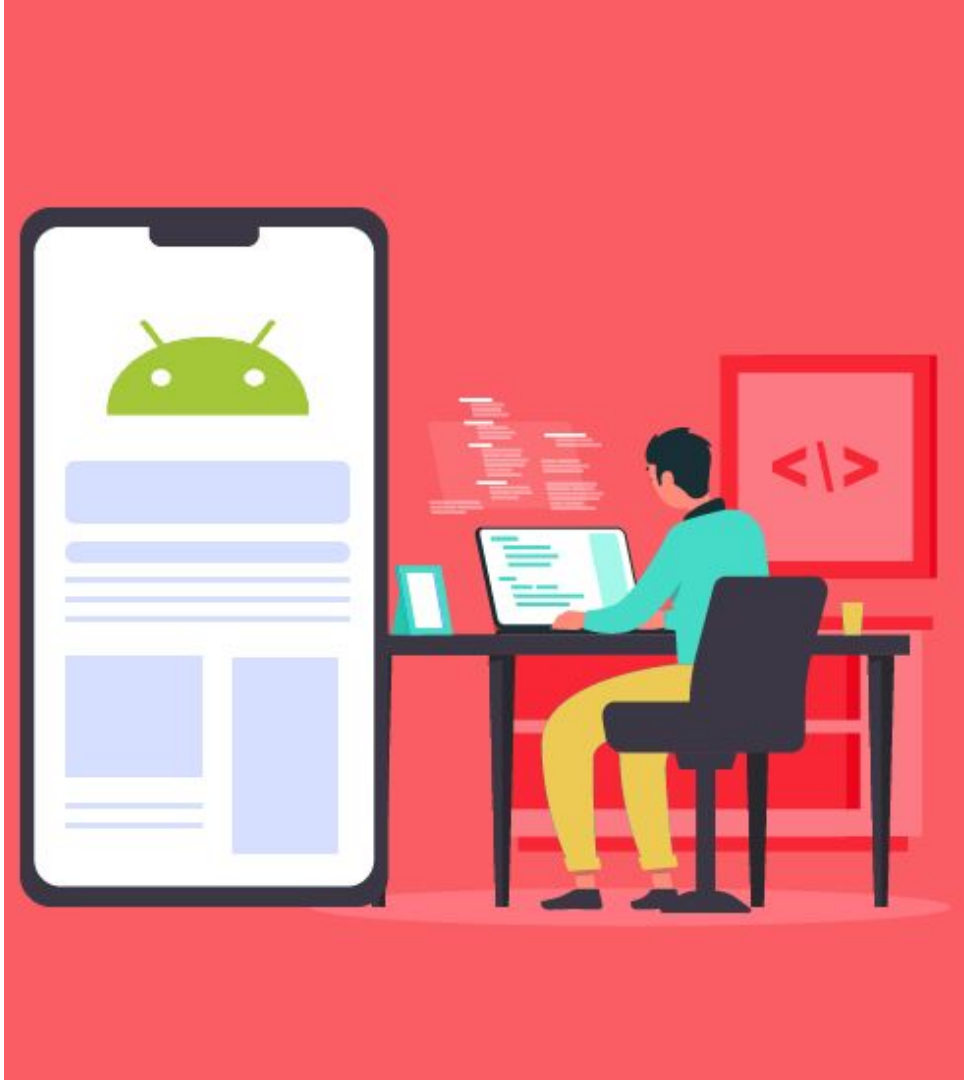
User Interface

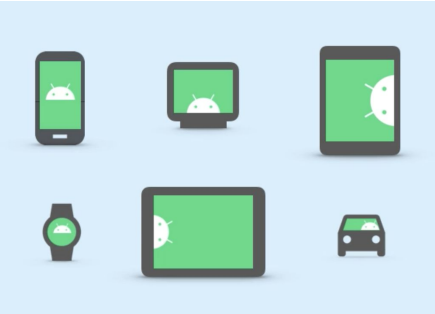
Overview

- UI Fundamentals
- Resources & IDs
- Constraint Layout
- RecyclerView



UI Fundamentals

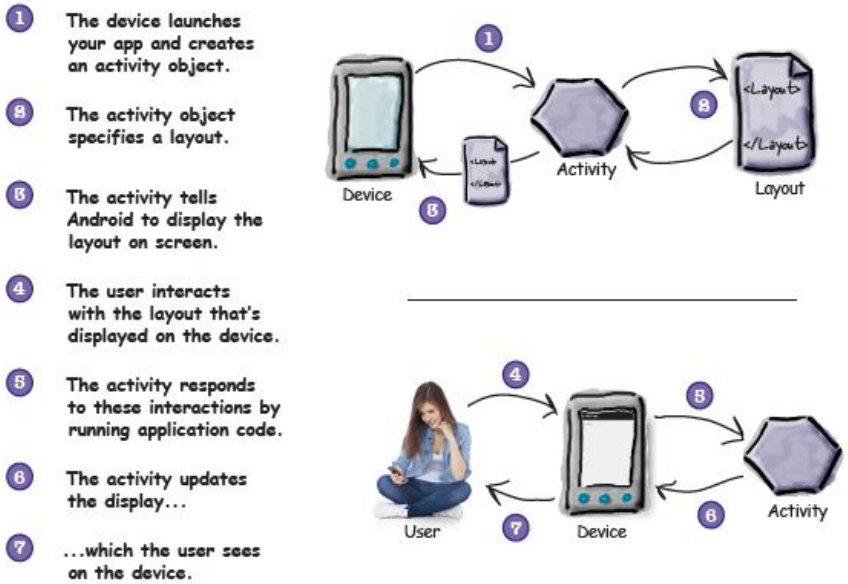
Activities and Layouts



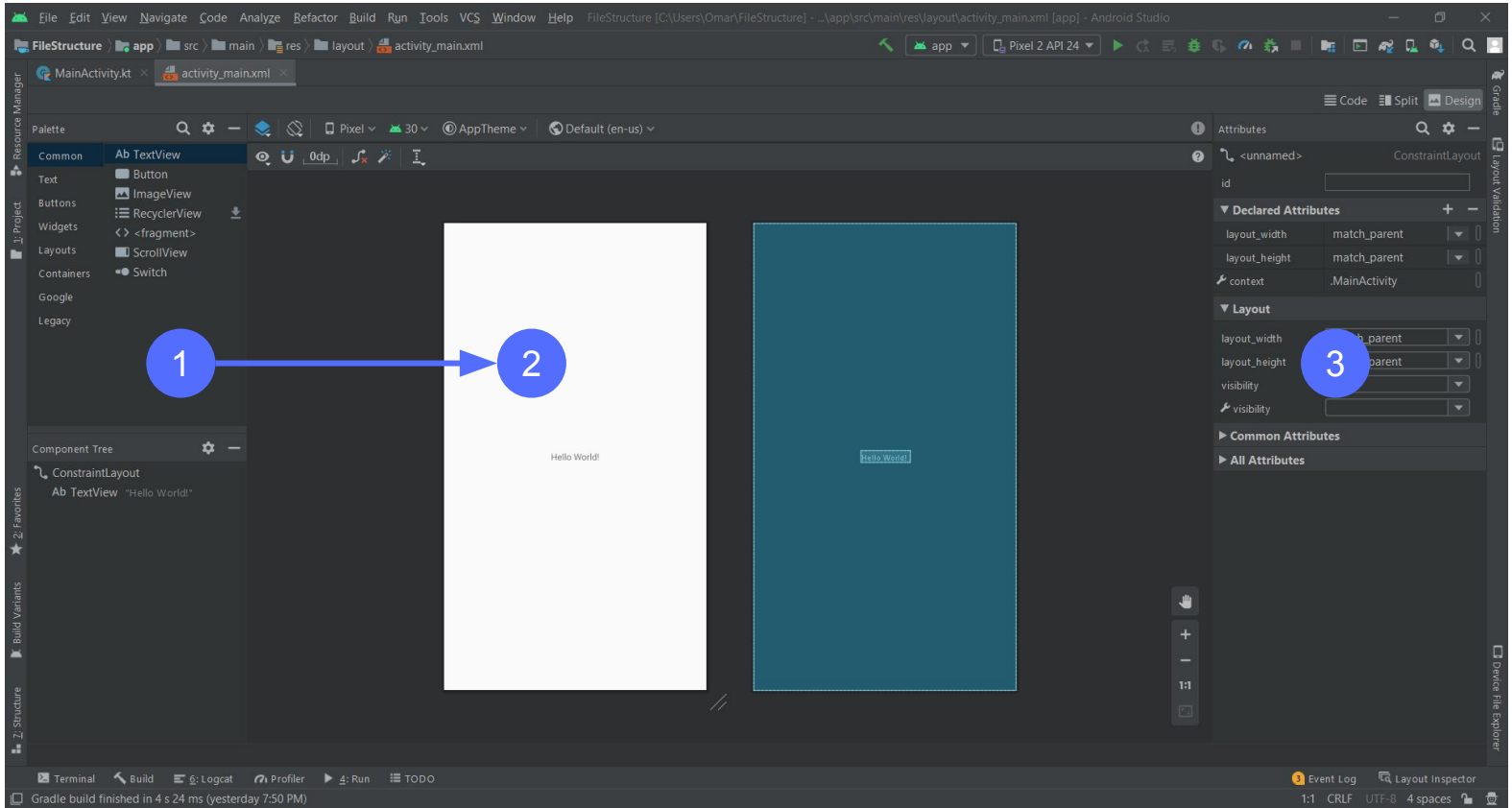
Every Android app is a collection of screens, and <u>each screen is comprised of an activity and a layout</u>.

Activities: Tell Android how the app should interact with the user

Layouts: Align views based on the rules of the layout manager.



Layout Editor



Layouts

- Layouts are xml files that define how an activity looks
  - Views and Viewgroups
- Connect app to layout files when each screen is created
- Simple UI is done in XML and advanced in Kotlin (ie: Games)

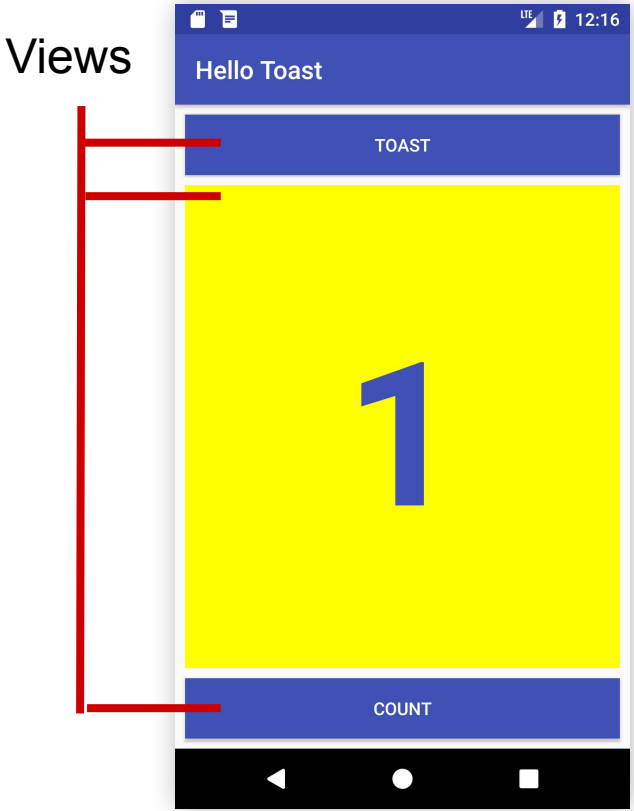
MainActivity.kt | activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:andr
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

XML

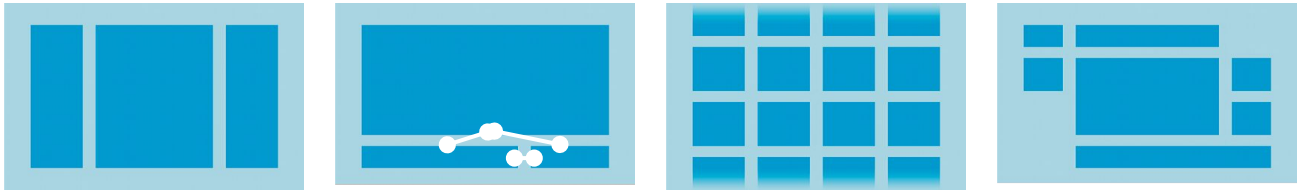
Views

- Every UI element is a View
- Basic building block for all UI components
  - Base class for interactive UI components
- Placed in layout resource files (xml)
- Predefined View Subclasses:
  - TextView display text
  - EditText enables the user to enter and edit text
  - Clickable elements provide interactive behavior
    - Ex: Button, RadioButton, CheckBox, Spinner
  - And more!
- Sometimes called a widget (don't get confused)
- Usually physically located under the app bar



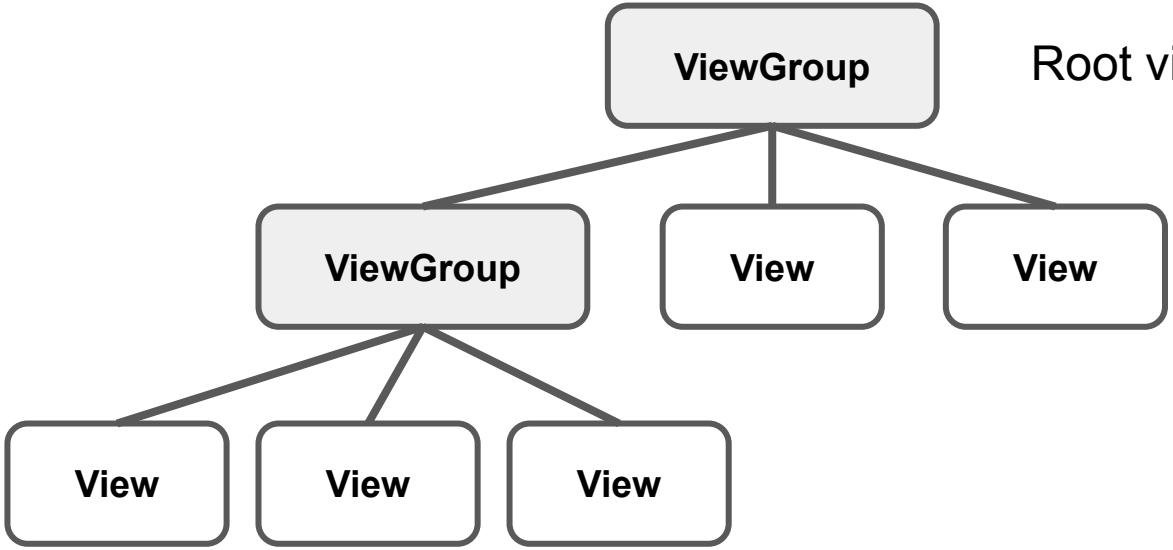
ViewGroup

- Special type of View that is used as a container to hold other Views and ViewGroups
- Commonly used ViewGroups
  - ConstraintLayout: a container that connects children Views using constraints
  - ScrollView: has one View child and enables scrolling on it
  - RecyclerView: Scrollable container for displaying views in a list
- Contain child views and can be in a row, column, grid, table, or absolute



`LinearLayout` `ConstraintLayout` `GridLayout` `TableLayout`

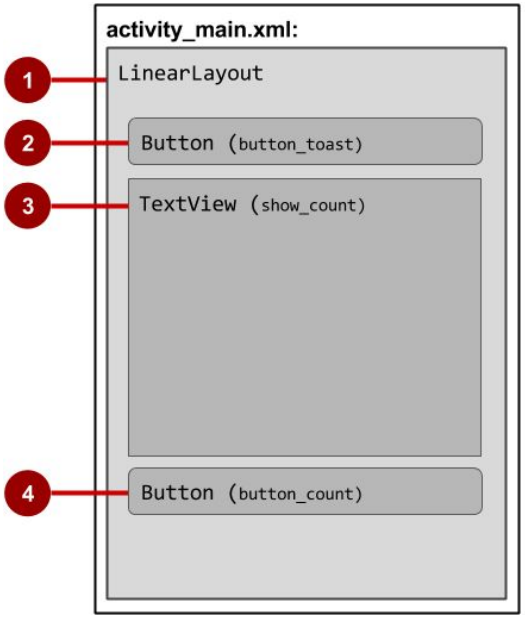
Views and ViewGroups Hierarchy

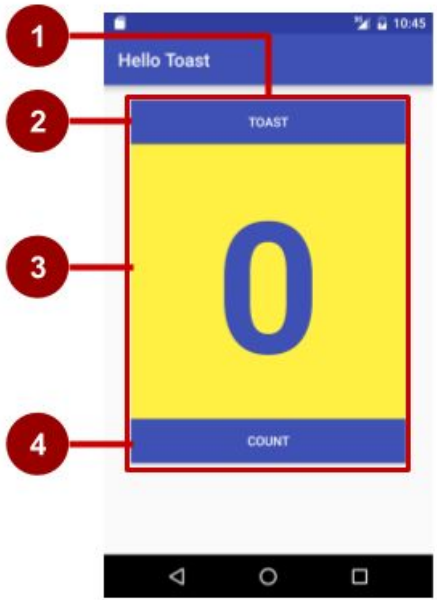


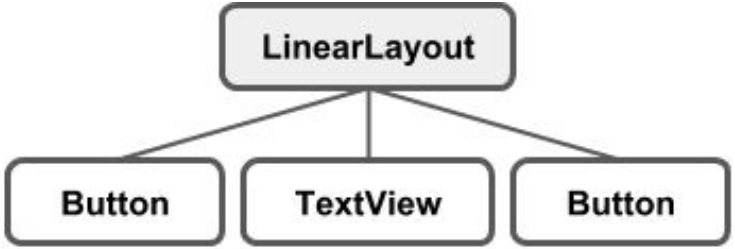
Root view is always a `ViewGroup`

View Hierarchy Example

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
    <Button
       ... />
    <TextView
       ... />
    <Button
       ... />
</LinearLayout>
```







View and ViewGroup xml Structures

ViewGroup xml

```
<ViewGroupType
      android:id="@+id/viewgroup_id"
      android:layout_height="match_parent"
      android:layout_width="match_parent"
      ...
      android:attributeName="attributeValue" >

      <!-- Place other Views
           and Viewgroups in here -->

</ViewGroupType>
```

View xml

```
<ViewType
      android:id="@+id/view_id"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:text="@string/view_text"
      ...
      android:attributeName="attributeValue" />
```

Layout Example

```
<LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="vertical">

      <Button ... />
      <Button ... />
      <Button ... />
      <Button ... />

</LinearLayout>
```

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button 1" />
```

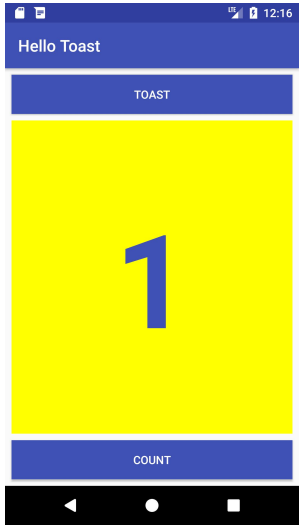
Example: Comparing ViewGroups

LinearLayout

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
    <Button
        ... />
    <TextView
        ... />
    <Button
        ... />
</LinearLayout>
```

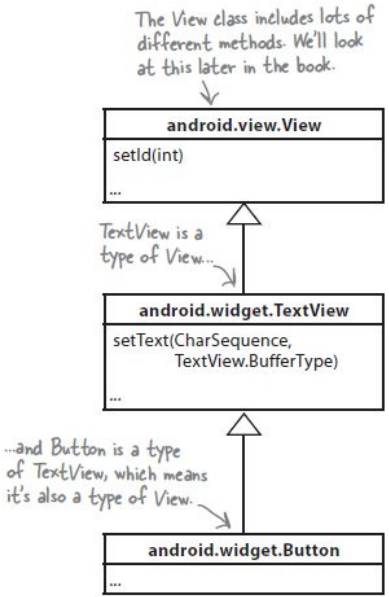
ConstraintLayout

```
<ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent">
    <Button
        ... />
    <Button
        ... />
    <TextView
        ... />
</ConstraintLayout>
```



Attributes

- Attributes give you control over Views in a layout
- Must be specified for all views
  - `android:layout_width`
  - `android:layout_height`



Sizing

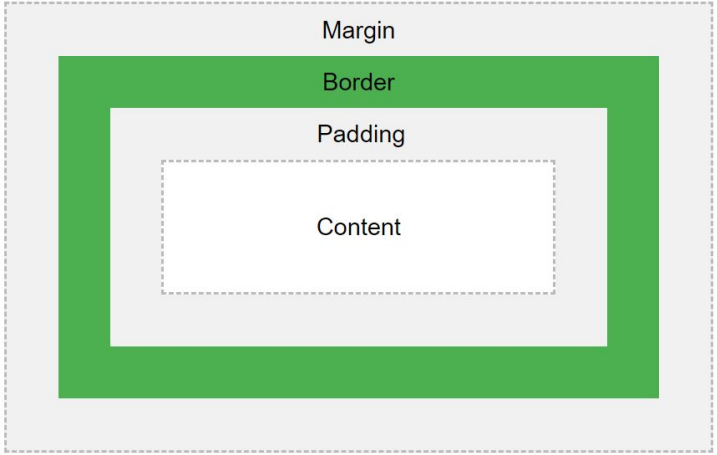
Common values you may see:

- match_parent - make the layout as big as its parent (minus any padding)
- wrap-content - make layout big enough to hold all of the views inside it'
- Specific values such as 10dp (density-independent pixels)

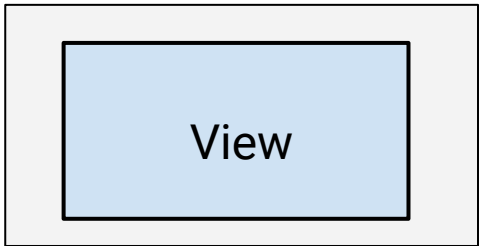
```
<TextView
    android:id="@+id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Text 1" />
```

View Box Model

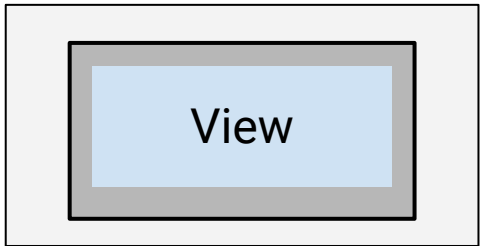
- **Content** - size of the view itself
- **Padding** - artificial increase to view size outside of content
- **Border** - outside padding, a line around edge of view
- **Margin** - invisible separation from neighboring views



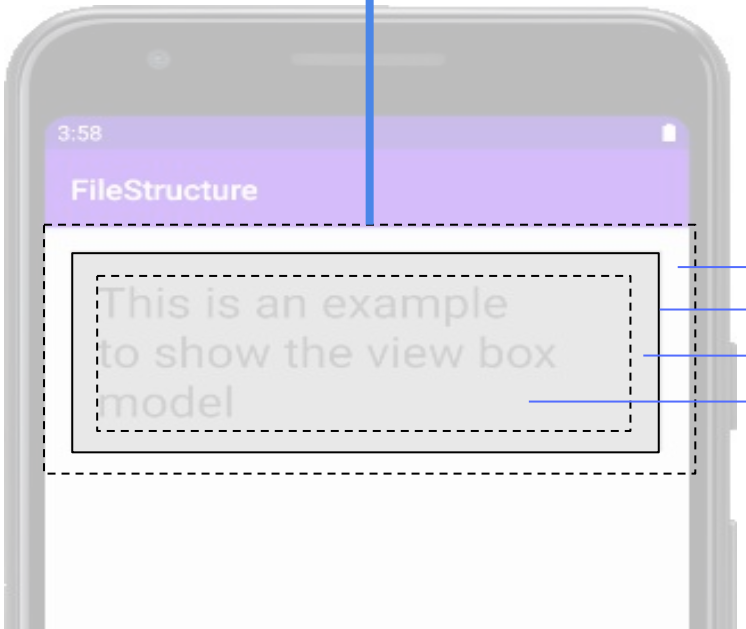
View with margin



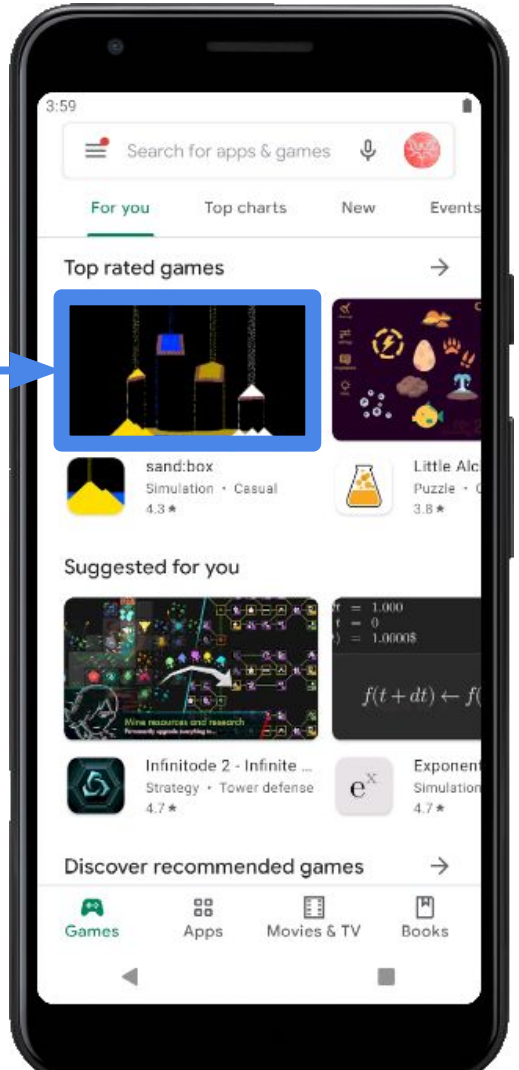
View with margin and padding



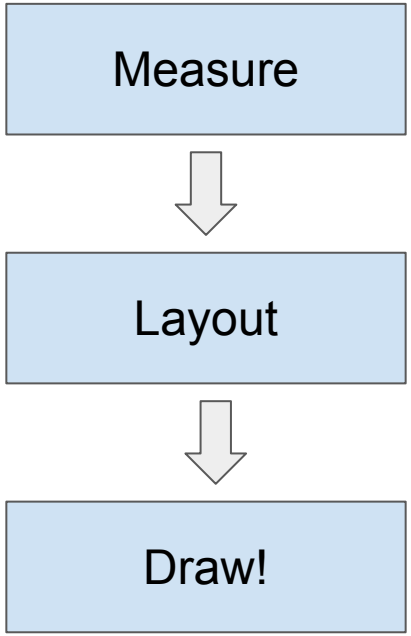
Box Model Example



Margin
Border
Padding
Content

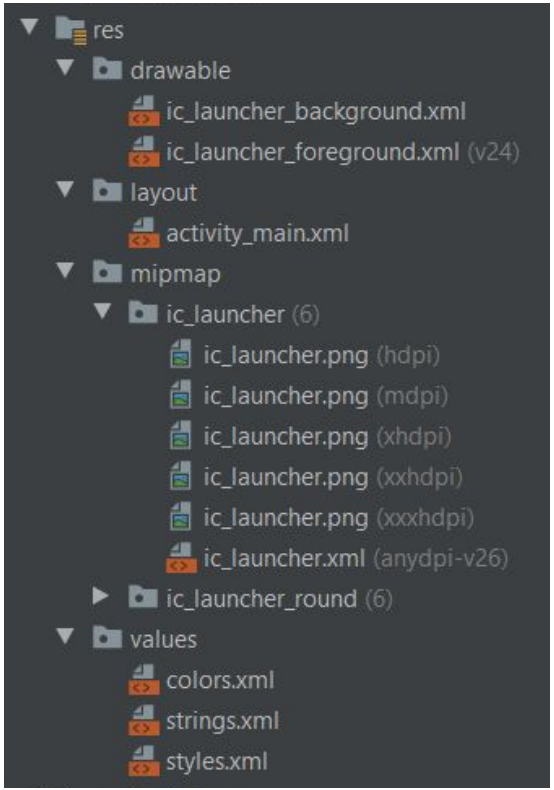


View Rendering Cycle



Resources & IDs

Resources Overview



What are Resources

- Anything from colors, images, layouts, menus, and string values
- Everything defined in resource files can be referenced within your application's code (flexible code)
- Resource files are stored in **\res**
- Useful for localization

Common Types of Resources

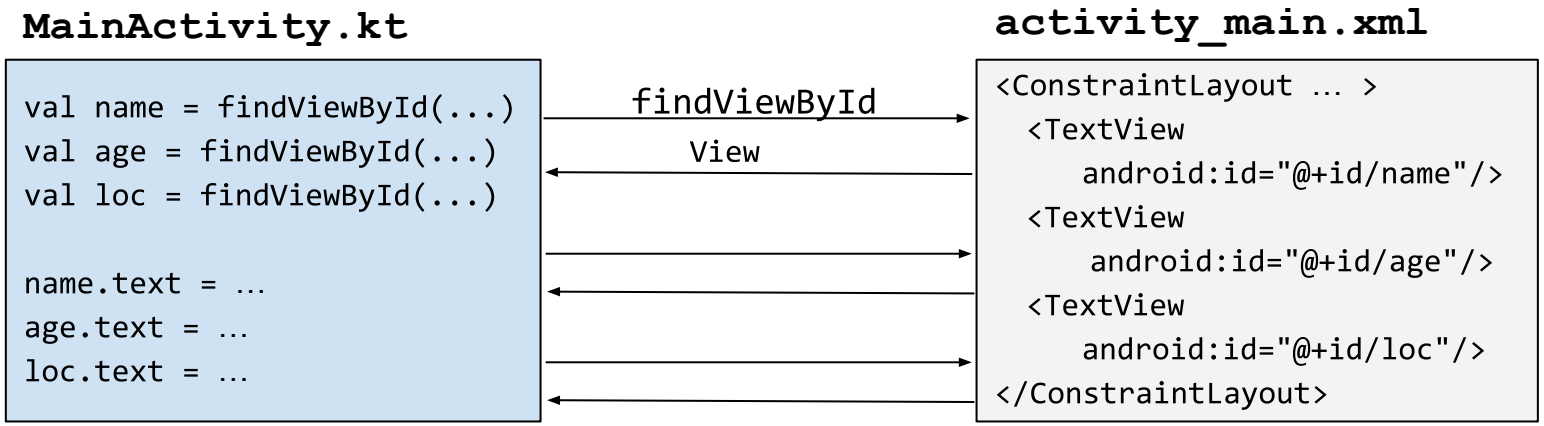
- **drawable:** images and icons
- **layout:** layout resource files for UI
- **mipmap:** pre-calculated, optimized collections of app icons used by the Launcher
- **values:** colors, dimensions, strings, and styles (themes)
- **menu:** menu items

Importance of View IDs

```
val resultTextView: TextView = findViewById(R.id.myText)
resultTextView.text = “this is the updated text”
```

```
<TextView
    android:id="@+id/myText" ... />
```

- View IDs give you the ability to reference views in the view hierarchy
- Can grab and update data/attributes of a view after connecting



What is R.java

- A special class that enables you to retrieve references to app resources
- Created when you build the app

Purpose

- @+id tells android to include the id of your view as a resource in the resource file R.java
- This makes it possible to access views inside an activity

```
<TextView
    android:id="@+id/myText"
    android:text="What is R.java?"
    ... />
```

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY. */
package com.example.helloandroid;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int menu_settings=0x7f070000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
    }
    ...
}
```

Using R.java

R.id can be used to connect to views by their id, the id's are stored in R.java

- Can access views using R.id.<resource_name>

R.string can be used to get text

R.layout can be used to get the layouts

View Resources

```
tv = findViewById<TextView>(R.id.textView);
```

String Resources

```
R.string.title
```

MainActivity.kt

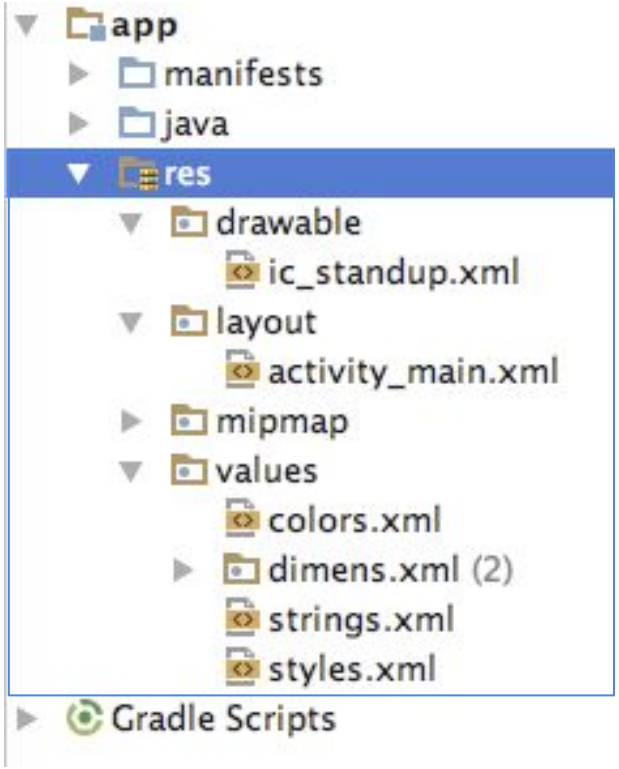
```
android:text="@string/title"
```

activity_main.xml

Layout Resources

```
setContentView(R.layout.activity_main);
```

Resources Overview



resources and resource files stored in **res** folder

- Layout:

  `R.layout.activity_main`

  `setContentView(R.layout.activity_main);`

- View:

  `R.id.textView`

  `tv = findViewById<TextView>(R.id.textView);`

- String:

  In Kotlin: `R.string.title`
  In XML: `android:text="@string/title"`

Data Binding

Data Binding can bind UI components in layouts to data sources in app

- Helps reduce potential for crashes & reduces amount of code
- An alternative to findViewById()
- Find incorrect layout ID associations at compile time

MainActivity.kt

```
Val binding:ActivityMainBinding

binding.name.text = …
binding.age.text = …
binding.loc.text = …
```

initialize binding

activity_main.xml

```
<layout>
   <ConstraintLayout … >
       <TextView
          android:id="@+id/name"/>
       <TextView
          android:id="@+id/age"/>
       <TextView
          android:id="@+id/loc"/>
   </ConstraintLayout>
</layout>
```

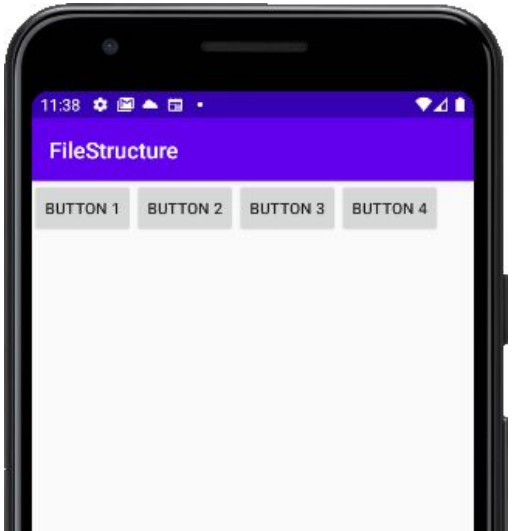
ConstraintLayout

Another Look at LinearLayout

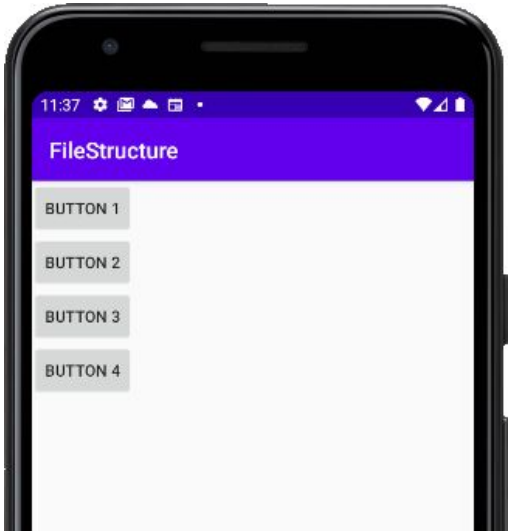
```
<LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="vertical">

      <Button ... />
      <Button ... />
      <Button ... />
      <Button ... />

</LinearLayout>
```

- Only Displays Vertical or Horizontal
- Useful but not very versatile

```
android:orientation="horizontal">
```



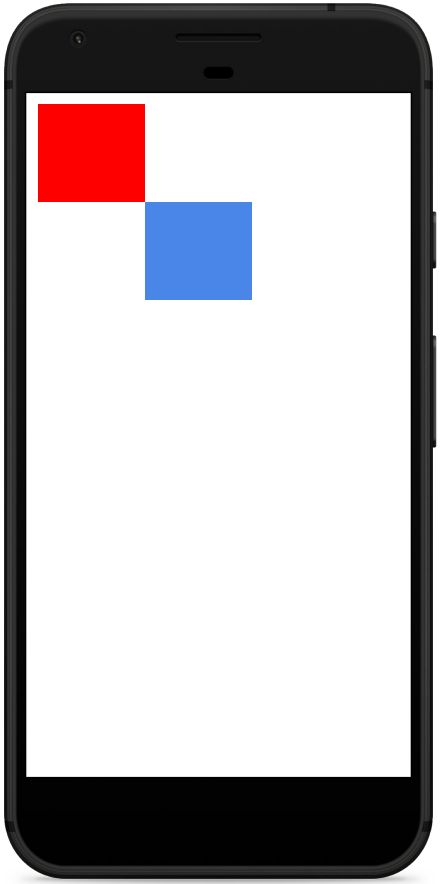

```
android:orientation="vertical">
```

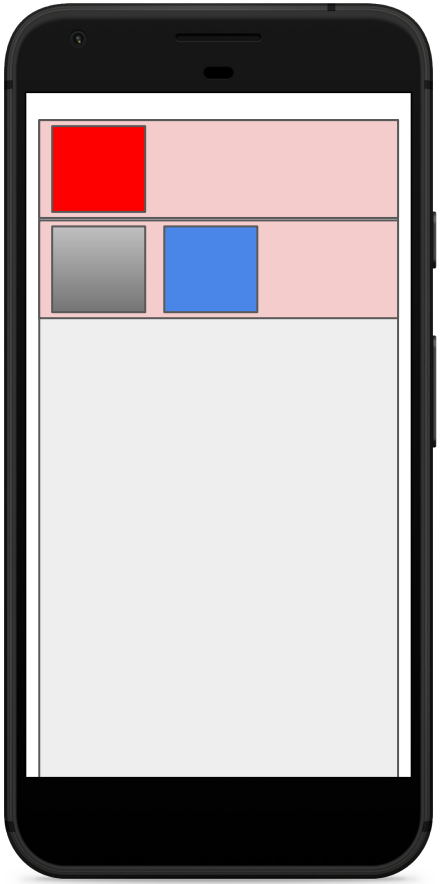


Cost of Nested Layouts

- Deeply nested ViewGroups require more computation.
- views may be measured multiple times.
- Can cause UI slowdown and lack of responsiveness

Use ConstraintLayout to avoid some of these issues!





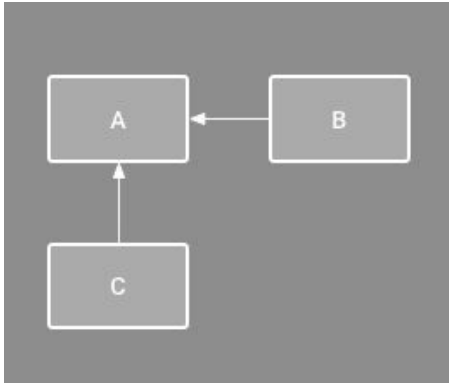
Benefits of Constraint Layout

- Recommended default layout for Android
- Solves costly issue of too many nested layouts, while allowing complex behavior
- Position and size views within it using a set of constraints

Core Layouts - Constraint Layout

Constraint Layout: Allows complex layout behavior while mitigating problem of having too many nested layout

- Default Android layout
- Uses constraints to create views for position and size
- Focuses positions of items respective to one another
  - B always constrained to the right of A
  - C always constrained to the bottom of A



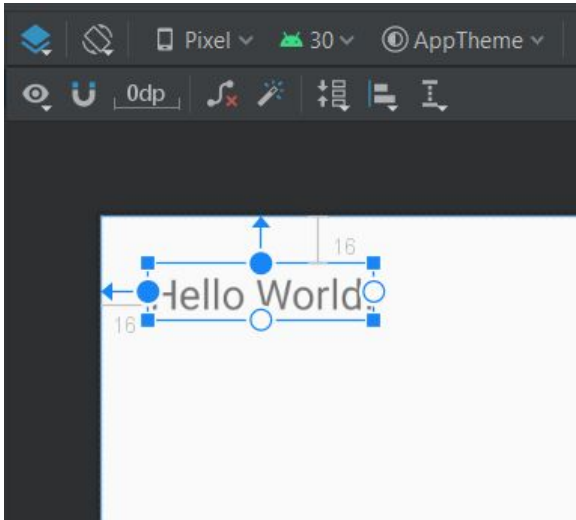
Parents and Positioning - Constraint Layout

Constraints can be set up relative to parent containers in the following general form:

- `layout_constraint<`**`SourceConstraint`**`>_to<`**`TargetConstraint`**`>Of=” ”`

Where on a TextView the following attributes could be observed as:

- `app:layout_constraint`**`Top`**`_to`**`Top`**`Of="`**`parent`**`"`
- `app:layout_constraint`**`Left`**`_to`**`Left`**`Of="`**`parent`**`"`



Relative Positioning - Constraint Layout

Vertical Direction

- Can add a constraint to the top or bottom of an element
- Or to the text's baseline if the View contains text

Horizontal Direction

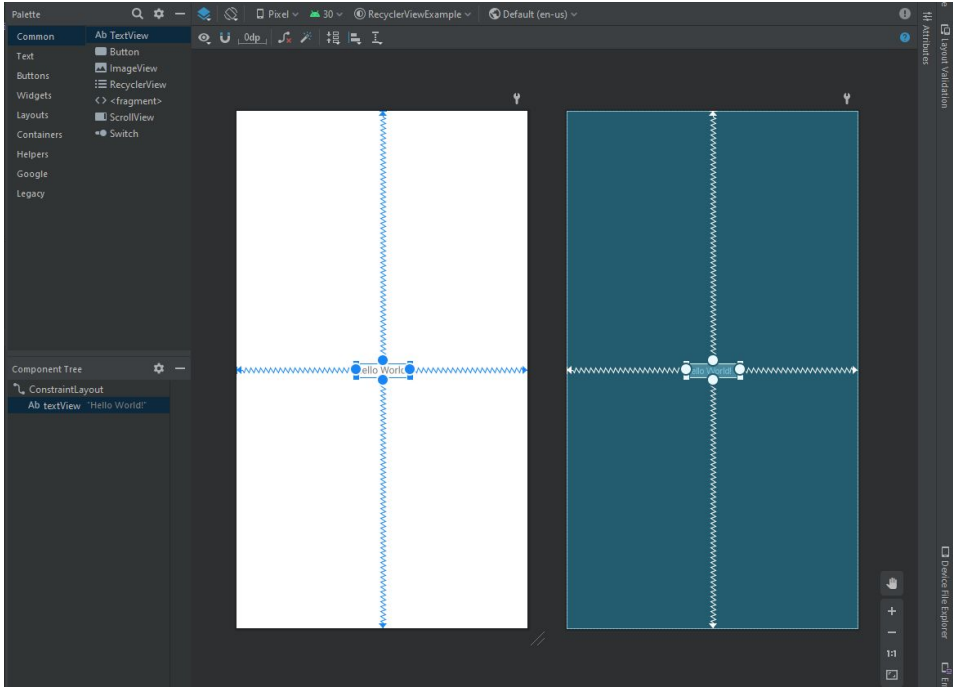
- Can constrain the start and end edges of a View
- Start = Left
- End = Right

Example (Code vs Drag & Drop) - Constraint Layout

Code XML elements OR Drag & Drop XML elements

MainActivity.kt | activity_main.xml

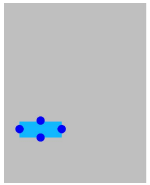
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:and
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

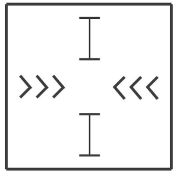


Layout Editor - Constraint Widget

There are three constraints & symbols for each type:

Fixed:

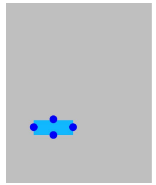


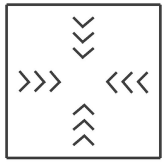


layout_width wrap_content

layout_height 48dp

Wrap Content:

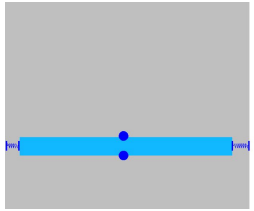


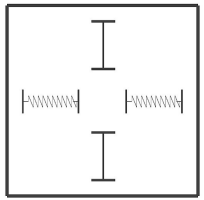


layout_width wrap_content

layout_height wrap_content

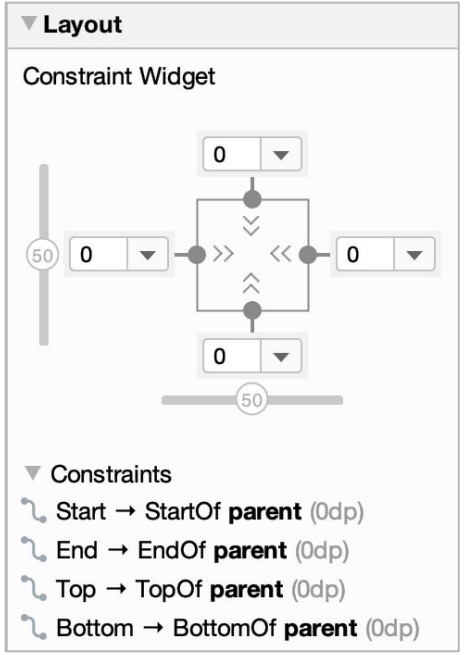
Match Constraints:





layout_width 0dp(match_constraint)

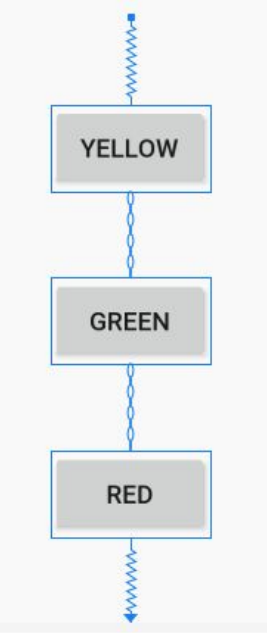
layout_height 48dp



Chains - Constraint Layout

A chain is a group of views that are bi-directionally linked to each other with position constraints.

- Can position views in relation with one another
- Can be linked horizontally or vertically

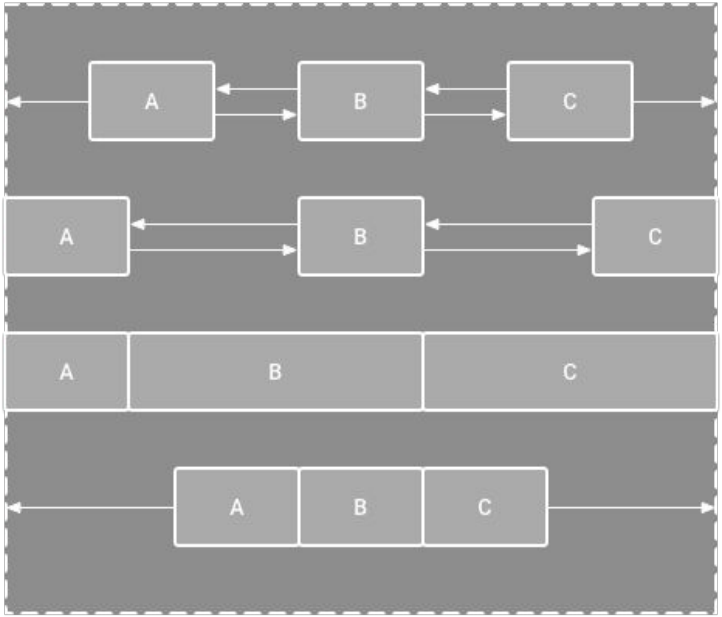


Chain creation is accomplished in the Layout Editor by:

1. Selecting the objects you want to be in the chain.
2. Right-clicking and select **Chains.**
3. Creating a horizontal or vertical chain.

Chain Styles - Constraint Layout

There are multiple types of chains for adjusting space between views:



Spread Chain

Spread Inside Chain

Weighted Chain

Packed Chain

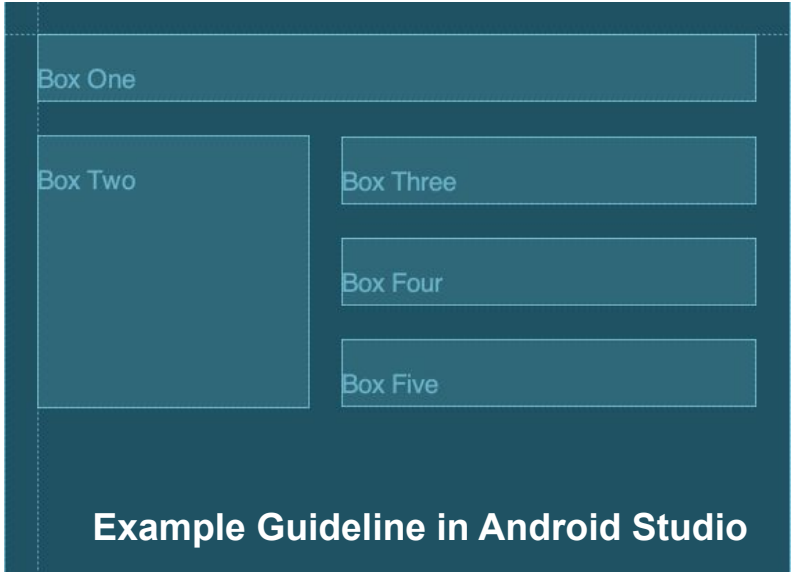
Guidelines (Advanced) - Constraint Layout

Guidelines are only used for layout purposes and allows positioning of multiple views in relation to a single guide

- Can be vertical or horizontal
- Are not displayed on the device
- Enables collaboration between UI/UX teams

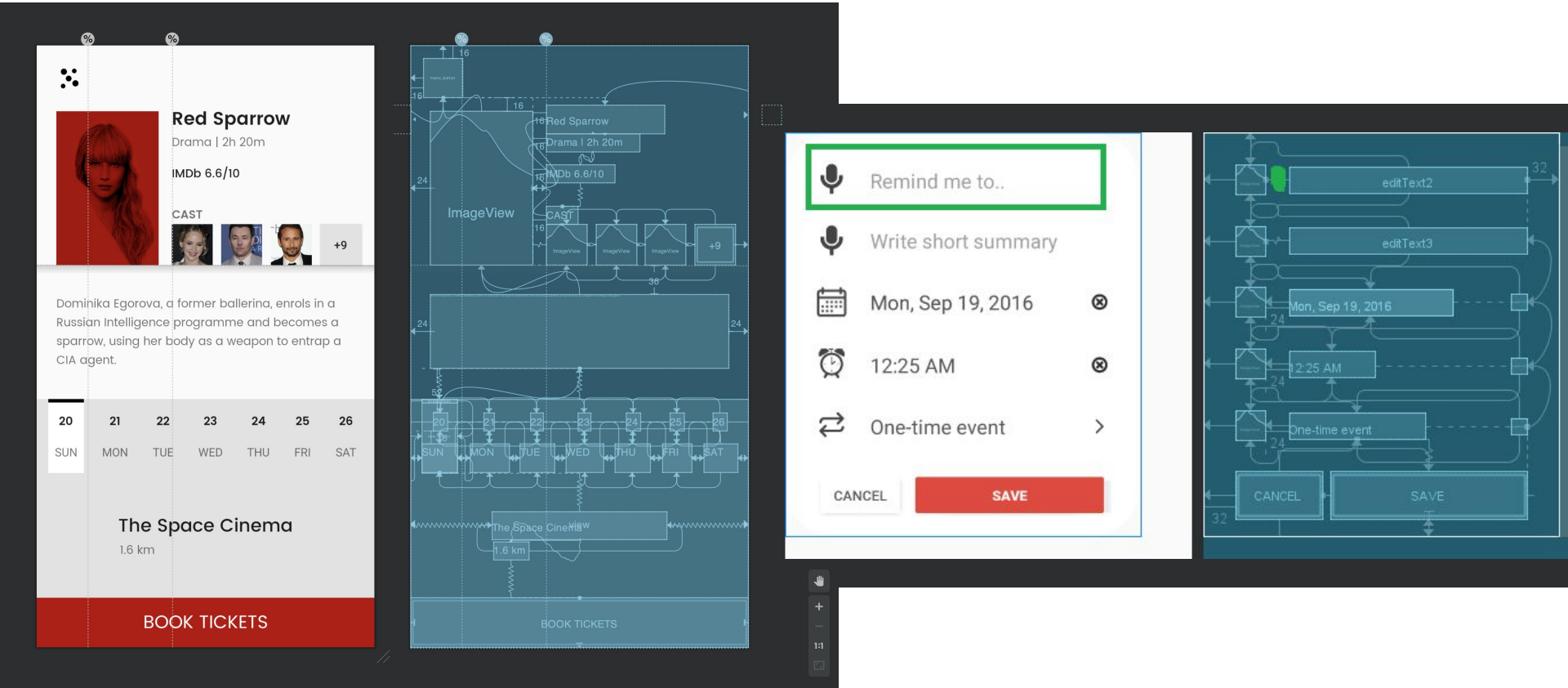
Guidelines can be created via:

- layout_constraintGuide_begin
- layout_constraintGuide_end
- layout_constraintGuide_percent
    - Between 0 and 1



Example Guideline in Android Studio

Real World Example - Constraint Layout



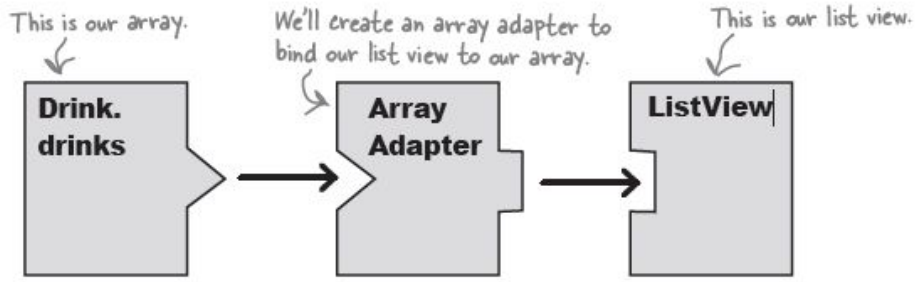
RecyclerView

Displaying Lists of information

- It is inevitable that as a developer you will need to display information and views in a list
- What methods can be used to achieve making a list
  - ✘ LinearLayout inside a ScrollView
  - ✘ ListView
  - ✔ RecyclerView

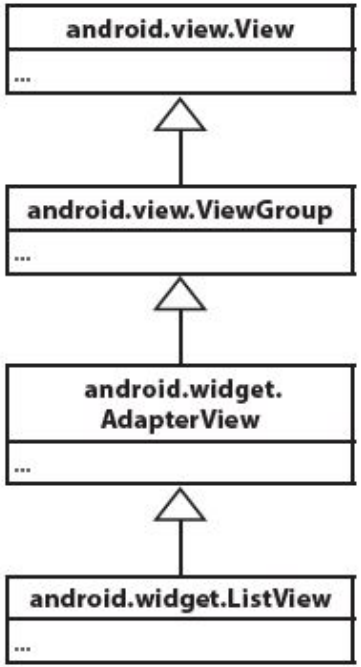
ListView

Displays a vertically-scrollable collection of views, where each view is positioned immediately below the previous view in the list.




```
//Add the listener to the list view
ListView listView = (ListView) findViewById(R.id.list_options);
listView.setOnItemClickListener(itemClickListener);
```

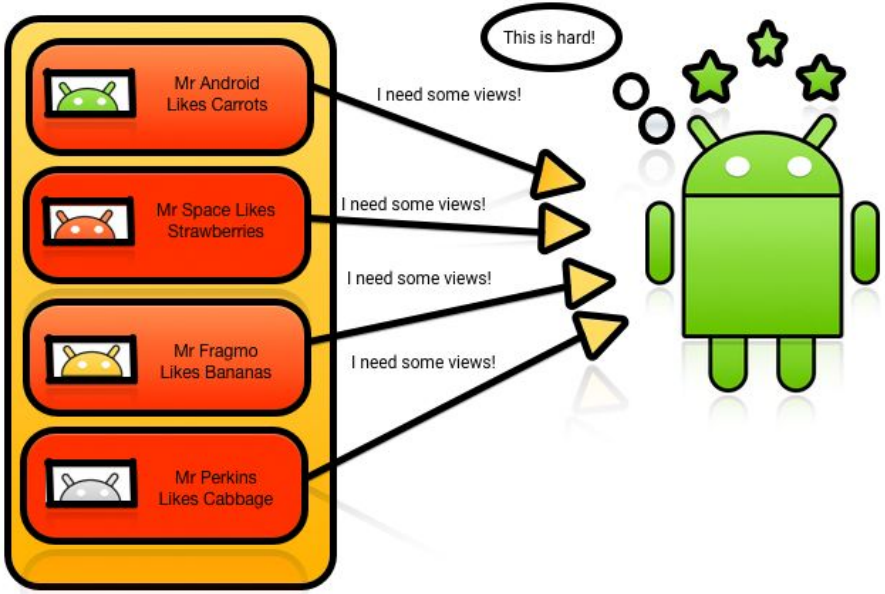
Add the listener to the list view.



So why is this not a good option?

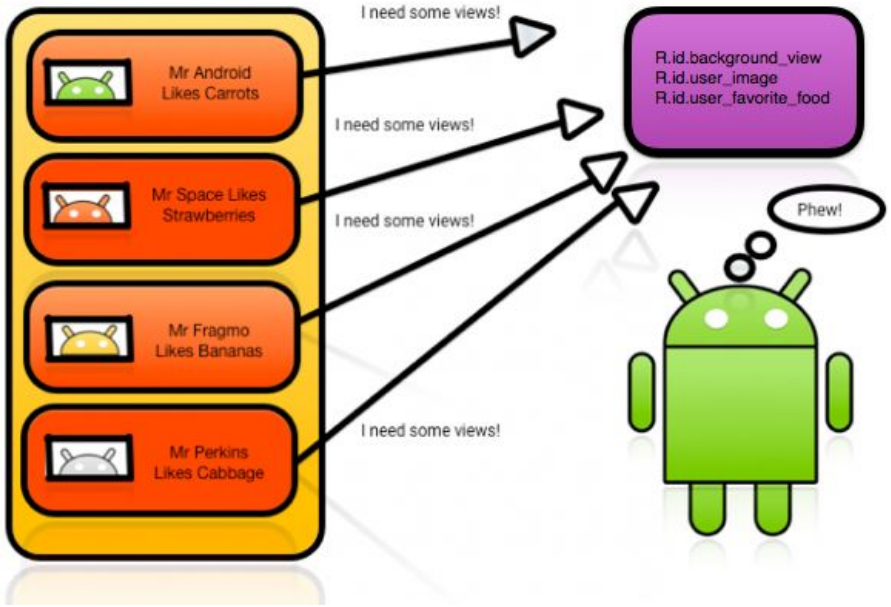
Problem with ListView

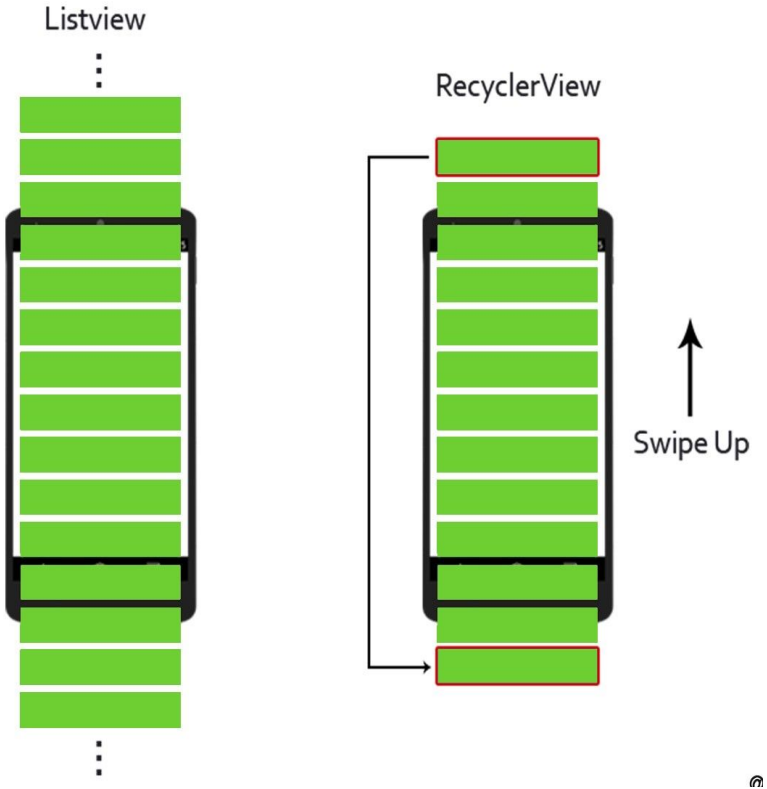
- Renders every list item's view in the ListView at all times
- Has a huge cost on performance for larger lists with more complex views
- Only has a LayoutManager that supports a vertical ListView
- Solution: RecyclerView



RecyclerView

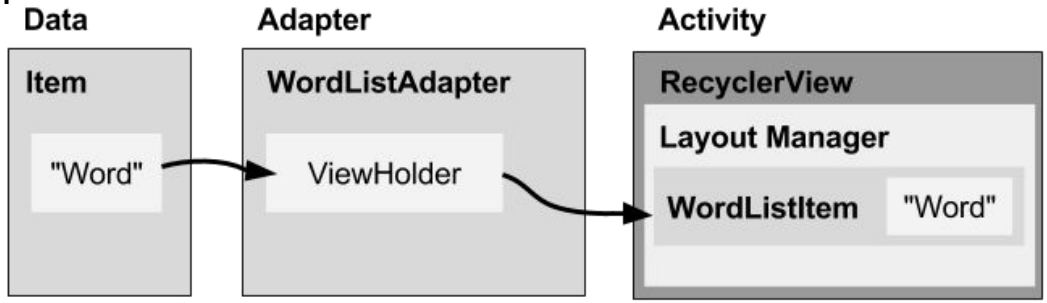
- Contains a fixed number of views
- Reuses views that are no longer visible



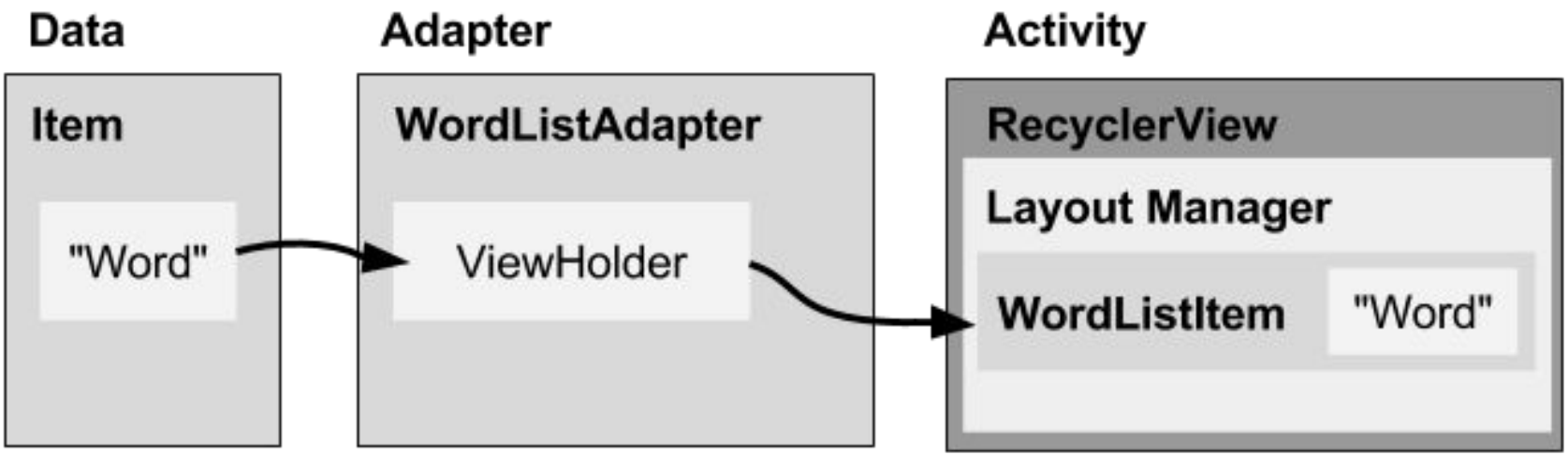


RecyclerView components

- **Data** contains the item
- **RecyclerView** scrolling list for list items—RecyclerView
- **Layout** for one item of data—XML file
- **Layout manager** handles the organization of UI components in a View—Recyclerview.LayoutManager
- **Adapter** connects data to the RecyclerView—RecyclerView.Adapter
- **ViewHolder** has view information for displaying one item—RecyclerView.ViewHolder

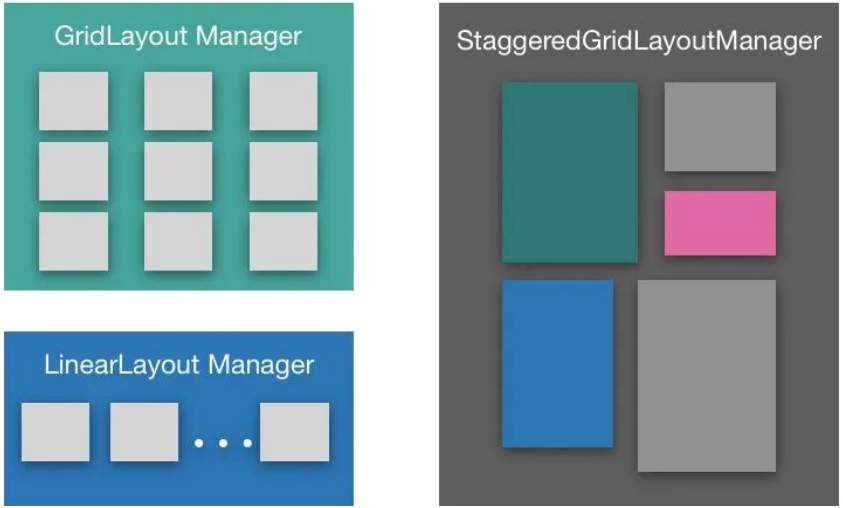


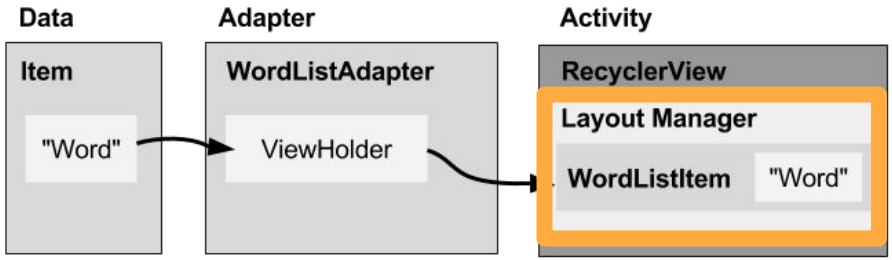
Putting the components together



Layout Manager

- Measures and positions item views
- Handles recycling unused views
- Common Layout Managers:
  - LinearLayoutManager
  - GridLayoutManager
  - StaggeredGridLayoutManager

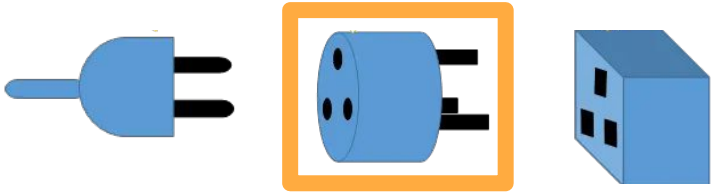


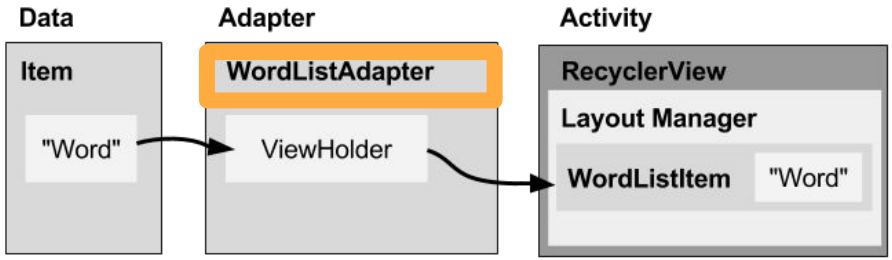


Adapter

Helps incompatible interfaces work together

- Takes data from a database and prepares data to put into a View
- Intermediary between the Data and View
- Manages creating, updating, adding, deleting View items as underlying data changes

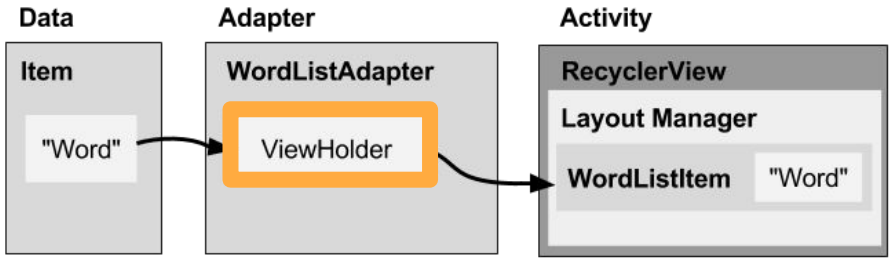




ViewHolder

Used by the adapter to prepare one View with data for one list item

- Describes an item view and metadata about its place inside a RecyclerView
- Specified with a layout file
- Is placed by the layout manager



Putting to Practice

1. RecyclerView dependency
2. Add RecyclerView to layout
3. Create list item layout
4. Create the list adapter
    4.1. Create ViewHolder
    4.2. Define onCreateViewHolder()
    4.3. Define onBindViewHolder()
    4.4. Define getItemCount()
5. Create the RecyclerView in Activity onCreate()

1 - Add Recycler dependency

```
dependencies {
      ...
      implementation 'androidx.recyclerview:recyclerview:1.1.0'
      ...
}
```

build.gradle (Module:app level)

2 - Add RecyclerView to XML Layout

```
<!-- place RecyclerView in Constraint Layout -->
<androidx.recyclerview.view.RecyclerView
    android:id="@+id/my_recycler_view"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

activity_main.xml

3 - Create list item layout (for 1 item)

```
<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
>
  <TextView
    android:id="@+id/word"
    style="@style/word_title" />
</LinearLayout>
```

wordlist_item.xml

4.0 - Create the list adapter

```
class WordListAdapter(private val mWordList: List<String>): RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {

    class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)  {
        val wordItemView: TextView = itemView.findViewById(R.id.word)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        val view = layoutInflater.inflate(R.layout.wordlist_item, parent, false)
        return WordViewHolder(view)
    }

    override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
        holder.wordItemView.text = mWordList[position]
    }

    override fun getItemCount(): Int = mWordList.size
}
```

WordListAdapter.kt

4.1 - Create the viewholder in adapter class

```
class WordListAdapter(private val mWordList: List<String>): RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {

    class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)  {
        val wordItemView: TextView = itemView.findViewById(R.id.word)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        val view = layoutInflater.inflate(R.layout.wordlist_item, parent, false)
        return WordViewHolder(view)
    }

    override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
        holder.wordItemView.text = mWordList[position]
    }

    override fun getItemCount(): Int = mWordList.size
}
```

WordListAdapter.kt

4.2 - onCreateViewHolder()

```
class WordListAdapter(private val mWordList: List<String>): RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {

    class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)  {
        val wordItemView: TextView = itemView.findViewById(R.id.word)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        val view = layoutInflater.inflate(R.layout.wordlist_item, parent, false)
        return WordViewHolder(view)
    }

    override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
        holder.wordItemView.text = mWordList[position]
    }

    override fun getItemCount(): Int = mWordList.size
}
```

WordListAdapter.kt

4.3 - onBindViewHolder()

```
class WordListAdapter(private val mWordList: List<String>): RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {

      class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)  {
            val wordItemView: TextView = itemView.findViewById(R.id.word)
      }

      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater.inflate(R.layout.wordlist_item, parent, false)
            return WordViewHolder(view)
      }

      override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
            holder.wordItemView.text = mWordList[position]
      }

      override fun getItemCount(): Int = mWordList.size
}
```

WordListAdapter.kt

4.4 - getItemCount()

```
class WordListAdapter(private val mWordList: List<String>): RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {

      class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)  {
            val wordItemView: TextView = itemView.findViewById(R.id.word)
      }

      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val view = layoutInflater.inflate(R.layout.wordlist_item, parent, false)
            return WordViewHolder(view)
      }

      override fun onBindViewHolder(holder: WordViewHolder, position: Int) {
            holder.wordItemView.text = mWordList[position]
      }

      override fun getItemCount(): Int = mWordList.size
}
```

WordListAdapter.kt

5 - Create the RecyclerView in Activity onCreate()

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

     val recyclerView = findViewById<RecyclerView>(R.id.my_recycler_view)
     recyclerView.layoutManager = LinearLayoutManager(this@MainActivity)
     recyclerView.adapter = WordListAdapter(listOf("one", "two", "three"))
}
```

MainActivity.kt

Final Result