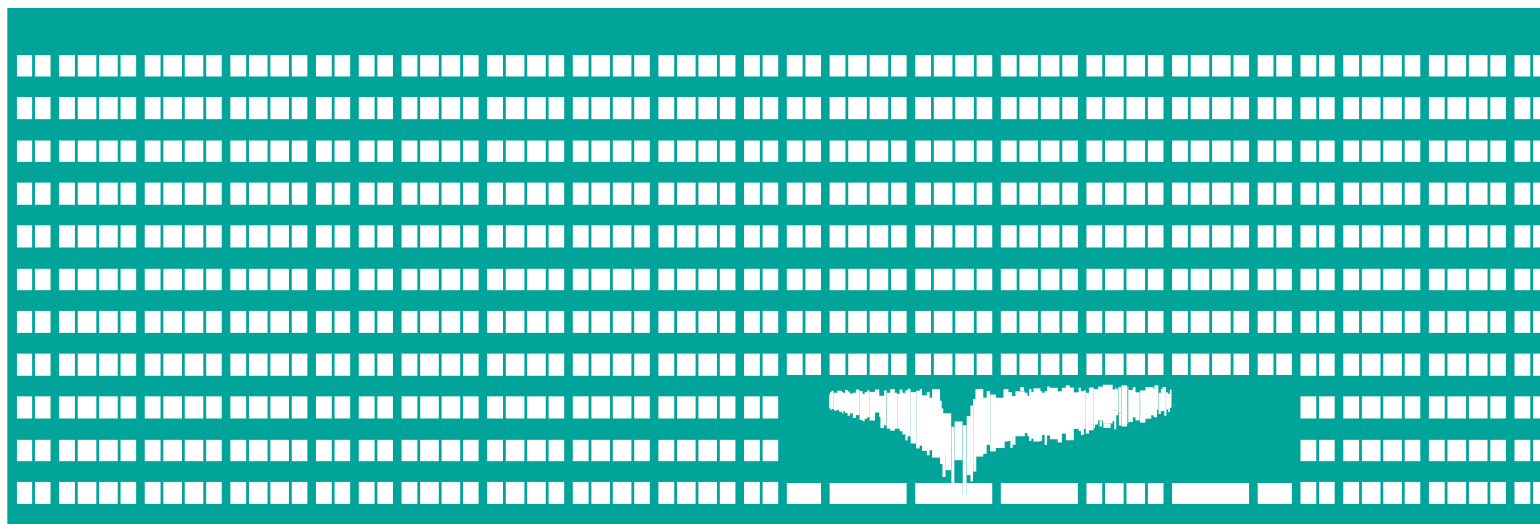


VŠB TECHNICKÁ
UNIVERZITA
OSTRAVA

VSB TECHNICAL
UNIVERSITY
OF OSTRAVA



www.vsb.cz

Android – Basic User Interface

Michal Krumnikl

UI Development in Android

- Screen area is limited.
- The user usually wants to do one specific action.
- Similar to other SDKs, the Android SDK provides text fields, buttons, lists, grids, and so on.
- In addition, Android provides a collection of controls that are appropriate for mobile devices.

„Android users expect UI elements to act in certain ways, so it's important that your app be consistent with other Android apps. To satisfy your users, create a layout that gives users predictable choices.“

Android Device Fragmentation



UI Nomenclature

- **View, Widget, Control**

- Each of these represents a user interface element.
- Every element of the screen is a View.
- Examples include a button, a grid, a list, a window, a dialog box, etc.

- **Container**

- This is a view used to contain other views. For example, a grid can be considered a container because it contains cells, each of which is a view.
- Relationship is parent-child (Container - View)

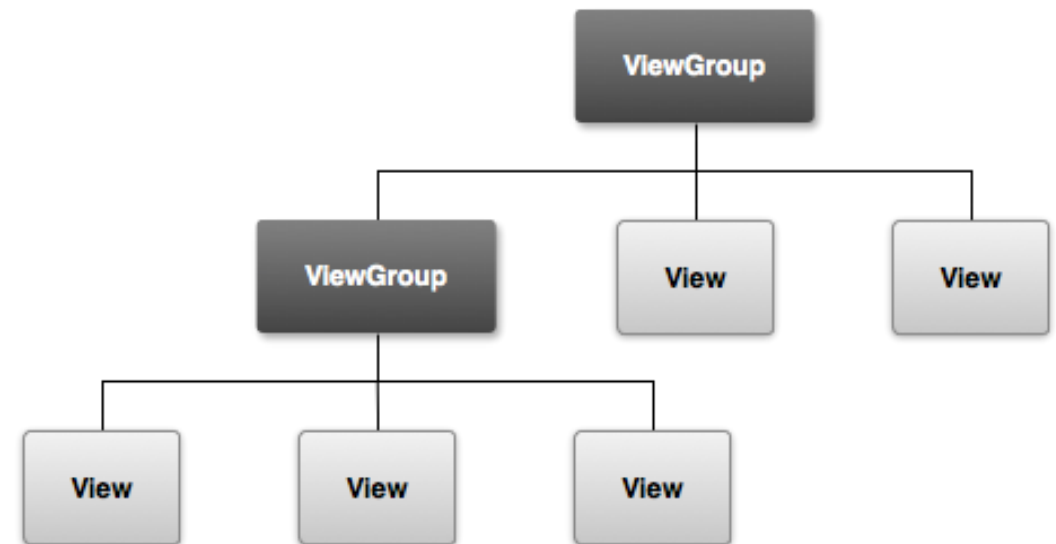
- **Layout**

- This is an XML file used to describe a view

Common Controls

- Hierarchy of **View** and **ViewGroup**

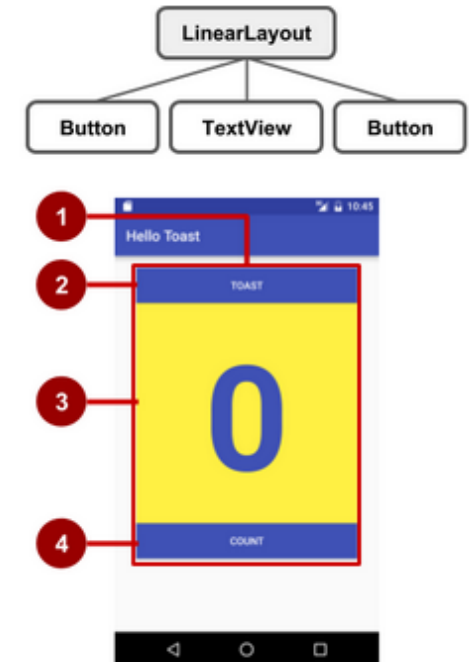
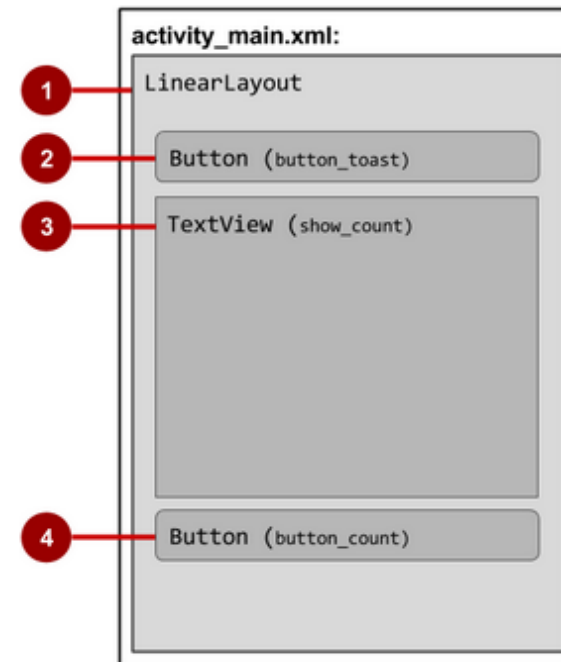
- **View** class represents a general-purpose View object, extending the View class.
- **ViewGroup** is also a view, but holds other views. ViewGroup is the base class for a list of layout classes.
- Android, like Swing, uses the concept of **layouts** to manage how controls are laid out within a container view.
- Any View object may have an **integer ID** associated with it, to uniquely identify the View within the tree.



Common Controls

- Hierarchy of **View** and **ViewGroup**

- View** class represents a general-purpose View object, extending the View class.
- ViewGroup** is also a view, but holds other views. ViewGroup is the base class for a list of layout classes.
- Android, like Swing, uses the concept of **layouts** to manage how controls are laid out within a container view.
- Any View object may have an **integer ID** associated with it, to uniquely identify the View within the tree.




View ID

- When the application is compiled, the **ID** is referenced as an **integer (in R.java)**
 - **@** at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string.
 - **+** means that this is a new resource name that must be created and added to our resources (in the R.java file).
 - Defining IDs for view objects is important when creating a RelativeLayout.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching.


```
<TextView android:id="@+id/text"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="I am a TextView" />
```



View ID Instancing

- **Common approach**
 - View in XML layout file

```
<TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="I am a TextView" />
```



- Getting an instance of the View

```
TextView myText = (TextView)findViewById(R.id.text);
```

- **View Binding**

- Since Android Studio 3.6 and higher provides compile-time type safety for code that interacts with views

```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

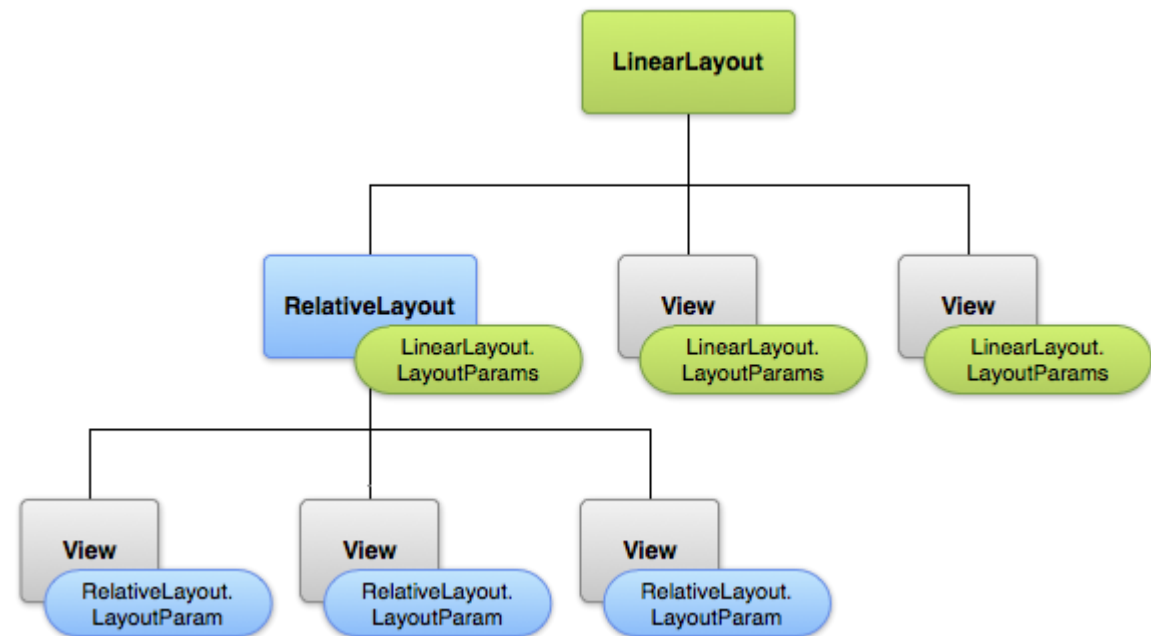
```
private ResultProfileBinding binding;

binding = ResultProfileBinding.inflate(
    getLayoutInflater());

binding.myText.setText(...)
```

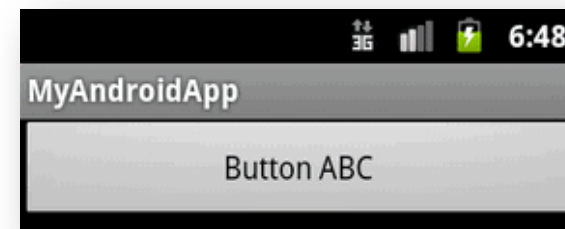
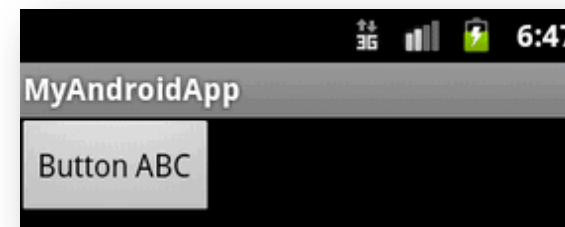
Approaches to build UI

- Originally
 - Construct user **interfaces entirely in code**.
 - Define user **interfaces in XML**.
 - Define the user interface in XML and then refer to it, and modify it, in code.
 - For debug use **Layout Inspector**
- Recently
 - **Jetpack Compose back to declarative API**



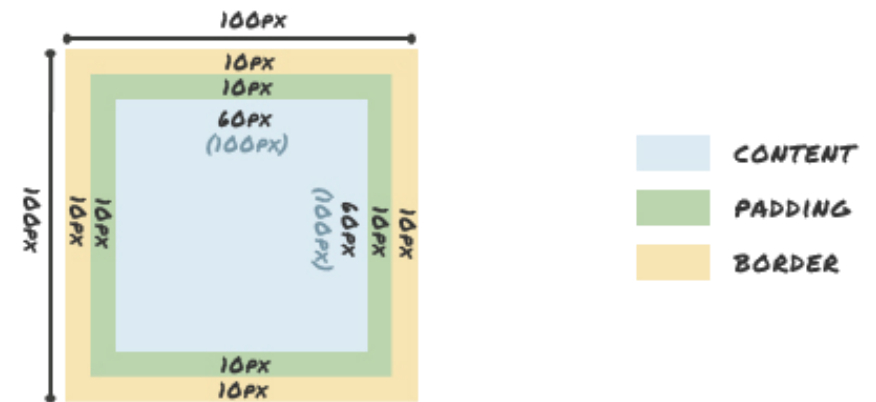
View Dimensions

- The geometry of a view is a **rectangle**.
- You can specify width and height with exact measurements, but this is not frequently used.
 - ***wrap_content*** tells your view to size itself to the dimensions required by its content
 - ***fill_parent*** (renamed ***match_parent*** in API 8) tells your view to become as big as its parent view group will allow.
- Specifying a layout width and height using absolute units such as pixels is not recommended.
- Using relative measurements such as **density-independent pixel units (*dp*)** is better approach.



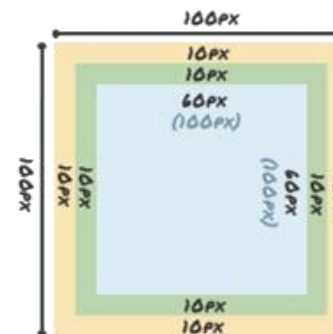
View Position

- A view has a location, expressed as a pair of **left** and **top** and two dimensions, expressed as a **width** and a **height**.
- It is possible to retrieve the location of a view by invoking the methods *getLeft()* and *getTop()*.
 - To avoid unnecessary computations, namely *getRight()* and *getBottom()*.



View Size, Padding and Margins

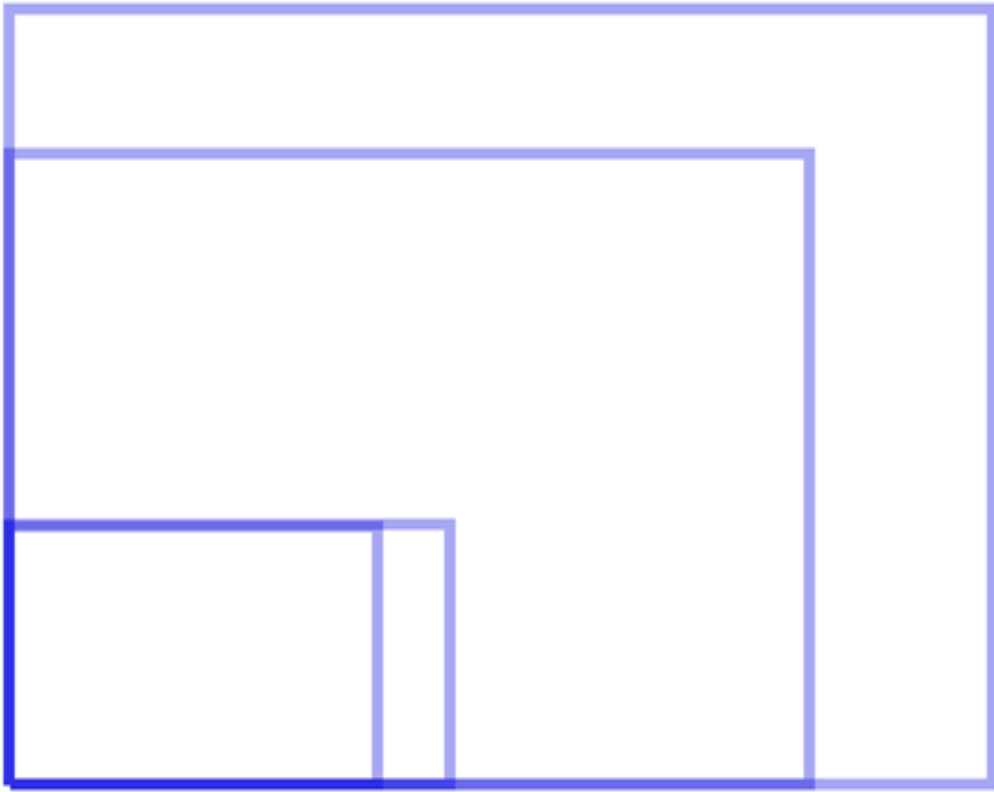
- The size is expressed with a width and a height
 - **Measured width** and **measured height**. These dimensions define how big a view wants to be within its parent - *getMeasuredWidth()* and *getMeasuredHeight()*.
 - **Drawing width** and **drawing height**. These dimensions define the actual size of the view on screen, at drawing time and after layout - *getWidth()* and *getHeight()*.
- **Padding** can be used to offset the content of the view by a specific amount of pixels.
 - Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support
 - Padding can be set using the *setPadding(int, int, int, int)*



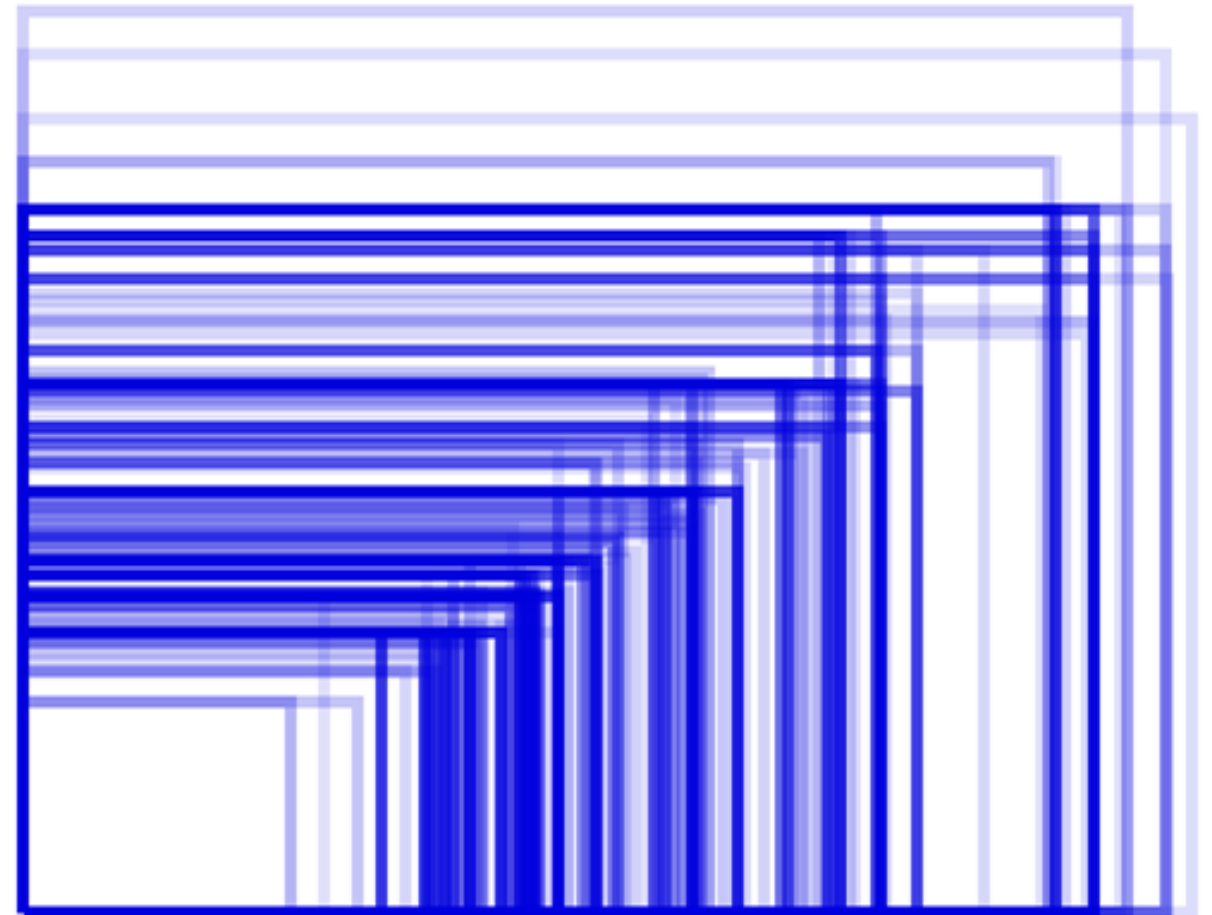
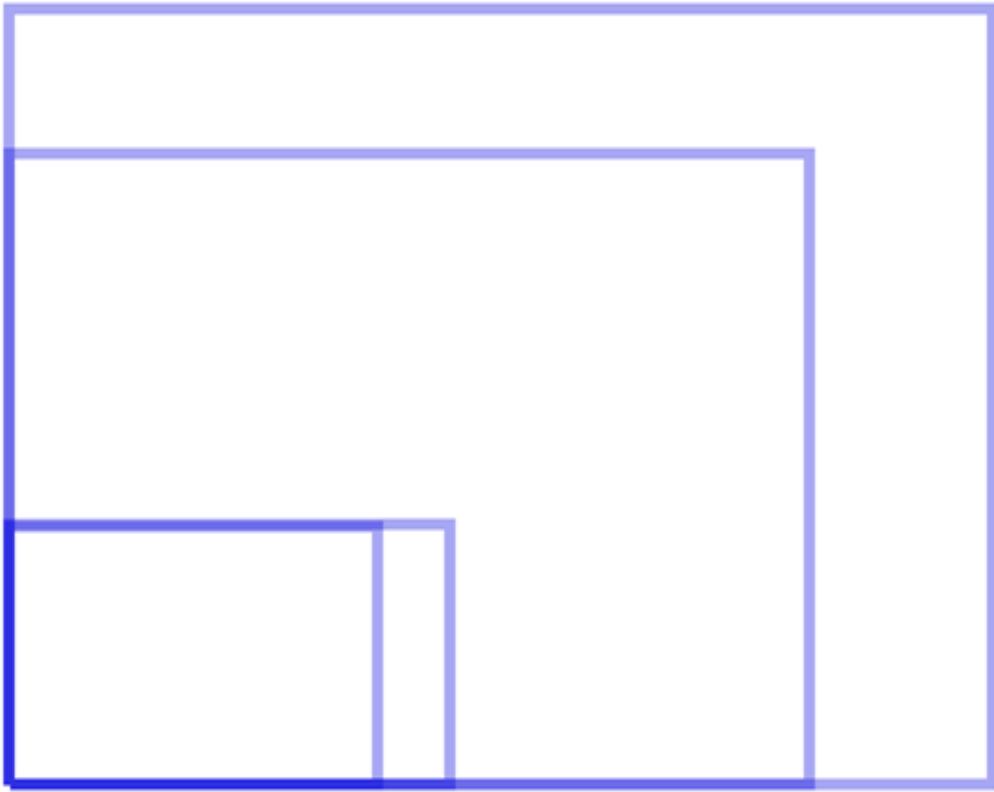
Implementing Your Own View

- **Override *onDraw()***
 - The parameter to *onDraw()* is a **Canvas** object that the view can use to draw itself. The **Canvas** class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in *onDraw()* to create your custom user interface (UI).
- **Handle Layout Events**
 - Often needed to perform multiple layout calculations depending on the size and shape of their area on screen.
 - ***onSizeChanged()*** is called when your view is first assigned a size, and again if the size of your view changes for any reason.
 - More finer control over the view's layout parameters can be implemented by ***onMeasure()***. This method's parameters are values that tell you how big your view's parent wants your view to be, and whether that size is a hard maximum or just a suggestion.

Screen Resolutions

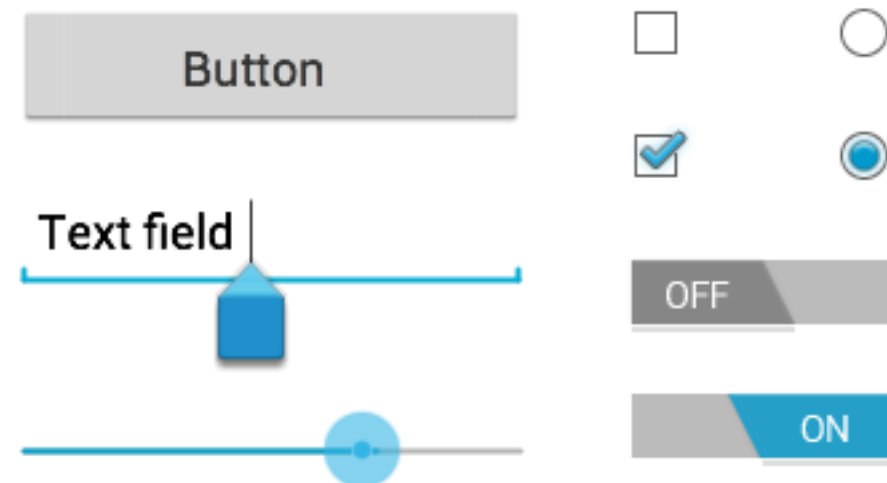


Screen Resolutions



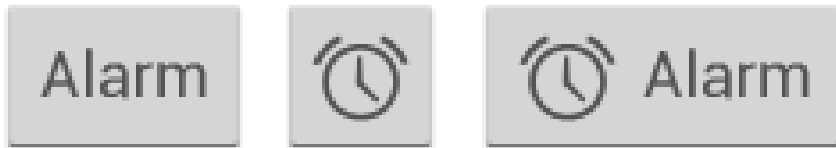
Basic Input Controls

- **Button** – *Button*
- **Text field** – *EditText*, *AutoCompleteTextView*
- **Checkbox** – *CheckBox*
- **Radio button** – *RadioGroup*, *RadioButton*
- **Toggle button** – *ToggleButton*
- **Spinned** – *Spinner*
- **Pickers** – *DatePicker*, *TimePicker*



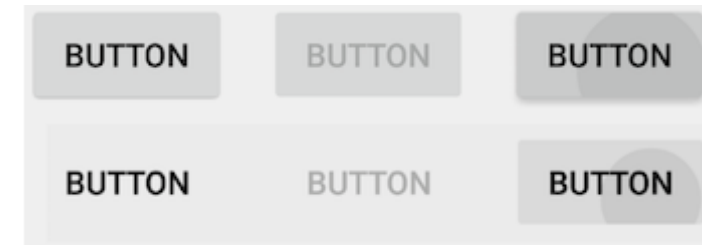
Button

- Consists of text or an icon (or both)
 - Button, ImageButton class



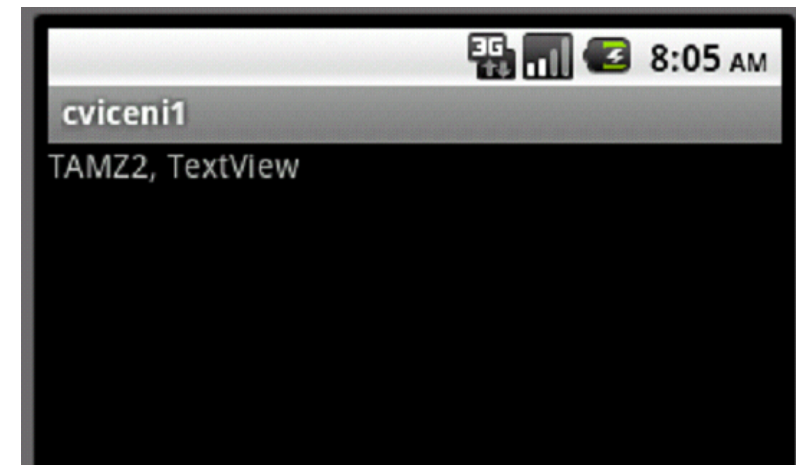
- When the user clicks a button, the Button object receives an **on-click event**.
 - Using android:onClick attribute
 - Using an OnClickListener (must be used for buttons created at runtime)

- The appearance of your button (background image and font) may vary. Controls are styled using a **theme**
 - e.g. android:theme="@android:style/Theme.Holo"



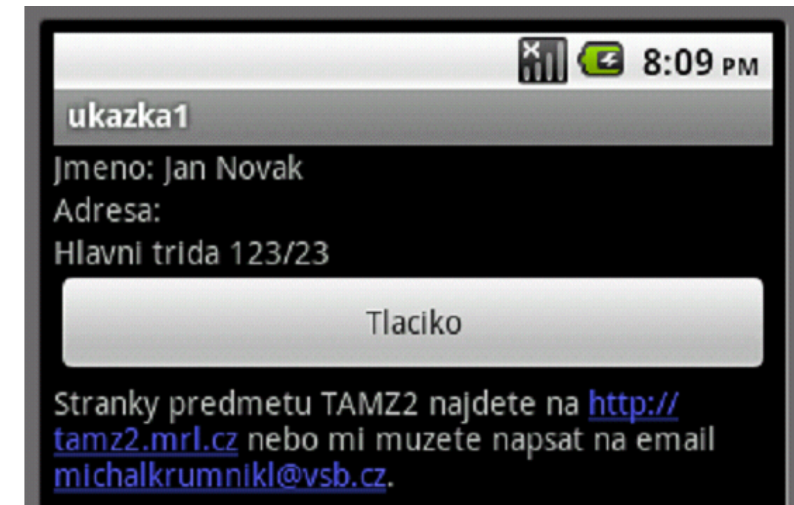
TextView

- The simplest widget representing the label. The text cannot be edited directly by user.
- Label is created in XML layout files by adding TextView element. Some of the numerous properties:
 - android:typeface – Sets the typeface (e.g. monospace)
 - android:textStyle – Type of typeface (e.g. bold, italic)
 - android:textColor – Color of the label's text
 - android:text – Value of the label itself (the text)



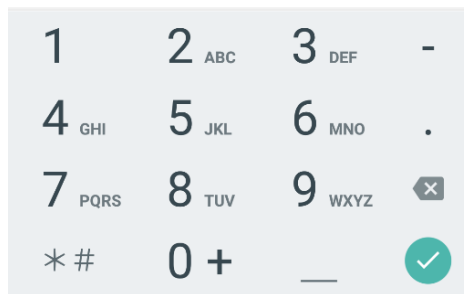
TextView

- You can pass a TextView to the **Linkify** class to find and add links to the content of the TextView.
- Use *addLinks()* method of Linkify to pass the TextView and a mask indicating what types of links that Linkify should look for. Linkify can create links for text that looks like a phone number, an e-mail address, a web URL, or a map address



EditText

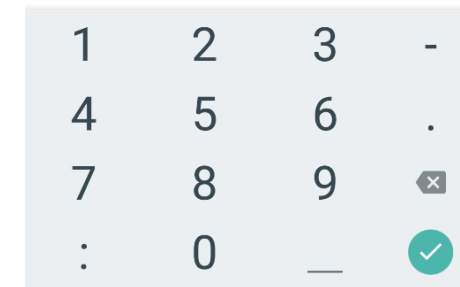
- The default behavior of the EditText control is to display text on **one line and expand** as needed.
- You can force the user to a single line by setting the **singleLine** property to true.
- You can specify the type of keyboard you want for your EditText object with the **android:inputType** attribute.



android:inputType="phone"



android:inputType="textPassword"



android:inputType="time"

EditText

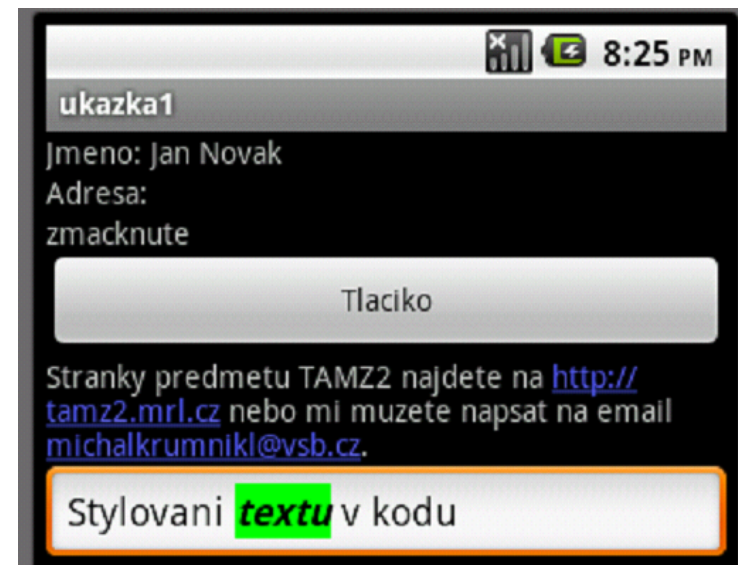
- Fields are implemented via the EditText widget, which is a **subclass of the TextView**.
- Some additional properties:
 - android:autoText – automatic spelling assistance.
 - android:capitalize
 - android:digits
 - android:singleLine
- Input is obtained using **getText()** method

```
<EditText android:id="@+id/EditText01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    android:capitalize="words"
    android:singleLine="false">
</EditText>
```

EditText Styling

- Statically, you can apply markup directly to the strings in your string resources


```
<string name="styledText">
  <i>Static</i>
</string>
```
- Styling an EditText control's content programmatically requires a little additional work but allows for much more flexibility.



AutoCompleteTextView

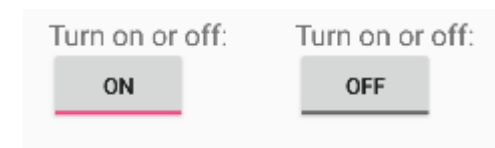
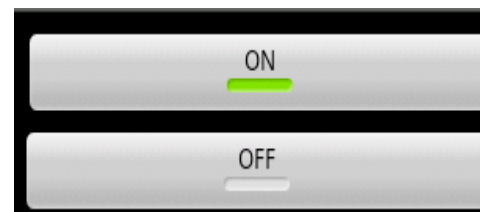
- TextView with **auto-complete functionality**.
- As the user types in the TextView, the control can display suggestions for the user to select.
- To use a control like this, you have to create the control, create the list of suggestions, tell the control the list of suggestions, and possibly tell the control how to display the suggestions.

```

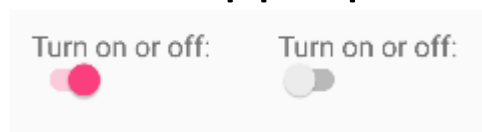
ArrayAdapter<String> aa = new
    ArrayAdapter<String>(this,
        android.R.layout.simple_dropdown_item_1line,
        new String[] {"English", "Hebrew", "Hindi"});
actv.setAdapter(aa);
    
```


ToggleButton

- The ToggleButton, like a check box or a radio button, is a **two-state button**. This button can be in either the On state or the Off state.
- The default behavior also sets the button's text to "On" when it's in the On state and "Off" when it's in the Off state.
- You can modify the text for the ToggleButton if On/Off is not appropriate.
- Similar **Switch** class -

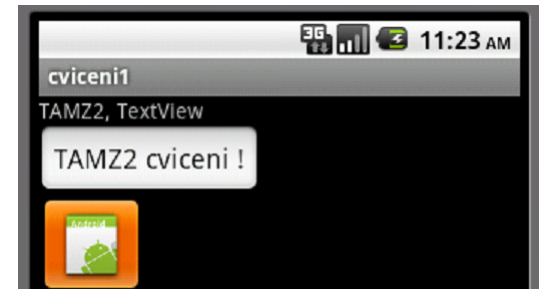


```
<ToggleButton android:id="@+id/cctglBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Run"
    android:textOff="Stop"
    android:text="Toggle Button"/>
```



ImageView and ImageButton

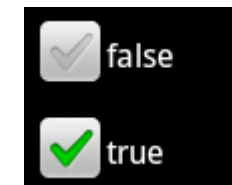
- ImageView is a widget that enables to embed images in activities. It is analogous to TextView
 - android:src attribute specify the image
 - Image content can be also on a Uri using *setImageURI()*
- ImageButton is a subclass of ImageView mixed with the standard Button behavior.
- Action is assigned in the same way as in Button



```
<ImageButton
    android:id="@+id/ImageButton01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/icon">
</ImageButton>
```

CheckBox

- The classis check box has two states:
 - checked and unchecked
- In the program code, you can invoke following
 - *isChecked()*, *setChecked()*, *toggle()*
- To be notified when the state changes, you can register a listener object *OnCheckedChangeListener*



```
CheckBox cb1 =  
    (CheckBox) this.findViewById(R.id.CheckBox01) ;  
  
cb1.setOnCheckedChangeListener(this) ;  
  
public void onCheckedChanged(CompoundButton  
    buttonView, boolean isC) {  
    TextView tv1 =  
        (TextView) this.findViewById(R.id.TextView01) ;  
    if (isC) tv1.setText("checked") ;  
    else tv1.setText("unchecked") ;  
}
```

RadioButton

- RadioButtons are **grouped inside a RadioGroup**.
- Assigning an *android:id* to RadioGroup allows to access the group from the code.
- RadioGroup is initially set to that none of the buttons are checked. To preset one of the buttons, use either *setChecked()* or *check()* on the RadioGroup withing *onCreate()* callback.



ListView / RecyclerView

- The user sees a list of items and can scroll through them.
- Class *ExpandableListView* supports also a grouping of items.
- The items in the list can be of any type.
- An **adapter** is used for managing the items in the list (the data model or data source).



ListView / RecyclerView - Adapter

- Manages the data model and adapts it to the individual rows in the list view.
 - An adapter extend the *BaseAdapter* class.
- The adapter would inflate the layout for each row in its ***getView()*** method and assign the data to the individual views in the row.
- The adapter is assigned to the *ListView* via the ***setAdapter*** method on the *ListView* object.

Android provides default adapter implementations; the most important are *ArrayAdapter* and *CursorAdapter*.



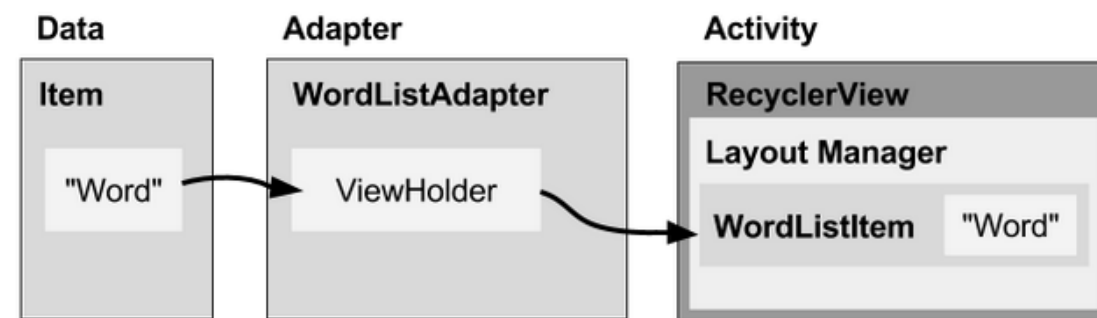
RecyclerView

- More advanced and flexible version of *ListView*
 - Minimize the number of View items that exist at any given point
 - Minimize the number of View items you have to create
- Reuse the View items with different data as list items scroll in and out of the display.

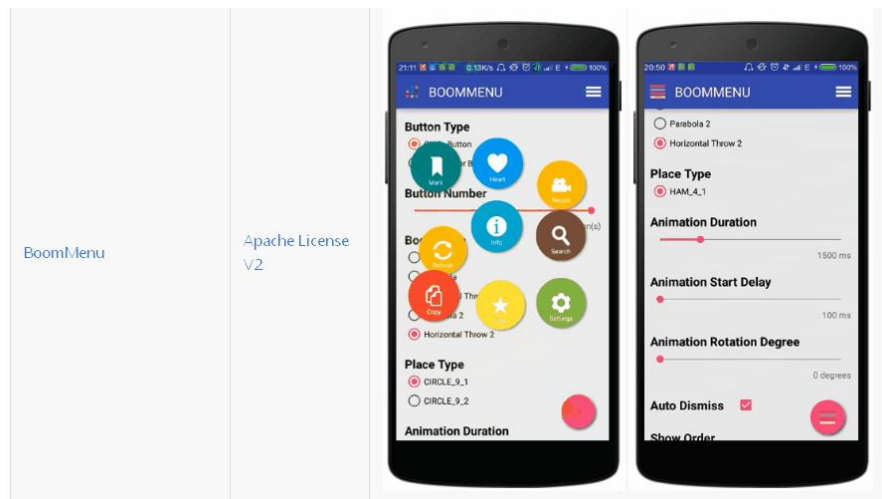


RecyclerView

1. Add the RecyclerView to the Activity layout.
2. Create a layout XML file for one View item.
3. Extend **RecyclerView.Adapter** and implement the *onCreateViewHolder()* and *onBindViewHolder()* methods.
4. Extend *RecyclerView.ViewHolder* to create a ViewHolder for your item layout. You can add click behavior by overriding the *onClick()* method.
5. In the Activity, inside the *onCreate()* method, create a *RecyclerView* and initialize it with the adapter and a layout manager.



More Awesome UI



ActionBar

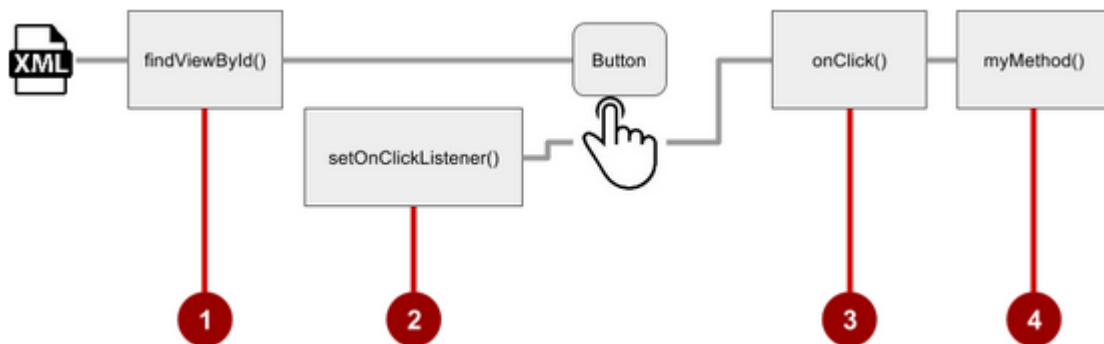
Name	License	Demo
ActionBar	Apache License V2	

<https://github.com/wasabeef/awesome-android-ui>

UI Events

- An event listener is an interface in the View class that contains a single callback method.
- Listener has to be registered** using *View.set...Listener()* method on the specific View.

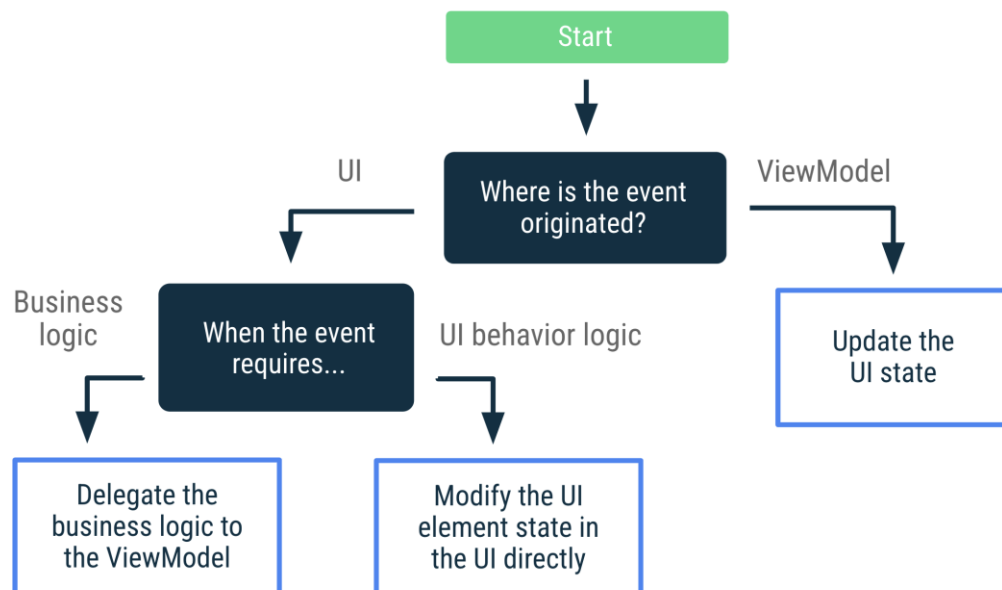
```
private OnClickListener mListener =
    new OnClickListener() {
        public void onClick(View v) {
            myMethod(); // do something
        }
    };
```



```
protected void onCreate(Bundle savedInstanceState) {
    Button button =
        (Button) findViewById(R.id.btn);
    button.setOnClickListener(mListener);
    ...
}
```

UI Events in Compose

- **UI can handle user events directly** if those events relate to modifying the state of a UI element.
- If the event **requires performing business logic**, it should be processed by the **ViewModel**.



@Composable

```
fun LatestNewsScreen(viewModel: LatestNewsViewModel = viewModel()) {
    // State of whether more details should be shown
    var expanded by remember { mutableStateOf(false) }
```

Column {

```
    Text("Some text")
    if (expanded) {
        Text("More details")
    }
```

Button(

```
    // The expand details event processed by UI that
    // modifies this composable's internal state.
```

```
    onClick = { expanded = !expanded }
```

```
) {
```

```
    val expandText = if (expanded) "Collapse" else "Expand"
    Text("$expandText details")
```

```
}
```

```
// Refresh event is processed by ViewModel that is in charge
// of the UI's business logic.
```

```
Button(onClick = { viewModel.refreshNews() }) {
    Text("Refresh data")
```

```
}
```

```
}
```

```
}
```

Common Callback Methods

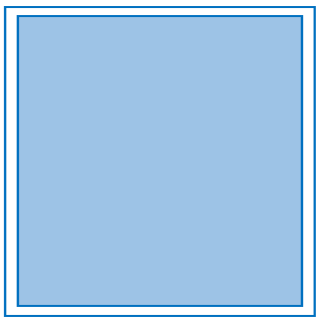
- **onClick()** - View.OnClickListener
- **onLongClick()** - View.OnLongClickListener
 - returns a boolean to indicate whether you have consumed the event
- **onFocusChange()** - View.OnFocusChangeListener
- **onKey()** - View.OnKeyListener
 - returns a boolean to indicate whether you have consumed the event
- **onTouch()** - View.OnTouchListener
 - returns a boolean to indicate whether you have consumed the event
- **onCreateContextMenu()** - OnCreateContextMenuListener
 - result of a sustained "long click"

Containers – Layout Managers

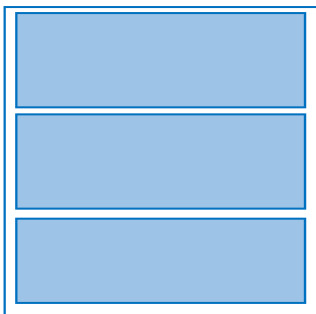
- Containers pour a collection of widgets (and possibly child containers) into specific structures. If you want a form with labels on the left and fields on the right, you need a container.
- These **container classes** are called layouts (or layout managers), and implements a specific strategy to manage the size and position of its children.
- **LinearLayout** - Organizes its children either horizontally or vertically
- **TableLayout** - Organizes its children in tabular form
- **RelativeLayout** - Organizes its children relative to one another or to the parent
- **FrameLayout** - Allows you to dynamically change the control(s) in the layout

Containers – Layout Managers

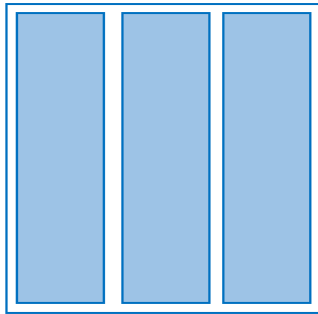
- Layout implements a specific strategy to manage the size and position of its children.



Frame layout



Linear layout (hor.)



Linear layout (vert.)

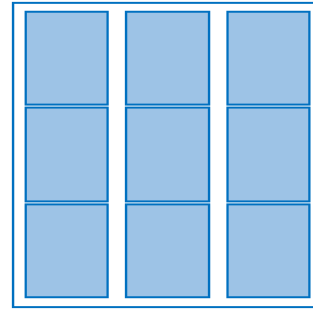
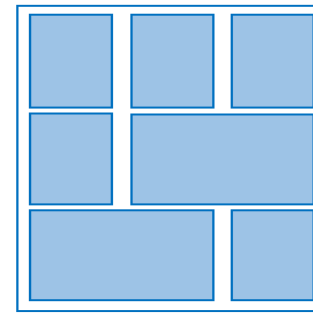
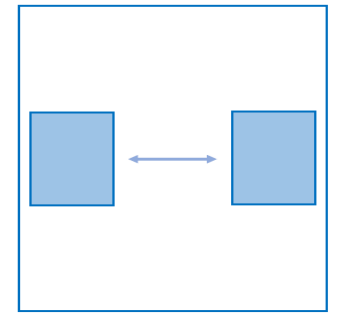


Table layout



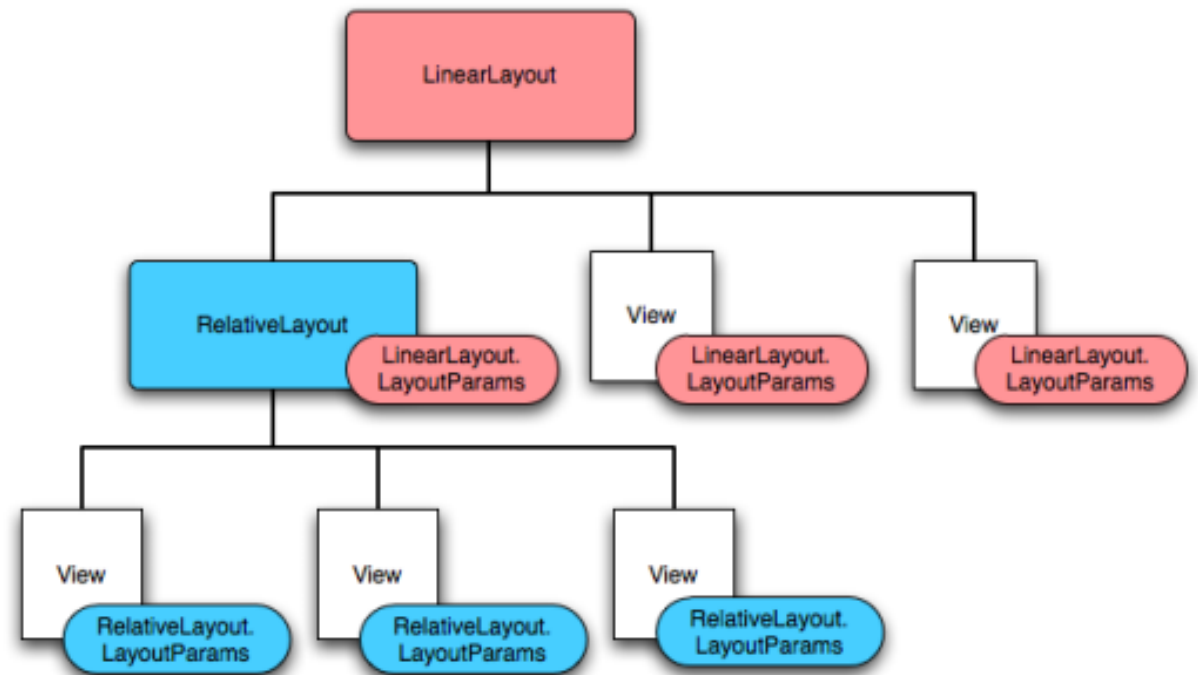
Grid layout (hor.)



Relative layout (vert.)

Layout Parameters

- 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.



LinearLayout

- The LinearLayout is the most basic layout. This layout manager organizes its children either **horizontally** or **vertically** based on the value of the orientation property.
- Fill model – fit the widget to “natural” dimensions.
- Controls include **weight** and **gravity**. You use **weight** to assign size importance to a control relative to the other controls in the container.

Gravity is essentially alignment. For example, if you want to a label’s text to the right, you would set its gravity to right.

Padding defines whitespace between widgets.



RelativeLayout

- RelativeLayout, as the name suggests, lays out widgets based on their **relationship** to other widgets in the container and the parent container. You can place widget X below and to the left of widget Y, have widget Z's bottom edge align with the bottom of the container, and so on.
- Ways of controlling the appearance
 - Positions Relative to Container
 - Positions Relative to Other Widgets
 - Order of Evaluation
- The defined RelativeLayout parameters
 - width, height, **below**, **alignTop**, **toLeft**
 - padding[Bottom | Left | Right | Top]
 - margin[Bottom | Left | Right | Top]



RelativeLayout

```
<?xml version="1.0" encoding="utf-8"?>
```

<RelativeLayout

```
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@drawable/blue"
        android:padding="10px" >

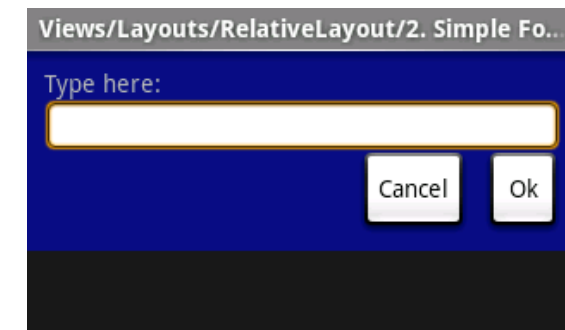
    <TextView android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Type here:" />

    <EditText android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/bcg"
        android:layout_below="@id/label" />
```

```
<Button android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
    android:layout_alignParentRight="true"
    android:layout_marginLeft="10px"
    android:text="OK" />
```

```
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
```

</RelativeLayout>



TableLayout

- TableLayout allows you to position your widgets in a **grid** to your specifications.
- You control the number of rows and columns, which columns might shrink or stretch to accommodate their contents, and so on.
- Rows are declared by the developer, by putting widgets as children of a TableRow inside the overall TableLayout. You, therefore, control **directly** how many **rows** appear in the table.
- The number of columns is determined by Android; you control the number of **columns** in an **indirect** fashion. First, there will be at least one column per widget in your longest row.

TableLayout

```

<?xml version="1.0" encoding="utf-8"?>
  <TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">

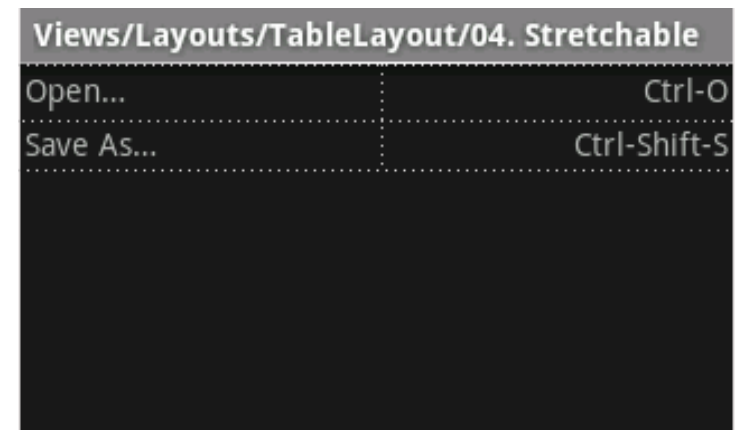
    <TableRow>
      <TextView
        android:text="@string/table_layout_4_open"
        android:padding="3dip" />
      <TextView
        android:text="@string/table_layout_4_o_sh"
        android:gravity="right"
        android:padding="3dip" />
    </TableRow>
  </TableLayout>

```

```

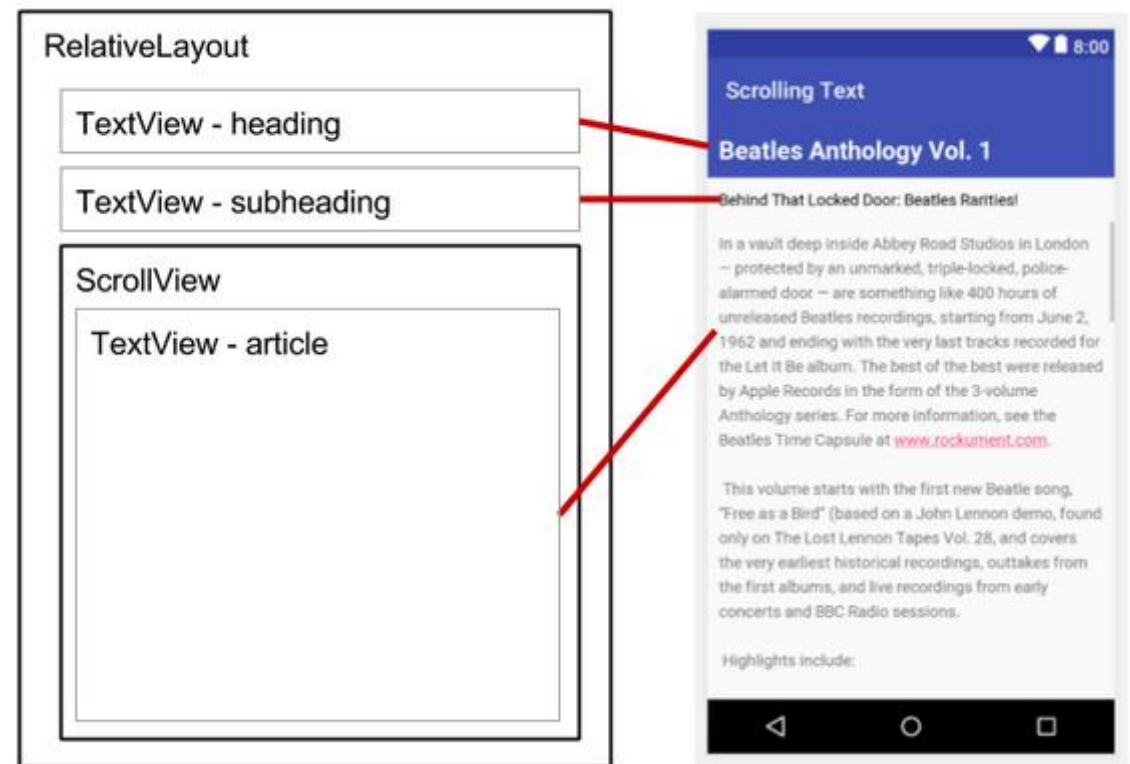
<TableRow>
  <TextView
    android:text="@string/table_layout_4_save"
    android:padding="3dip" />
  <TextView
    android:text="@string/table_layout_4_st"
    android:gravity="right"
    android:padding="3dip" />
</TableRow>
</TableLayout>

```



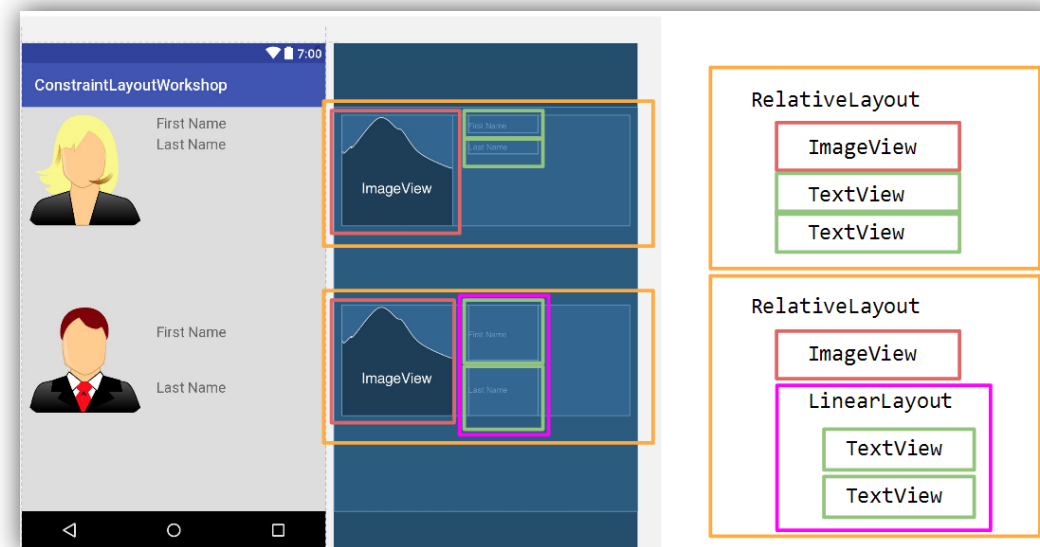
ScrollView

- ScrollView is a container that **provides scrolling** for its contents. You can take a layout that might be too big for some screens, wrap it in a ScrollView, and still use your existing layout logic.
- It just so happens that the user can see only part of your layout at one time.
 - Android 1.5 introduced HorizontalScrollView, which works like ScrollView, but horizontally. This can be useful for forms that might be too wide rather than too tall.



ConstraintLayout

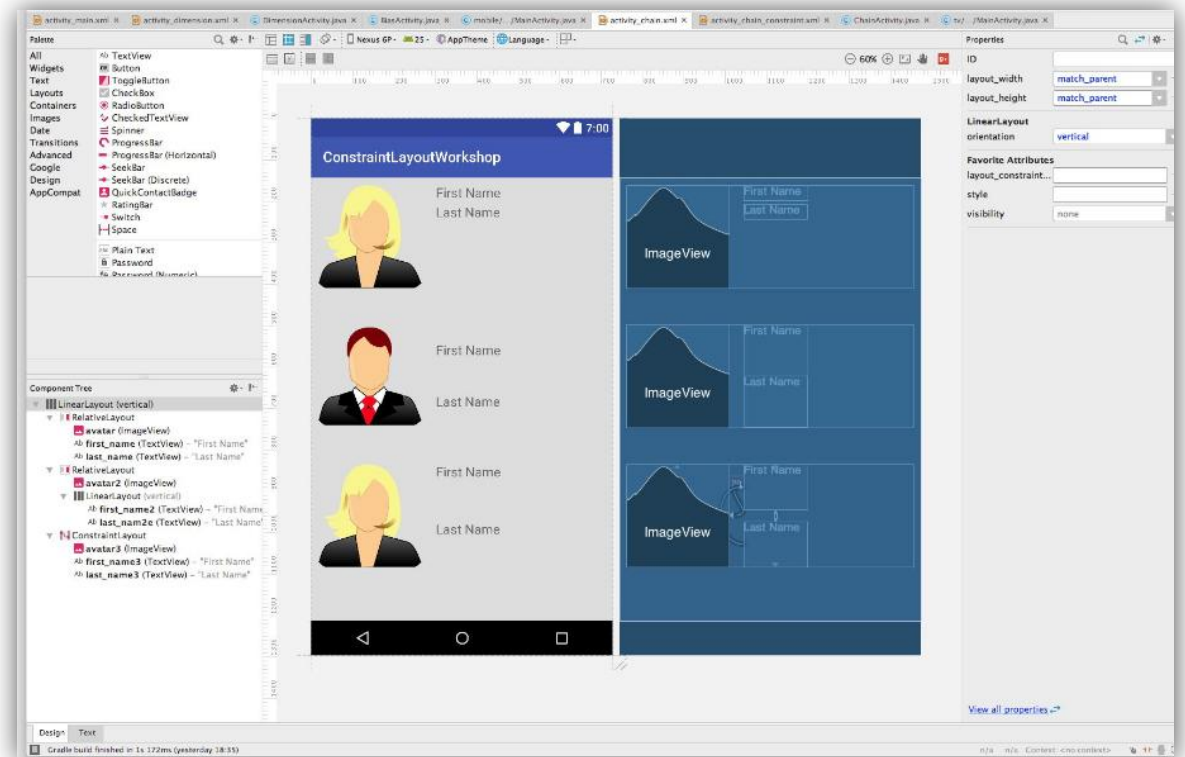
- Unbundled library
 - **Distributed like other libraries**
 - Compatible with **API9 and up**
- Reduce nesting
 - Traditional layouts are simple and require nesting
 - **Nesting is bad for performance**
 - Flatter hierarchy is better
- **Google wasn't first**
 - <https://github.com/anandsainath/constraint-layout> (Nov 2013)
 - <https://github.com/alexbirkett/android-cassowary-layout> (2014)



Source: Aleksander Piotrowski, ConstraintLayout all the things!

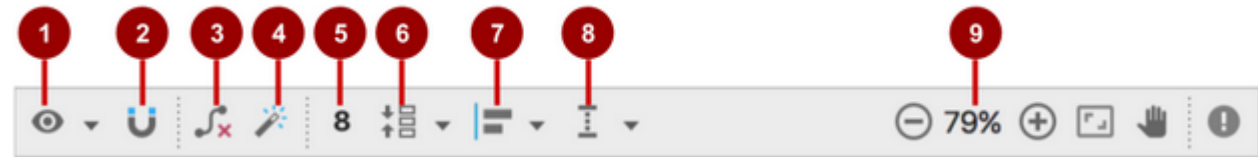
ConstraintLayout

- **Constraint**
 - Created **manually** in the XML or Java code
- **Autoinfer**
 - works most of time
 - don't break already existing constraints
 - don't move widgets around
- **Autoconnect**
 - per widget
- **Fully Constraint**
 - Constraints from both sides
 - Centered by default



ConstraintLayout

- **Constraint**
 - Created **manually** in the XML or Java code
 - **Autoinfer**
 - works most of time
 - don't break already existing constraints
 - don't move widgets around
 - **Autoconnect**
 - per widget
- **Fully Constraint**
 - Constraints from both sides
 - Centered by default



1. **Show:** Select **Show Constraints** and **Show Margins** to show them in the preview, or to stop showing them.
2. **Autoconnect:** Enable or disable Autoconnect. With Autoconnect enabled, you can drag any element (such as a Button) to any part of a layout to generate constraints against the parent layout.
3. **Clear All Constraints:** Clear all constraints in the entire layout.
4. **Infer Constraints:** Create constraints by inference.
5. **Default Margins:** Set the default margins.
6. **Pack:** Pack or expand the selected elements.
7. **Align:** Align the selected elements.
8. **Guidelines:** Add vertical or horizontal guidelines.

ConstraintLayout

- **Constraint**
 - A **connection** between *View* and another *View* or to *View* and *parent*



```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="50dp"
    android:layout_marginTop="50dp"
    android:text="First Name"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    android:textSize="40sp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="0dp"
    android:layout_marginTop="50dp"
    android:text="LastName"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    android:textSize="40sp"
    app:layout_constraintLeft_toLeftOf="@+id/textView2"
    app:layout_constraintTop_toBottomOf="@+id/textView2" />
```

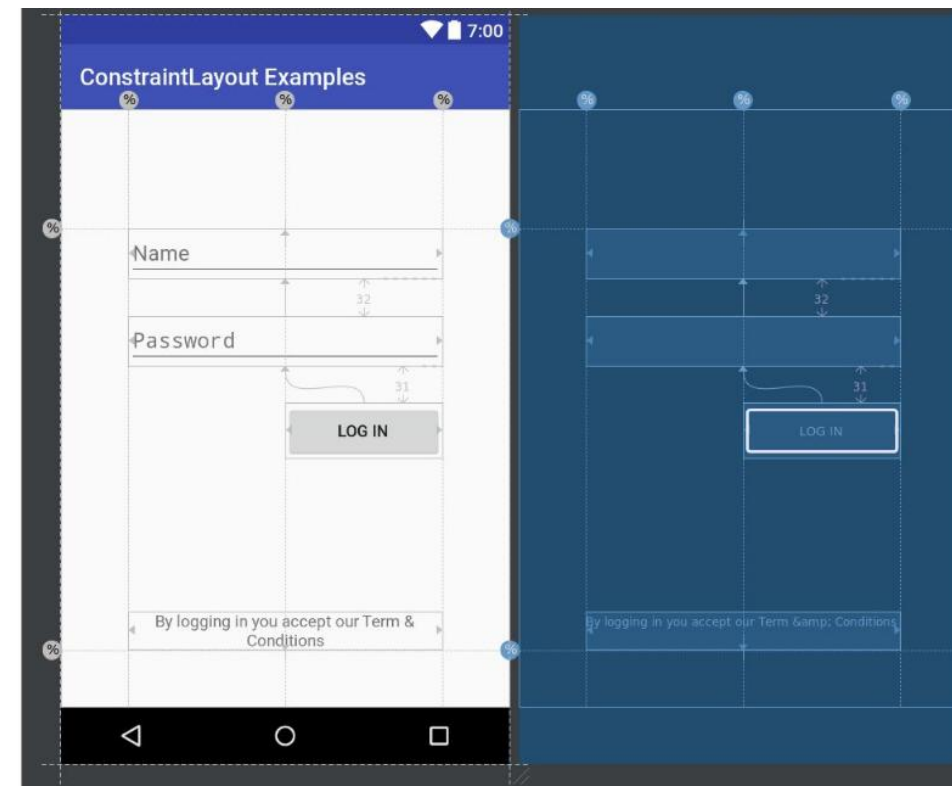
ConstraintLayout

- Guidelines**

- new View, used to generate new equations
- have a relation to parent
- can be in dp or %

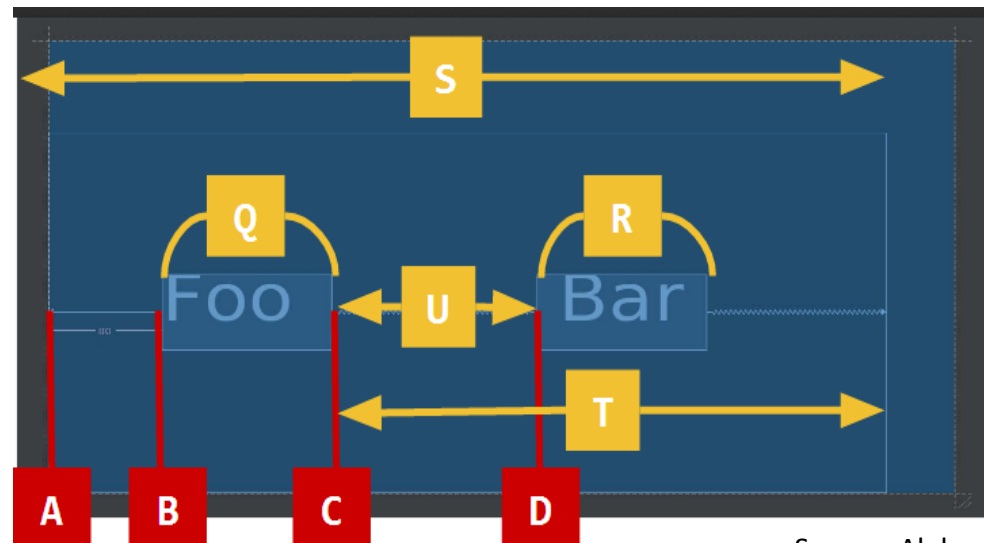
```
<android.support.constraint.ConstraintLayout
...>
```

```
<android.support.constraint.Guideline
    android:id="@+id/guideline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_percent="0.2" />
```



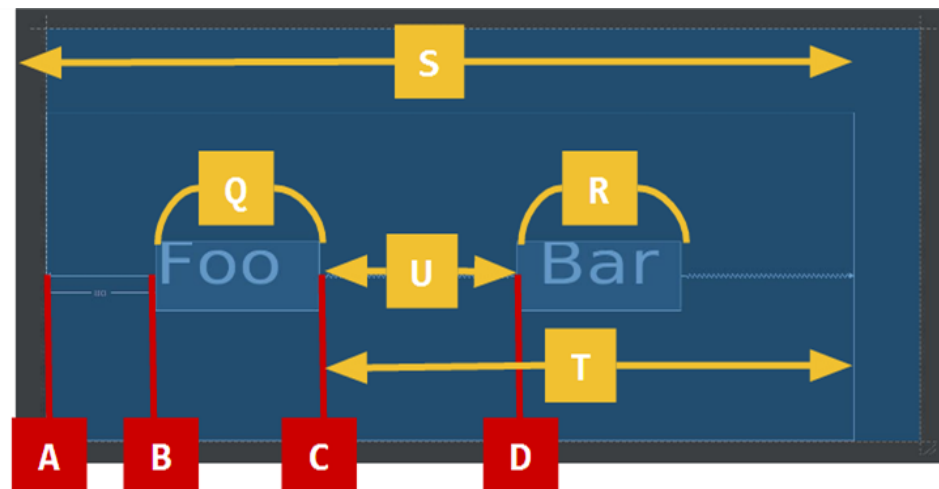
ConstraintLayout

- **Internals - Cassowary**
 - The Cassowary Linear Arithmetic Constraint Solving Algorithm
 - <https://constraints.cs.washington.edu/solvers/cassowary-tochi.pdf>
 - An incremental constraint solving toolkit that efficiently solves systems of linear equalities and inequalities



Source: Aleksander Piotrowski, ConstraintLayout all the things!

ConstraintLayout



$$A = 0$$

$$B = A + 80$$

$$C = B + Q$$

$$D = C + U$$

$$Q = 100$$

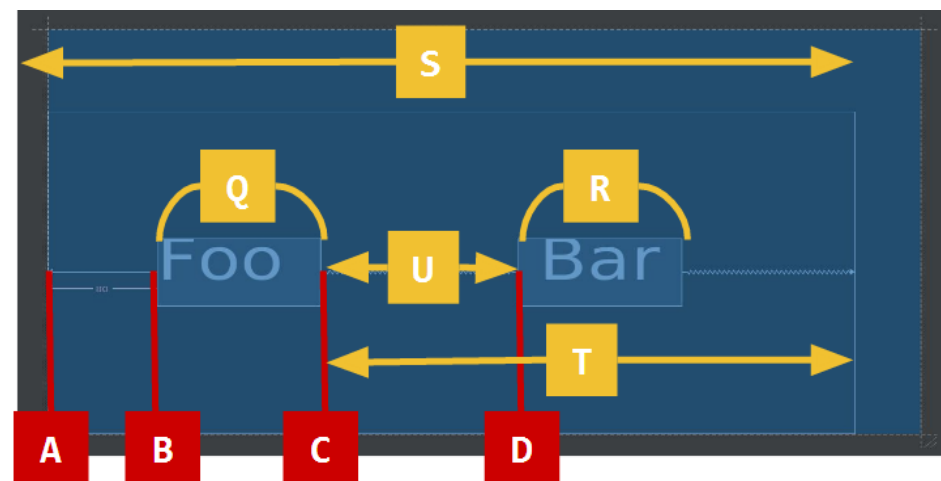
$$R = 100$$

$$S = 500$$

$$U = (T - R) / 2$$

$$T = S - C$$

ConstraintLayout



A = 0

B = 60

C = 160

D = 280

Q = 100

R = 100

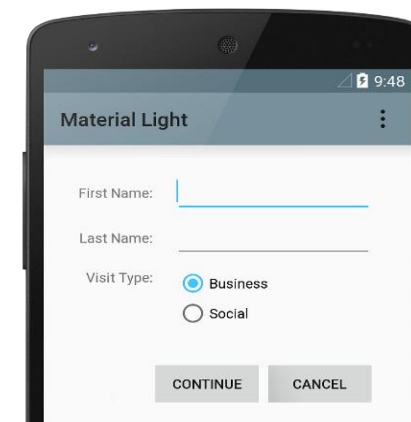
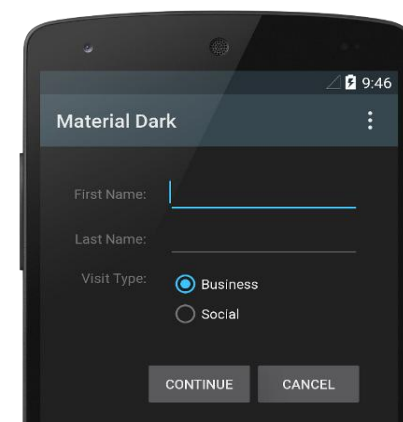
S = 500

U = 120

T = 340

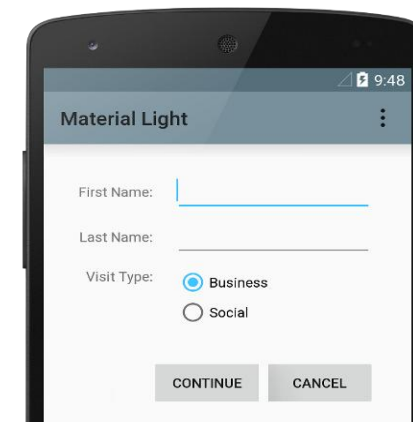
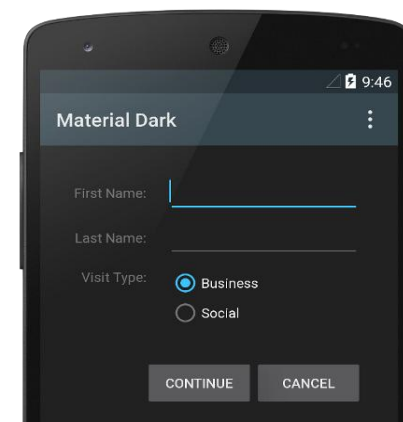
Material Design

- **Visual, motion, and interaction design across platforms and devices.**
 - Supported from Android 5.0 (API 21)
 - A new theme, widgets for complex views
 - New APIs for custom shadows and animations
 - Available on other platforms: Angular Material (angular), bootstrap-material-design (Bootstrap), Material Design In XAML Toolkit (C#) , Jfoenix (Java)
 - <https://design.google.com/#resources>
 - **Material 3** - <https://m3.material.io/>



Material Design

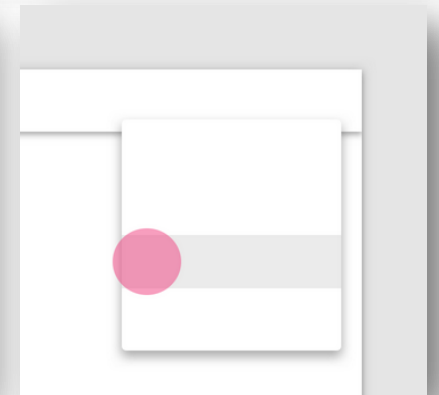
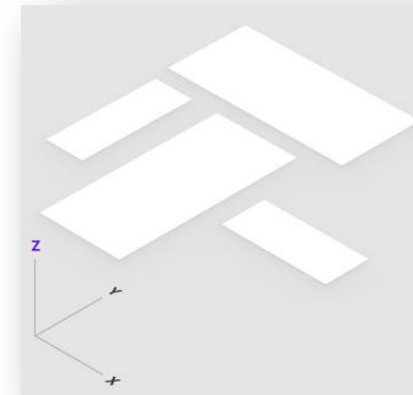
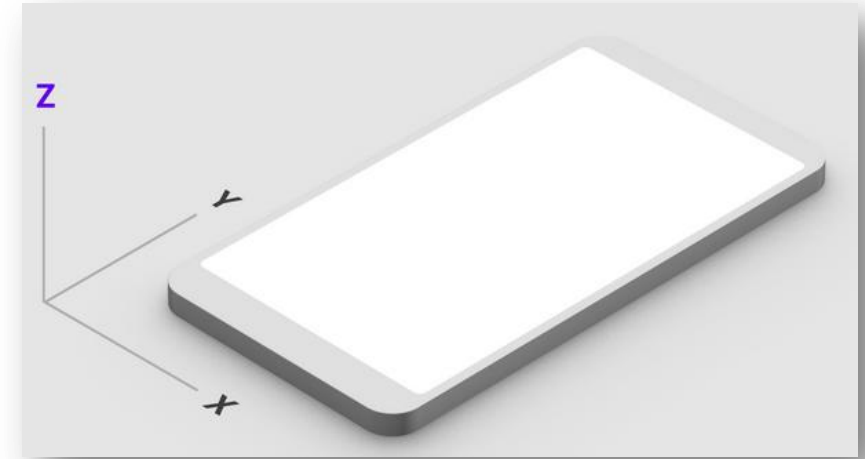
- **Unifies user experience** across platforms and device sizes
- **Set of guidelines** for style, layout, motion, and other aspects of app design
- „Deliberate color choices, edge-to-edge imagery, large-scale typography, and intentional white space that create a bold and graphic interface“



Material Environment

- Objects can be stacked or attached to one another, but **cannot pass through each other**
- They **cast shadows** and **reflect light**
- Material surfaces have consistent, unchangeable characteristics and behaviors
- Material is solid. User input and interaction cannot pass through material

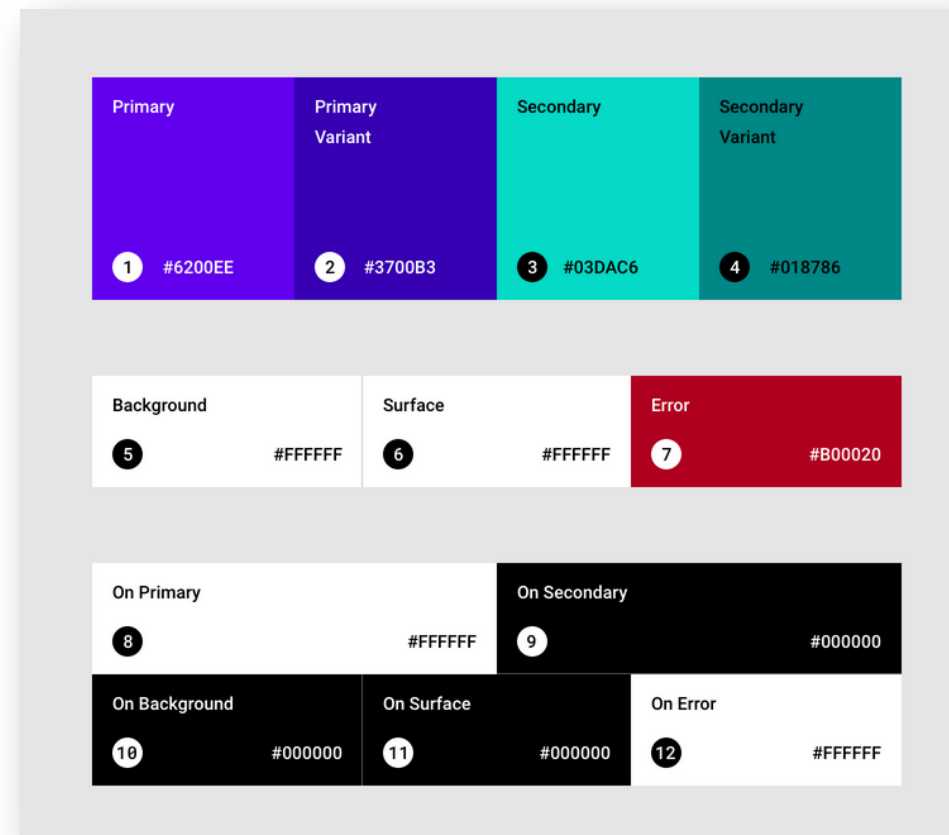
<https://material.io/design/environment/surfaces.html>



Material Design Color System

- **Primary and secondary colors**, their light and dark variants create harmonious color system.
- Material palette generator can be used to generate a palette.

<https://material.io/design/color/the-color-system.html>



The Type System

- Android supplies a type family named **Roboto**, created specifically for the requirements of UI and high-resolution screens.
- It is provide a font as a resource in XML that is bundled with the APK

<https://material.io/design/typography/the-type-system.html>

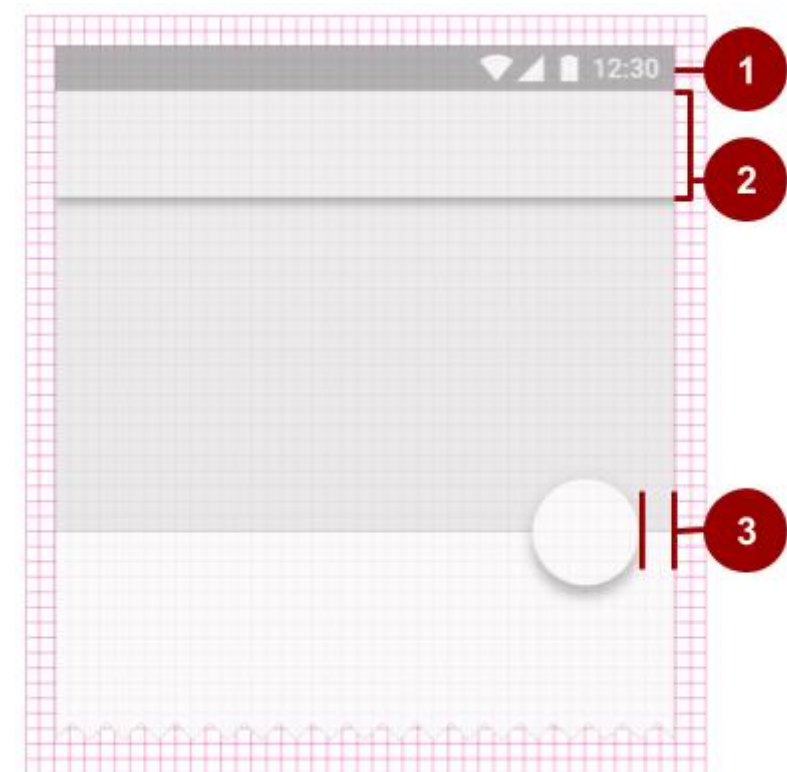
Roboto Thin
Roboto Light
Roboto Regular
Roboto Medium
Roboto Bold
Roboto Black
Roboto Thin Italic
Roboto Light Italic
Roboto Italic
Roboto Medium Italic
Roboto Bold Italic
Roboto Black Italic

Display 4	Light 112sp
Display 3	Regular 56sp
Display 2	Regular 45sp
Display 1	Regular 34sp
Headline	Regular 24sp
Title	Medium 20sp
Subheading	Regular 16sp (Device), Regular 15sp (Desktop)
Body 2	Medium 14sp (Device), Medium 13sp (Desktop)
Body 1	Regular 14sp (Device), Regular 13sp (Desktop)
Caption	Regular 12sp
Button	MEDIUM (ALL CAPS) 14sp

Layout

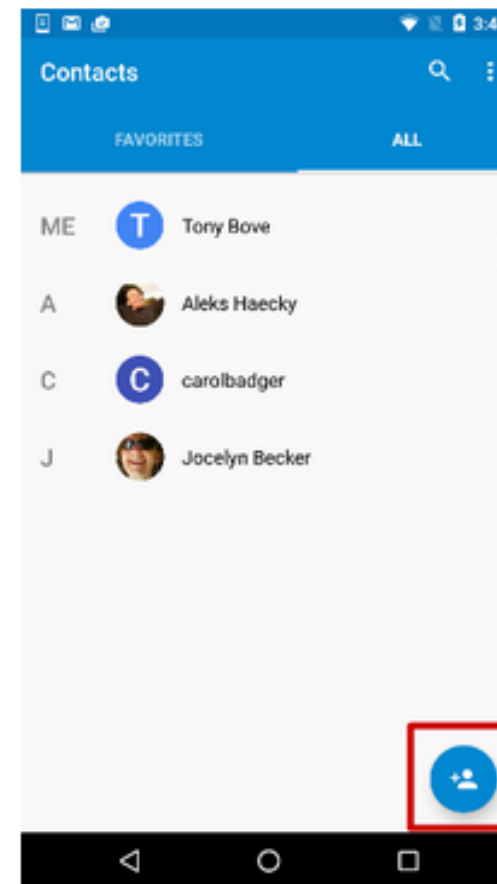
- Components are align to an **8dp square grid**
- **sp** is similar to an **dp**, but **sp** is also scaled by the user's font size preference
- **sp** is preferred for accessibility

1. The status bar in this layout is 24dp tall, the height of three grid squares.
2. The toolbar is 56dp tall, the height of seven grid squares.
3. One of the right-hand content margins is 16dp from the edge of the screen, the width of two grid squares.



Floating Action Button

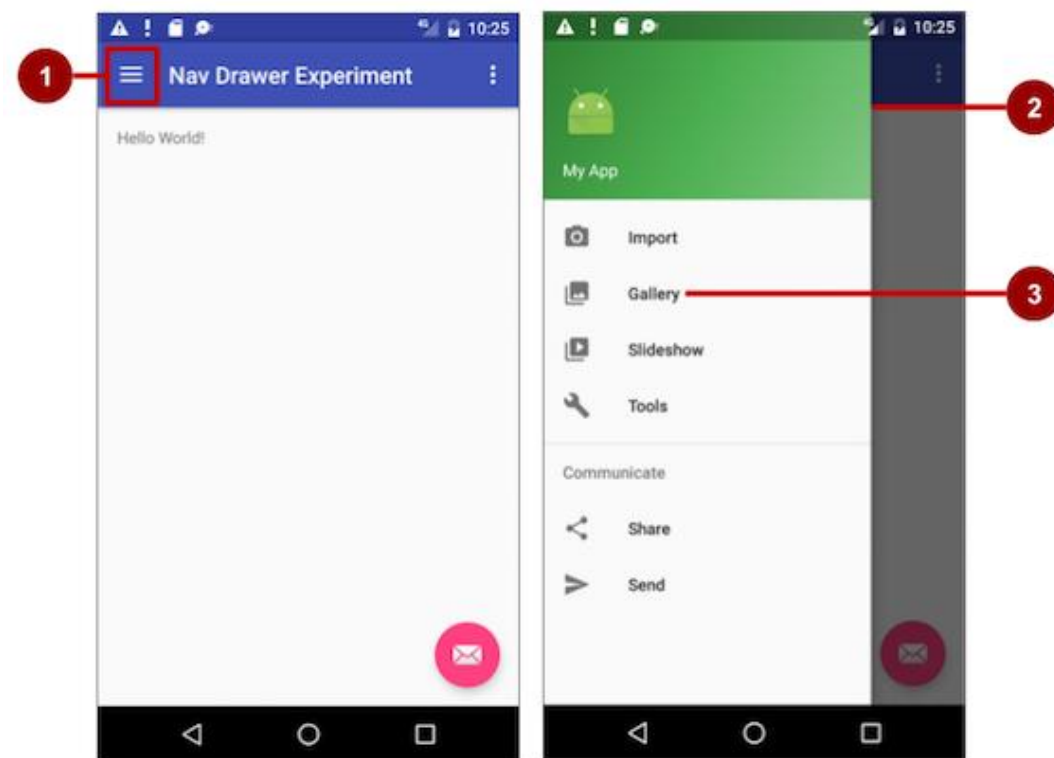
- Represent the **primary action** for a screen
- *FloatingActionButton* extends the *ImageButton* class
- By default, are 56 x 56 dp in size



Navigation drawer

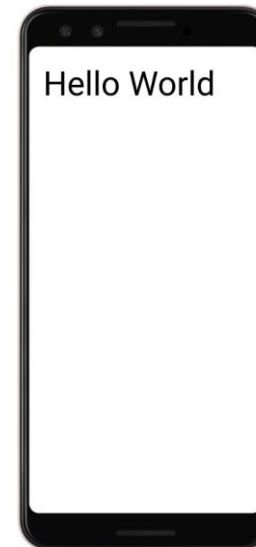
- Panel on the left edge of the screen, revealed when the user swipes a finger.
 1. Navigation icon in the app bar
 2. Navigation drawer
 3. Navigation drawer menu item
- *DrawerLayout* API is available in the Support Library.

<https://material.io/components/navigation-drawer/>



Composable

- Shifting back to a **declarative UI model**
- UI is defined by a set of ***composable*** functions that take in data and emit UI elements.
- **Basics**
 - **@Composable** annotation informs the Compose compiler that this function is intended to convert data into UI.
 - Composable functions can accept **parameters**, which allow the app logic to describe the UI.
 - **Text()** is composable function, which actually creates the text UI element.
 - To enable a preview of this composable create a new and annotate with **@Preview**

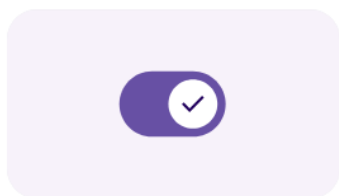


```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

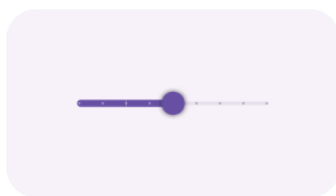
Composable

- Material Components

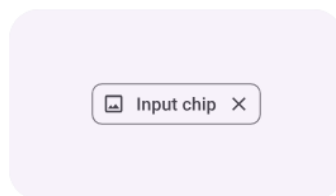
<https://developer.android.com/develop/ui/compose/components>



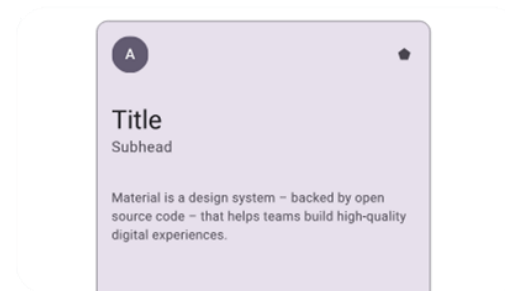
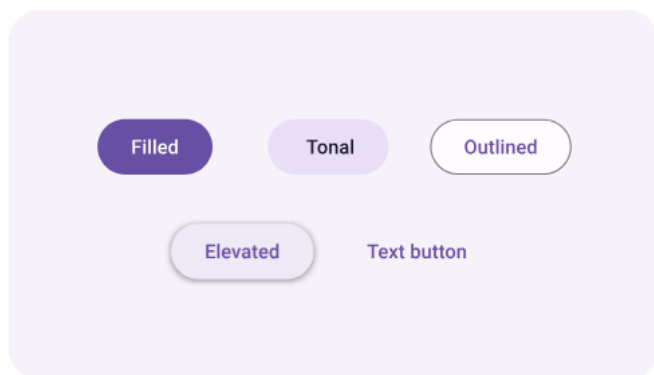
Switch



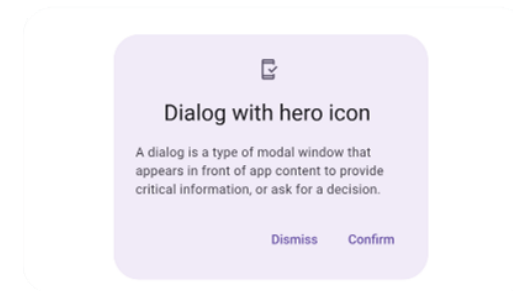
Slider



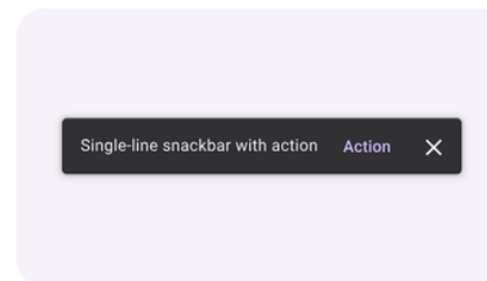
Chip



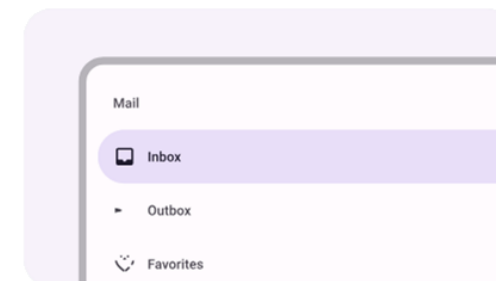
Card



Dialog



Snackbars



Drawers

Composable Layout Basics

- Standard Layout elements – **Column**, **Row**, **Box**



```
@Composable
fun ArtistCardColumn() {
    Column {
        Text("Alfred Sisley")
        Text("3 minutes ago")
    }
}
```

Alfred Sisley
3 minutes ago

```
@Composable
fun ArtistCardRow(artist: Artist) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(bitmap = artist.image, contentDescription = "Artist image")
        Column {
            Text(artist.name)
            Text(artist.lastSeenOnline)
        }
    }
}
```



Alfred Sisley
3 minutes ago



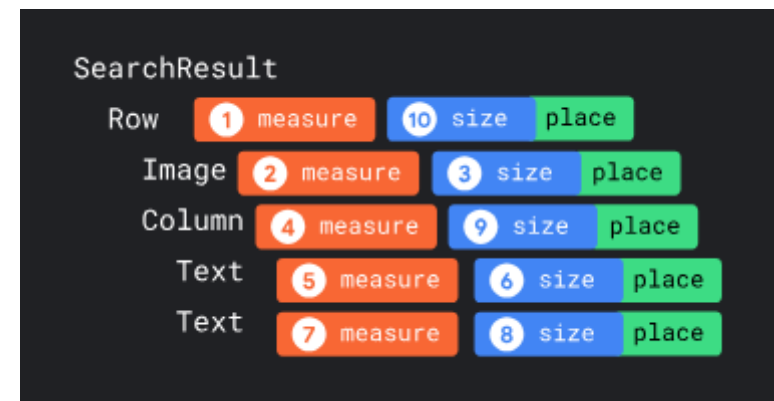
```
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(bitmap = artist.image, contentDescription = "Artist image")
        Icon(Icons.Filled.Check, contentDescription = "Check mark")
    }
}
```

Layout

Composable Layout Model

- **UI tree is laid out in a single pass.**
 - Compose achieves high performance by measuring children only once.
- Each node is first asked to measure itself, then measure any children recursively, passing size constraints down the tree to children.
- Then, leaf nodes are sized and placed, with the resolved sizes and placement instructions passed back up the tree.

```
@Composable
fun SearchResult() {
    Row {
        Image(
            // ...
        )
        Column {
            Text(
                // ...
            )
            Text(
                // ...
            )
        }
    }
}
```

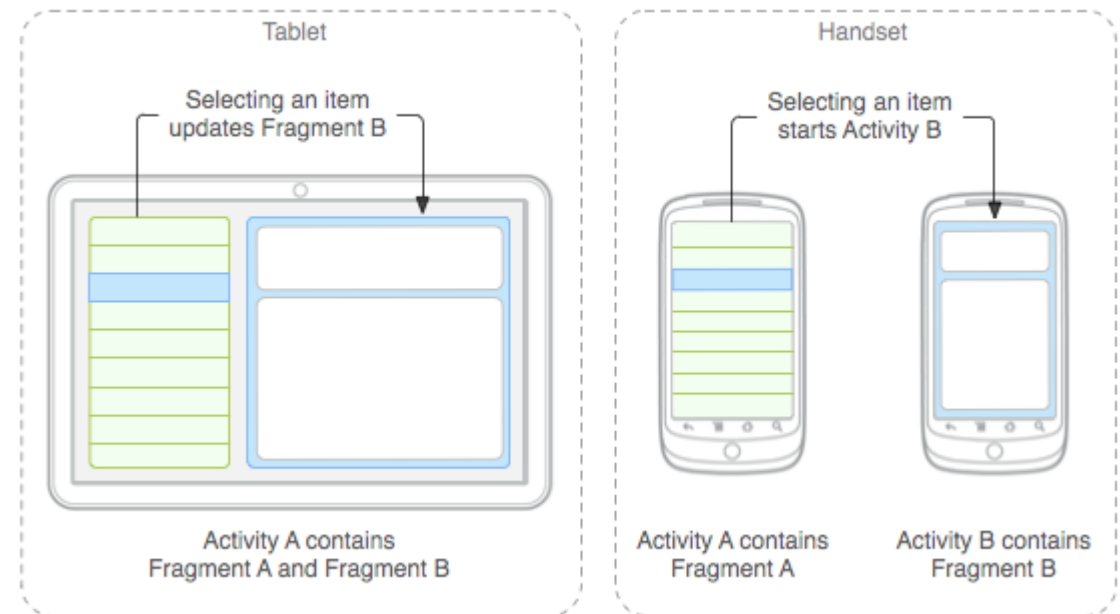


<https://developer.android.com/develop/ui/compose/layouts/basics>

<https://github.com/vinaygaba/Learn-Jetpack-Compose-By-Example>

Fragments

- Android introduced fragments in Android 3.0 (API11), primarily to support more dynamic and flexible UI designs on large screens.
- A Fragment represents a **behavior or a portion of user interface** in an Activity.
 - Fragment is a modular section of an activity, which has its own lifecycle, receives its own input events, and which can be added or removed while the activity is running.

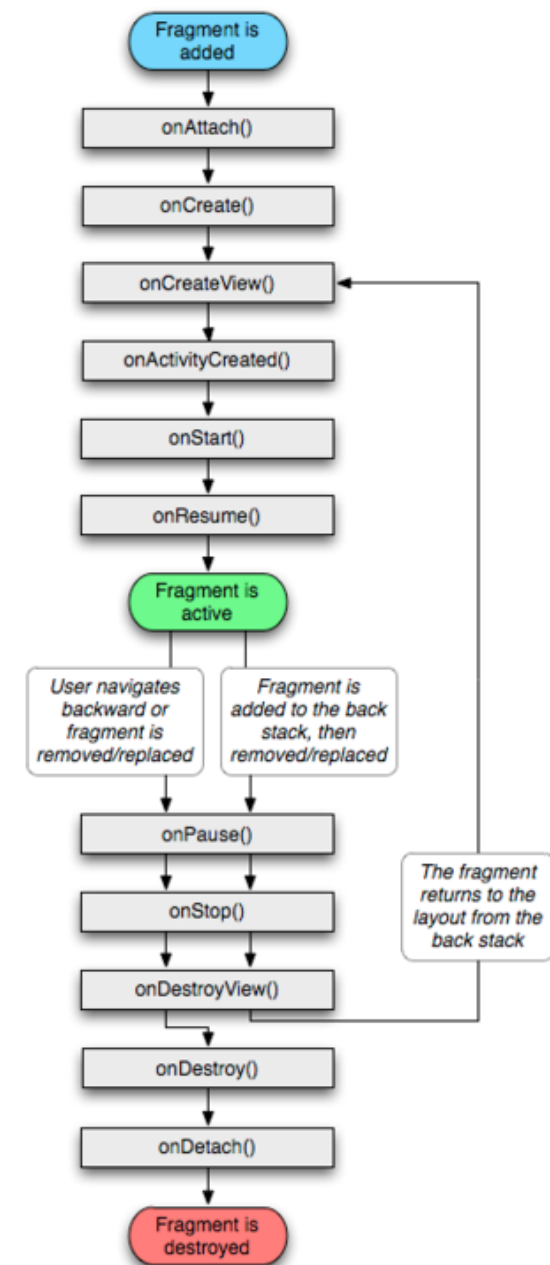


Fragments

Nowadays fragments require a dependency on the AndroidX Fragment library.

Fragment Lifecycle States

- **Resumed**
 - Fragment is visible in the running activity
- **Paused**
 - Another activity is in the foreground and has focus, containing activity is visible
- **Stopped**
 - The fragment is not visible



Fragment Lifecycle States

The Fragment class has code that **looks a lot like an Activity**.

- **onAttach()**
 - Fragment is first attached to its activity.
- **onCreate()**
 - The system calls this when creating the fragment. Initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView()**
 - The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- **onActivityCreated()**
 - Containing activity has completed onCreate() and the fragment has been installed.

Fragment Lifecycle

Activity destroyed:

- **onDestroyView()**
 - View previously created by `onCreateView()` has been detached from the Activity. Typical actions are to clean up view resources.
- **onDestroy()**
 - Fragment is no longer in use. Typical actions are to clean up fragment resources.
- **onDetach()**
 - Fragment no longer attached to its activity. Typical actions - null out references to hosting activity.

Adding Fragments

- Two general ways to add Fragments to an Activity's layout
- **Declare it statically** in the Activity's layout file
 - Layout can be inflated / implemented in `onCreateView()`
 - `onCreateView()` must return the View at the root of the Fragment's layout
 - This View is added to the containing Activity
- Added in code using the **FragmentManager**
 - Get reference to the `FragmentManager`
 - Begin a `FragmentTransaction`
 - Add the Fragment
 - Commit the `FragmentTransaction`

```
getSupportFragmentManager().beginTransaction()  
    .setReorderingAllowed(true)  
    .add(R.id.fragment_container_view, ExampleFragment.class, bundle)  
    .commit();
```

Adding Fragments

```
public class TabFragment1 extends Fragment {
```

```
    public TabFragment1() {
        // Required empty public constructor
    }
```

```
@Override
```

```
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    return inflater.inflate(R.layout.fragment_tab_fragment1, container, false);
}
```

```
}
```

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="pl.edu.agh.wyklad3.TabFragment1">
```

```
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:textSize="30sp"
        android:text="Hello 1" />
```

```
</FrameLayout>
```

Adding Fragments

```

public class MyPagerAdapter extends FragmentPagerAdapter {
    final int PAGE_COUNT = 3;

    private String tabTitles[] = new String[] { "Tab1", "Tab2", "Tab3" };
    private Context context;

    public MyPagerAdapter(FragmentManager fm, Context context) {
        super(fm);
        this.context = context;
    }

    @Override
    public int getCount() { return PAGE_COUNT; }

```

```

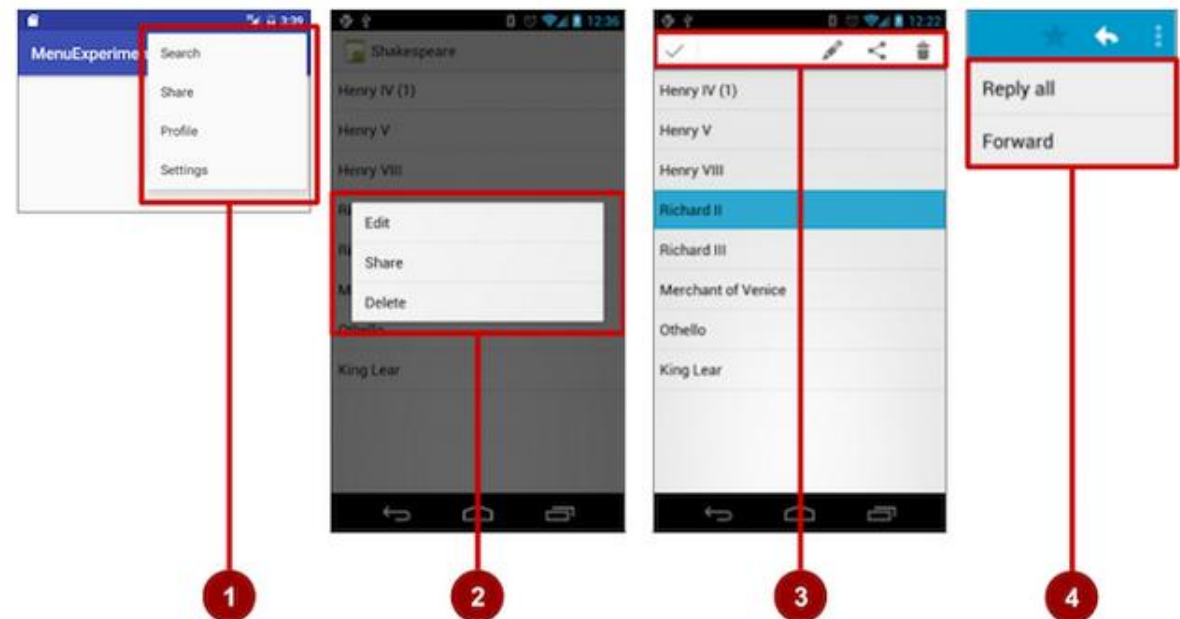
@Override
public Fragment getItem(int position) {
    switch (position) {
        case 0:
            TabFragment1 tab1 = new TabFragment1();
            return tab1;
        case 1:
            TabFragment2 tab2 = new TabFragment2();
            return tab2;
        case 2:
            TabFragment3 tab3 = new TabFragment3();
            return tab3;
        default:
            return null;
    }
}

@Override
public CharSequence getPageTitle(int position) { return tabTit
}

```


Menus

1. **Options menu:** Appears in the app bar and provides the primary options that affect use of the app itself.
2. **Contextual menu:** Appears as a floating list of choices when the user performs a long tap on an element on the screen.
3. **Contextual action bar:** Appears at the top of the screen overlaying the app bar, with action items that affect the selected element or elements.
4. **Popup menu:** Appears anchored to a View such as an ImageButton, and provides an overflow of actions or the second part of a two-part command.



Menus

- **Options Menu**
 - The primary collection of menu items for an activity, which appears when the user touches the MENU button. When your application is running on Android 3.0 or later, you can provide quick access to select menu items by placing them directly in the Action Bar, as "action items."
- **Context Menu**
 - A floating list of menu items that appears when the user touches and holds a view that's registered to provide a context menu.
- **Submenu**
 - A floating list of menu items that appears when the user touches a menu item that contains a nested menu.

Menus

- Menu and all its items are defined in an **XML menu resource** (res/menu/).
- **Item element**
 - Represents a single item in a menu. This element may contain a nested **<menu>** element in order to create a submenu.
- **Group element**
 - Optional, invisible container for **<item>** elements. It allows you to categorize menu items so they share properties such as active state and visibility.
- **Submenus**
 - **<menu>** element as the child of an **<item>**
- **Menu groups**
 - Show or hide all items, enable or disable all items
- **Checkable menu items**
- **Shortcut keys**
 - Using hardware keyboard
- **Dynamically adding menu intents**
 - **Menu.addIntentOptions()** method

References

- <https://developer.android.com/develop/ui/compose/documentation>
- <https://developer.android.com/develop/ui/compose/layouts>
- <https://google-developer-training.github.io/android-developer-advanced-course-concepts/>
- <https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-2-user-experience/lesson-4-user-interaction/4-1-c-buttons-and-clickable-images/4-1-c-buttons-and-clickable-images.html>
- **Aleksander Piotrowski, ConstraintLayout all the things!**
[https://assets.contentful.com/2grufn031spf/w8yoOn0pa0w0MOQ8yeAu6/1f703688ff94a64ee9ac06575d7506db/Piotrowski Aleksander ConstraintLayout all the things .pdf](https://assets.contentful.com/2grufn031spf/w8yoOn0pa0w0MOQ8yeAu6/1f703688ff94a64ee9ac06575d7506db/Piotrowski_Aleksander_ConstraintLayout_all_the_things.pdf)

Thank you for your attention

Mgr. Ing. **Michal Krumnikl**, Ph.D.

+420 597 325 867

michal.krumnikl@vsb.cz

www.vsb.cz