapter);

pListView.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {

        public void onItemSelected(AdapterView<?> adapterView,
        View view, int index, long l) {…}

} );
```

# List View with custom adapter

```
private class PersonAdapter extends ArrayAdapter<Person> {

    @Override

    public View getView(int position, View convertView, ViewGroup
parent) {

// used to load any XML layout at runtime

LayoutInflater inflater = ((Activity) context).getLayoutInflater();

convertView = inflater.inflate(R.layout.person_list_entry, null);

TextView nameTextView = (TextView)
convertView.findViewById(R.id.person_name);

Person person = getItem(position);

nameTextView.setText(person.getName());

return convertView;

}
```

# ViewHolder pattern

Used to increase the speed at which a ListView renders data.

The reason for this improvement is that:

1. The number of times which the findViewById method is invoked is drastically reduced.
2. Existing views do not have to be garbage collected.
3. New views do not have to be inflated.

Internally, a ListView keeps a reference to views it has already seen. Not reusing the *convertView* field will continuously add new views to the ListView, causing a noticeable slowdown of your application and eventually lead to your application crashing (*OutOfMemoryException*).

# ViewHolder pattern

1. Create a ViewHolder for storing the views for each list entry inside the adapter class:

```
private static class ViewHolder {
    private TextView nameTextView;
    private TextView surnameTextView;
    private ImageView personImageView;
}
```

2. Inflate views and store them inside a new instance of ViewHolder.
3. Save ViewHolder in view's tag field (aka view cache)

# List View with custom adapter

```
public View getView(int position, View convertView, ViewGroup
parent) {
ViewHolder holder;
if (convertView == null) {
        convertView = inflater.inflate(R.layout.person_list_entry, null);
        holder = new ViewHolder();
        holder.nameTextView = (TextView)
        convertView.findViewById(R.id.person_name);
        holder.surnameTextView = (TextView)
        convertView.findViewById(R.id.person_surname);
        holder.personImageView = (ImageView)
        convertView.findViewById(R.id.person_image);
        convertView.setTag(holder);
}
else {
        holder = (ViewHolder) convertView.getTag();
};
}
```

# Grid View

Similar to a List View, the grid items are automatically inserted to the layout using a *ListAdapter*.

```
<GridView
xmlns:android="http://schemas.android.com/apk/res/an
droid"
    android:id="@+id/gridview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="90dp“
    android:numColumns="auto_fit“ // or integer
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

# Grid View column stretch mode

stretchMode possible values:

- columnWidth – each column is stretched equally.
- none – stretching is disabled.
- spacingWidth – the spacing between each column is stretched.
- spacingWidthUniform – the spacing between each column is uniformly stretched.

# Overwriting default layouts

E.g.,

`res/layout/activity1.xml`

`res/strings.xml`

// used just for landscape orientation

`res/layout-land/activity1.xml`

// used for Romanian language only

`res/strings-ro.xml`

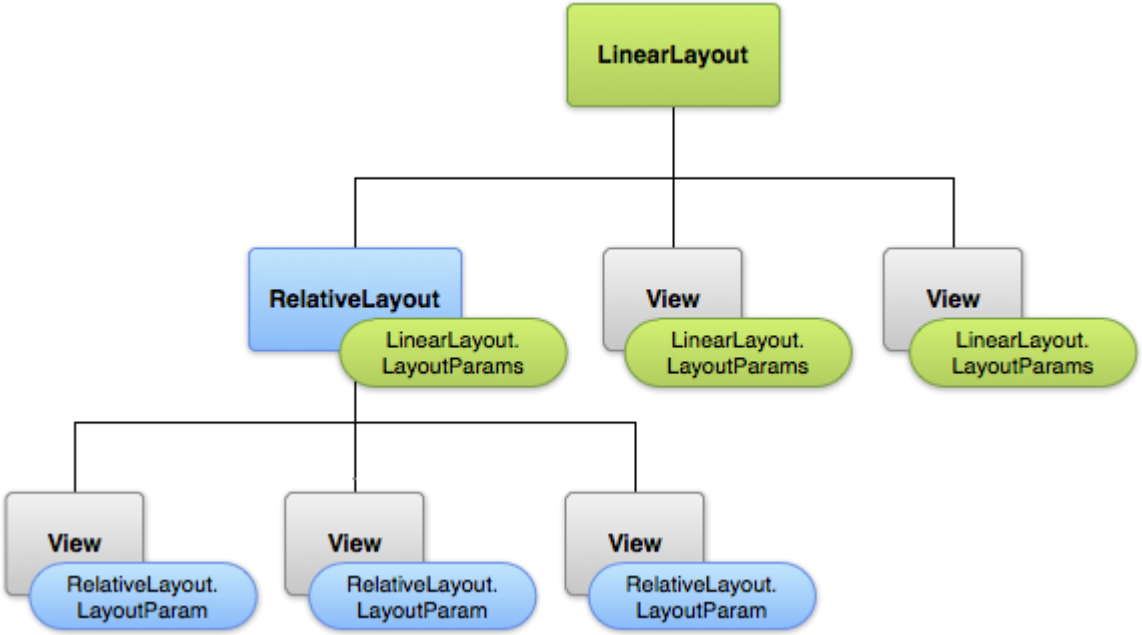
// used solely on 7” & 10” tablets

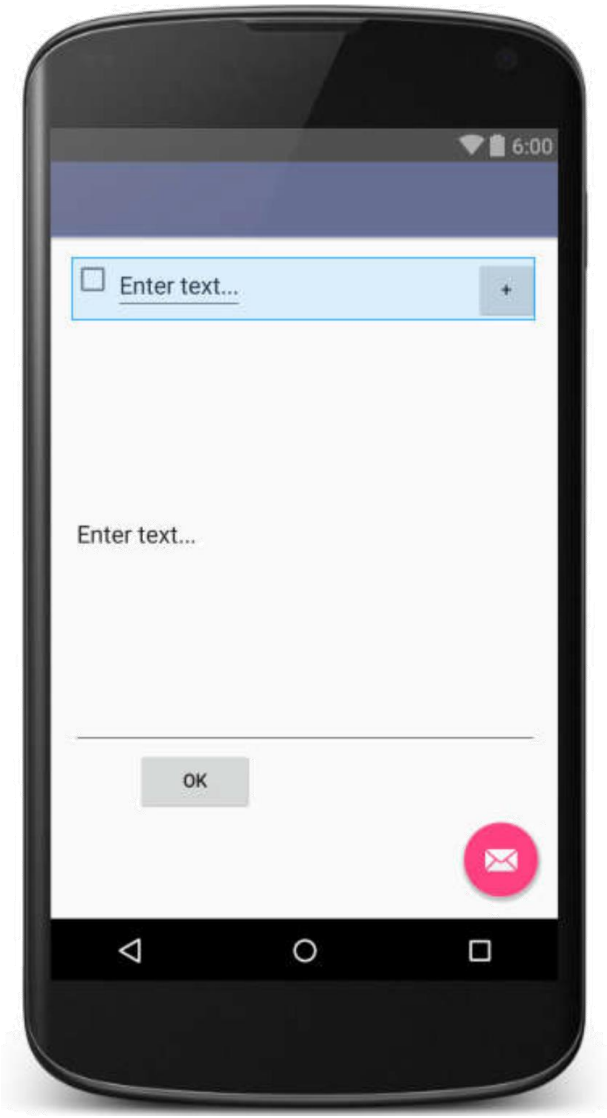
`res/layout-sw600dp/activity1.xml`

`res/layout-sw720dp/activity1.xml`

# GUI design alternatives

1. Programmatic (code, runtime)
2. Graphical (designer drag and drop)
3. Declarative (XML)



# UI design alternatives

| Declarative | Programmatic |
| --- | --- |
| XML tags with properties | Java code can do anything XML can |
| Easy to read | Harder to understand |
| WYSIWYG* tools | Object and method calls |
| Static | Dynamic |
| Slow to parse | More performant |

*what you see is what you get

# GUI declarative definition

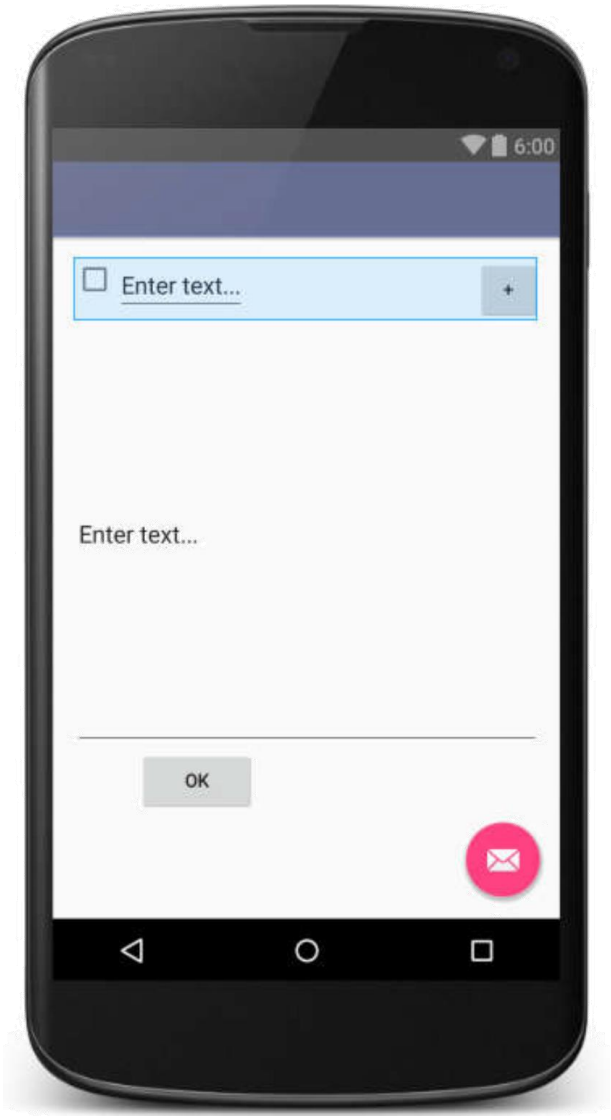
An XML-based layout specifies:

- A model for specifying UI components and the relation between them and towards the container.
- Considered resources
- Contains a hierarchy of elements and their properties.

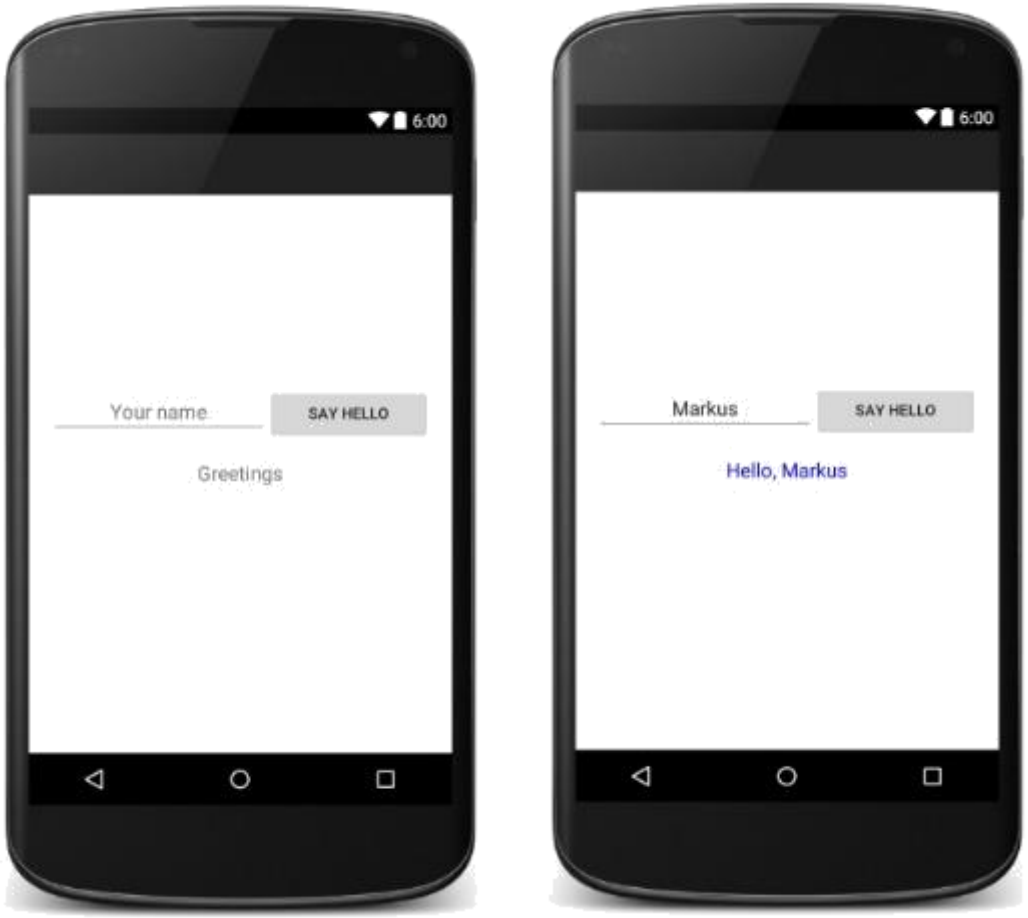
```
<RelativeLayout
   android:layout_width="fill_parent"
   android:layout_height="wrap_content">
   <CheckBox
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_alignParentLeft="true"
          android:text=""
          android:id="@+id/checkBox" />
   <EditText
          android:id="@+id/text1"
          android:text="@string/default_text"
          android:focusable="false"
          android:layout_toRightOf="@id/checkBox"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"/>
   <Button
          style="?android:attr/buttonStyleSmall"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="+"
          android:layout_alignParentRight ="true"
          android:id="@+id/button2"
          android:layout_gravity="right" />
</RelativeLayout>
```

# GUI programmatic definition

```
setContentView(R.layout.activity_main);
EditText tb = new EditText(this);
tb.setText(R.string.default_text);
tb.setFocusable(false);
tb.setLayoutParams(widgetParams);
LinearLayout layout = (LinearLayout)
findViewById(R.id.layout1);
layout.addView(tb);
```



# UI design alternatives example



# UI design alternatives example

| Declarative | Programmatic |
|---|---|
| Define XML view elements in the *activity_main.xml* file in *res/layout* | Writing Java code in the *onCreate* method of the activity. |
| Edit file *res/strings.xml*:<br><string name="app_name"><br>Hello World</string> | // set title bar<br>this.setTitle("Hello World"); |
| < RelativeLayout<br>xmlns:android="…"<br>android:layout_width="match_parent"<br>android:layout_height="match_parent"<br>android:gravity=<br>"center_vertical\|center_horizontal" ><br></ RelativeLayout> | // define a relative layout<br>RelativeLayout mainLayout;<br>mainLayout = new RelativeLayout(this);<br><br>// center contents<br>mainLayout.setGravity(<br>Gravity.CENTER_HORIZONTAL \|<br>Gravity.CENTER_VERTICAL); |

# UI design alternatives example

| Declarative | Programmatic |
| --- | --- |
| <LinearLayout<br>android:orientation="horizontal"<br>android:layout_width="match_parent"<br>android:layout_height="match_parent"><br></LinearLayout> | // define a linear layout for the first two elements (first row)<br>LinearLayout mainLayout;<br>secondLayout = new LinearLayout(this);<br>// horizontal orientation<br>secondLayout.setOrientation(0); |
| <EditText<br>android:layout_width="wrap_content"<br>android:layout_height="wrap_content"<br>android:hint="Your name"<br>android:layout_weight=”0.6”<br>/> | EditText eName = new EditText (this);<br>eName.setHint(“Your name");<br>// dimensions<br>LinearLayout.LayoutParams btnLayoutParams = new LinearLayout.LayoutParams(<br>ViewGroup.LayoutParams.WRAP_CONTENT,<br>ViewGroup.LayoutParams.WRAP_CONTENT);<br>// set these params<br>eName.setLayoutParams(btnLayoutParams); |

# UI design alternatives example

| Declarative | Programmatic |
| --- | --- |
| <Button<br>android:layout_width="wrap_content"<br>android:layout_height="40dp"<br>android:text="Say hello"<br>android:layout_weight=”0.4”<br>android:onClick=”clicked”<br>/> | Button bName = new Button(this);<br>bName.setText("Say hello");<br>// define dimension<br>LinearLayout.LayoutParams btnLayoutParams = new<br>LinearLayout.LayoutParams(<br>ViewGroup.LayoutParams.WRAP_CONTENT, 40);<br>// set params<br>bName.setLayoutParams(btnLayoutParams); |
| <TextView<br>android:layout_width="wrap_content"<br>android:layout_height="wrap_content"<br>android:hint="Greetings"<br>android:centerInParent=”true”/> | TextView tName = new Label(this);<br>tName.setHint(“Greetings");<br>LinearLayout.LayoutParams btnLayoutParams = new<br>LinearLayout.LayoutParams(<br>ViewGroup.LayoutParams.WRAP_CONTENT,<br>ViewGroup.LayoutParams.WRAP_CONTENT);<br>tName.setLayoutParams(btnLayoutParams); |

# UI design alternatives example

| Declarative | Programmatic |
| --- | --- |
| The <EditText> and <Button> elements are placed between the <LinearLayout> tags, which, in turn, is placed inside <RelaiveLayout>, alongside the <TextView>. | // add controls to layout<br>secondLayout.addView(eName);<br>secondLayout.addView(bName);<br>mainLayout.addView(secondLayout);<br>mainLayout.addView(tName); |
| // attach activity layout<br>this.setContentView(R.layout.activity_main); | // attach activity layout<br>this.setContentView(mainLayout); |