CSCI3310 Mobile Computing & Application Development

Lab 09 – Hello Kotlin: fun with colors

Introduction

The FunWithColors app is adopted from Kotlin Android Fundamentals. The app consists of clickable text views that change color when tapped, and button views in a constraintLayout.

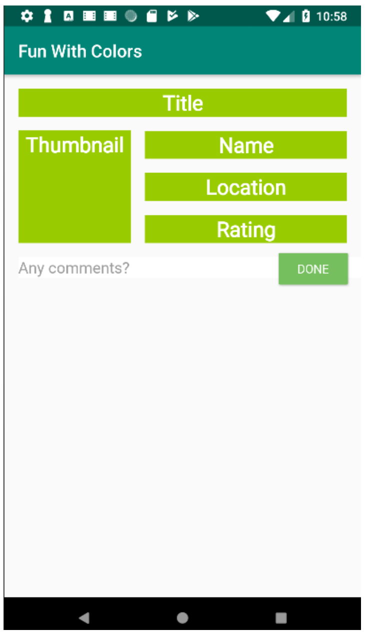
Objective

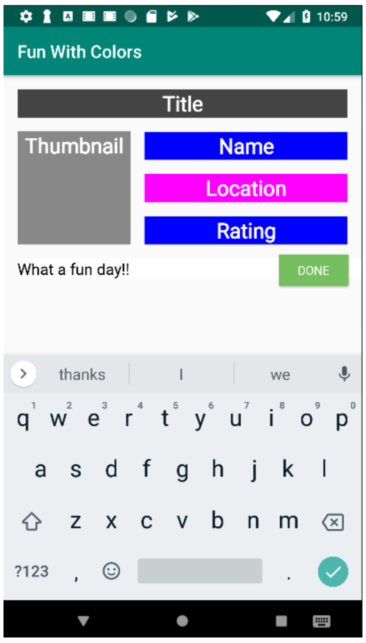
Get familiar with Kotlin Android through the following tasks:

- How to use ConstraintLayout in your app to arrange views.
- How to change a text view's background color.
- How to align views using a baseline constraint.
- How to create vertical and horizontal chains from a group of views.
- How to add interactivity to the UI.

Todo

- Create a FunWithColors app.
- Add click handlers to text views to change the views' color when the user taps them.
- Add text views with different font sizes and align them using a baseline constraint.





1: Create the FunWithColors project

In this task, you create a new Kotlin app.

1. Open Android Studio if it is not already opened. In this example, we use Android Studio 4.1; options available in earlier versions shall differ.
2. In the main **Welcome to Android Studio** dialog, click **Start a new Android Studio project**.

3. The **Choose your project** dialog appears. Select **Empty Activity** as shown below, and click **Next**
4. In the **Configure your project** dialog, enter "FunWithColors" for the **Name**.
5. Create your own package name, say **edu.cuhk.csci3310.funwithcolors**.
6. Make sure the **Language** is Kotlin. (or check Kotlin support for Android Studio 4.x)
7. Make sure the Minimum API level is **API 19: Android 4.4 (KitKat)**. At the time this lab was written, Android Studio indicated that with this API level, the app would run on approximately 98.1% of devices.
8. Leave all the other checkboxes cleared, and click **Finish**. If your project requires more components for your chosen target SDK, Android Studio installs them automatically, which might take a while. Follow the prompts and accept the default options.

1.1: Create and examine MainActivity.kt

Similar to coding in Java, MainActivity is inherited from the core Android class Activity that draws an Android app user interface (UI) and receives input events. Many programming languages including Kotlin define the main method that starts the program. As you should have noticed, Android apps don't have a main method. It is when the user taps the app's launcher icon to launch an activity based on the information in the AndroidManifest.xml file. There is no difference between Java or Kotlin based apps.

1. In the **Project > Android** pane, expand **java > edu.cuhk.csci3310.funwithcolors**. Double-click **MainActivity**. The code editor shows the code in MainActivity.
2. Below the package name and import statements is the class declaration for MainActivity. The MainActivity class extends AppCompatActivity.

```
class MainActivity : AppCompatActivity() {
```

3. Notice the onCreate() method. Same as Java, Activities in Kotlin do not use a constructor to initialize the object. Again, a series of lifecycle methods are called as part of the activity setup. In onCreate(), you specify which layout is associated with the activity, and you inflate the layout. The setContentView() method does both those things.

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

Note: the nullable variable Bundle which means the savedInstanceState Bundle could be NULL.

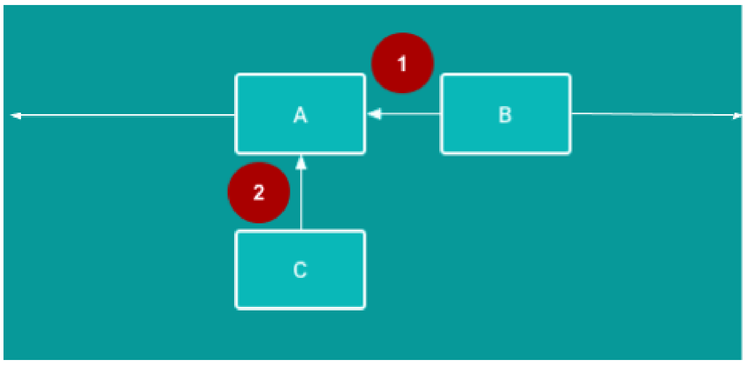
1.2: Build more on the ConstraintLayout through the Layout Editor

A ConstraintLayout is a ViewGroup that allows you to position and size child views in a flexible way. A constraint layout allows you to create large, complex layouts with flat view hierarchies (no nested view groups). To build a constraint layout, you can use the Layout Editor to add constraints and to drag-and-drop views. You don't need to edit the XML. In this task, you use more functionalities in the Android Studio Layout Editor to build a more advanced constraint layout for your app.

Note: ConstraintLayout is available as a support library, which is available in API level 9 and higher.

Constraints

A constraint is a connection or alignment between two UI elements. Each constraint connects or aligns one view to another view, to the parent layout, or to an invisible guideline. In a constraint layout, you position a view by defining at least one horizontal and one vertical constraint.



1 Horizontal constraint: B is constrained to stay to the right of A. (In a finished app, B would need at least one vertical constraint in addition to this horizontal constraint.)

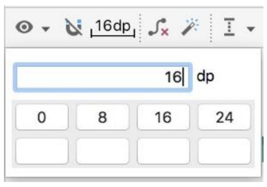
2 Vertical constraint: C is constrained to stay below A. (In a finished app, C would need at least one horizontal constraint in addition to this vertical constraint.)

1.2.1. Set up your Android Studio work area

1. Open the activity_main.xml file and click the **Design** tab.
2. You'll add constraints manually, so you want autoconnect turned off. In the toolbar, locate the **Turn Off/On Autoconnect** toggle button, which is shown below. (If you can't see the toolbar, click inside the design editor area of the Layout Editor.) Make sure autoconnect is off.

   Autoconnect is on.

   Autoconnect is off—this is what you want for this lab.
3. Use the toolbar to set the default margins to 16dp. (The default is 8dp.)

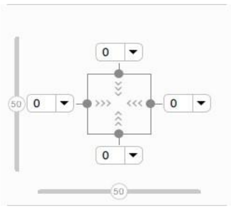


   When you set the default margin to 16dp, new constraints are created with this margin, then you don't have to add the margin each time you add a constraint.

4. Zoom in using the **+** icon 33% on the right side of the toolbar, until the **Hello World** text is visible inside its text view.
5. Double-click on the **Hello World** text view to open the **Attributes** pane.

The view inspector

The view inspector, shown in the screenshot below, is a part of the **Attributes** pane. The view inspector includes controls for layout attributes such as constraints, constraint types, constraint bias, and view margins.

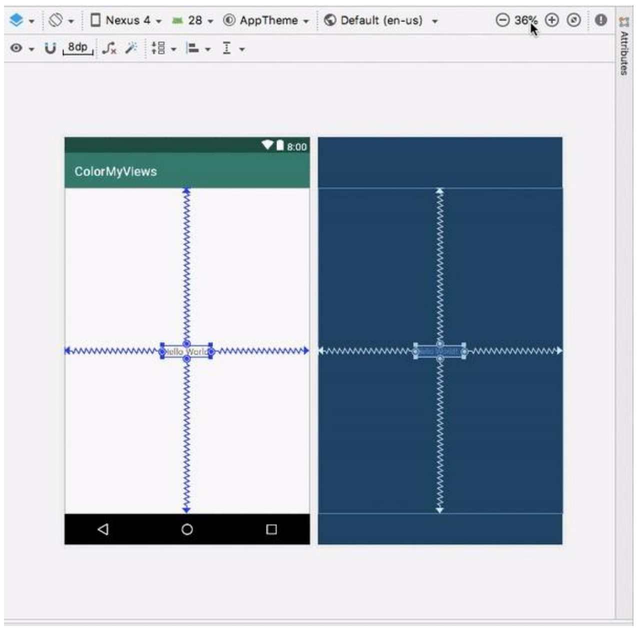


Tip: The view inspector is available *only* for views that are inside a ConstraintLayout.

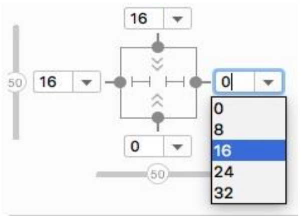
Constraint bias

Constraint bias positions the view element along the horizontal and vertical axes. By default, the view is centered between the two constraints with a bias of 50%.

To adjust the bias, you can drag the bias sliders in the view inspector. Dragging a bias slider changes the view's position along the axis.



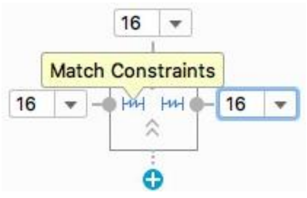
1.2.2: Add margins for the Hello World text view



1. Notice that in the view inspector, the left, right, top, and bottom margins for the text view are 0. The default margin was not automatically added because this view was created before you changed the default margin.
2. For the left, right, and top margins, select 16dp from the drop-down menu in the view inspector. For example, in the following screenshot, you are adding layout_marginEnd (layout_marginRight).

1.2.3: Adjust constraints and margins for the text view

In the view inspector, the arrows >>> inside the square represents the type of the constraint:

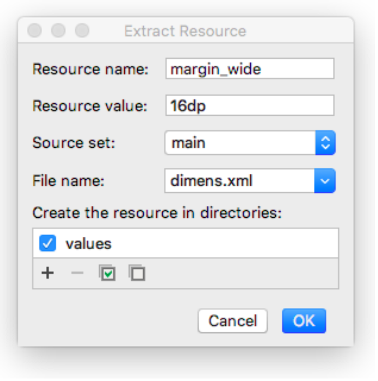


- >>>
- ⊢⊣ **Wrap Content**: The view expands only as much as needed to contain its contents.
- ⊢⋀⋀⊣ **Fixed**: You can specify a dimension as the view margin in the text box next to the fixed-constraint arrows.

Match Constraints: The view expands as much as possible to meet the constraints on each side, after accounting for the view's own margins. This constraint is very flexible because it allows the layout to adapt to different screen sizes and orientations. By letting the view match the constraints, you need fewer layouts for the app you're building.

1. In the view inspector, change the left and right constraints to **Match Constraints** . (Click the arrow symbol to toggle between the constraint types.)
2. In the view inspector, click the **Delete Bottom Constraint** dot on the square to delete the bottom constraint.
3. Switch to the **Code** tab. Extract the dimension resource (select **16dp** and press Alt-Enter) for layout_marginStart (layout_marginLeft)., and set the **Resource name** to margin_wide.
4. Set the same dimension resource, @dimen/margin_wide, for the top and end margins.

```
android:layout_marginStart="@dimen/margin_wide"
android:layout_marginTop="@dimen/margin_wide"
android:layout_marginEnd="@dimen/margin_wide"
```

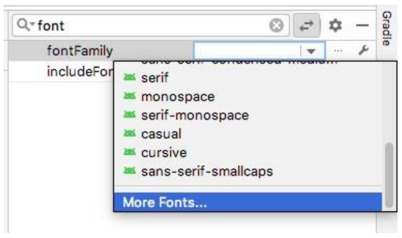


Resource name: margin_wide
Resource value: 16dp
Source set: main
File name: dimens.xml
Create the resource in directories:
[✓] values

1.3 Style the TextView

1.3.1: Add a font

1. In the **Attributes** pane, search for fontFamily and select the drop-down arrow next to it. Scroll down to **More Fonts** and select it. The **Resources** dialog opens.
2. In the **Resources** dialog, search for roboto.
3. Click **Roboto** and select **Regular** in the **Preview** list.
4. Select the **Add font to project** radio button.
5. Click **OK**.



This adds a res/font folder that contains a roboto.ttf font file. Also, the @font/roboto attribute is set for your text view.

1.3.2: Add a style

1. Open res/values/dimens.xml, and add the following dimension resource for the font size.

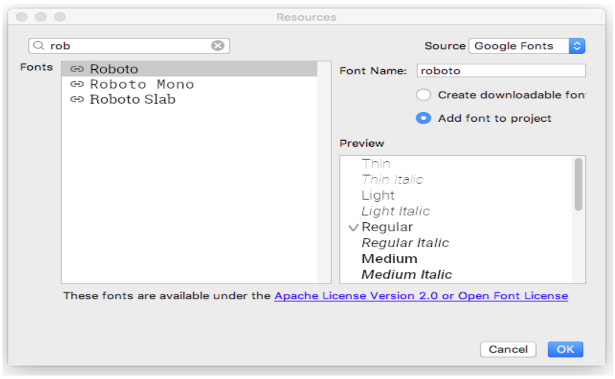
```
<dimen name="box_text_size">24sp</dimen>
```

2. Right-click res/values and create a Value Resources File named “style.xml”. Add the following style code into the file, which you will use for the text view.

```
<resources>
<style name="greenBox">
    <item name="android:background">@android:color/holo_green_light</item>
    <item name="android:textAlignment">center</item>
    <item name="android:textSize">@dimen/box_text_size</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">@android:color/white</item>
    <item name="android:fontFamily">@font/roboto</item>
</style>
</resources>
```

In this style, the background color and the text color are set to default Android color resources. The font is set to Roboto. The text is center aligned and bolded, and the text size is set to box_text_size.

1.3.3: Add a string resource for the text view



Resources

rob

Source: Google Fonts

Fonts: Roboto, Roboto Mono, Roboto Slab

Font Name: roboto

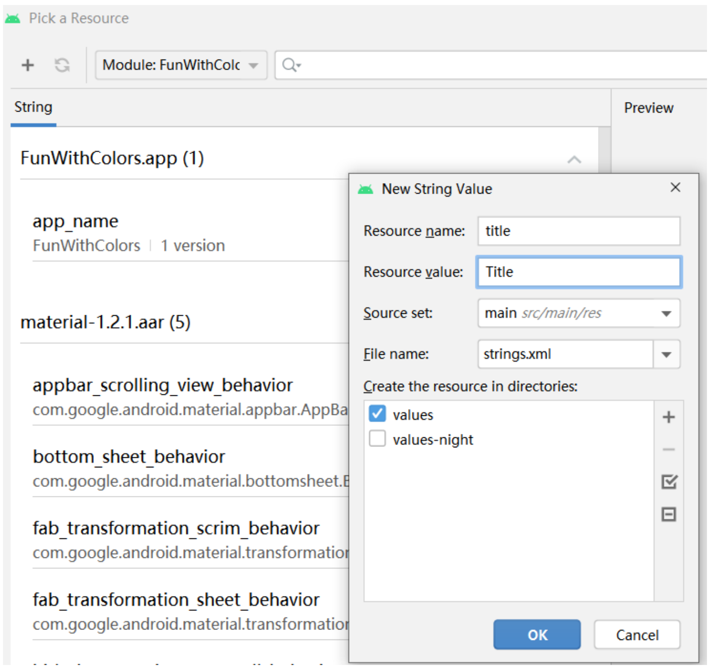
( ) Create downloadable fon
(●) Add font to project

Preview: Thin, Thin Italic, Light, Light Italic, ✓ Regular, Regular Italic, Medium, Medium Italic

These fonts are available under the Apache License Version 2.0 or Open Font License

Cancel | OK

1. In the **Attributes** pane, find the **text** attribute. (You want the one without the wrench icon.)
2. Click the **...** (three dots) next to the **text** attribute to open the **Resources** dialog.
3. In the **Resources** dialog, find and click + on the left top of the pane. Select **String Value**. Set the resource name to the title, and set the value to Title.
4. Click **OK**.


Pick a Resource

Module: FunWithColc

String

Preview

FunWithColors.app (1)

app_name
FunWithColors | 1 version

material-1.2.1.aar (5)

appbar_scrolling_view_behavior
com.google.android.material.appbar.AppBa

bottom_sheet_behavior
com.google.android.material.bottomsheet.E

fab_transformation_scrim_behavior
com.google.android.material.transformatior

fab_transformation_sheet_behavior
com.google.android.material.transformatior

New String Value

Resource name: title

Resource value: Title

Source set: main *src/main/res*

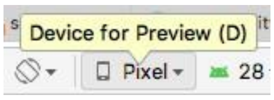
File name: strings.xml

Create the resource in directories:

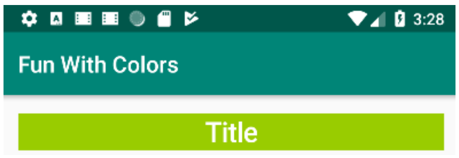
- [x] values
- [ ] values-night

1.3.4: Finish setting attributes for the text view

1. In the **Attributes** pane, set the id (android:id) of the text view to title_text.
2. Set the style to @style/greenBox.
3. To clean up the code, switch to the **Code** tab and remove the android:fontFamily="@font/roboto" attribute, because this font is present in the greenBox style.
4. Switch back to the **Design** tab. At the top of the design editor, click the **Device for preview (D)** button. A list of device types with different screen configurations is displayed. The default device is **Pixel**.



5. Select different devices from the list and see how the TextView adapts to the different screen configurations.
6. Run your app. You see a styled green text view with the text "Title".



1.4 Add a second TextView and add constraints

In this task, you add another text view below the title_text. You constrain the new text view to title_text and the layout's parent element.

1.4.1: Add a new text view

1. Open the activity_main.xml file and switch to the **Design** tab.
2. Drag a TextView from the **Palette** pane directly into the design editor preview, as shown below. Place the text view below the title_text, aligned with the left margin.
3. In the design editor, click the new text view, then hold the pointer over the dot on the top side of the text view. This dot, shown below, is called a *constraint handle*.

Notice that when you hold the pointer over the constraint handle, the handle turns green and blinks.

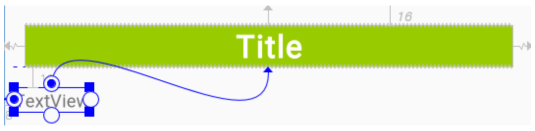
1.4.2: Add constraints to the new text view

Create a constraint that connects the top of the new text view to the bottom of the **Title** text view:

1. Hold the pointer over the top constraint handle on the new text view.

   Click the top constraint handle of the view and drag it up to connect the constraint line to the bottom of the **Title** textview, as shown below. As you release the click, the constraint is created, and the new text view jumps to within 16dp of the bottom of the **Title**. (The new text view has a top margin of 16dp because that's the default you set earlier.)

Now create a left constraint:



2. Click the constraint handle on the left side of the new view.

3. Drag the constraint line to the left edge of the layout.

1.4.3: Set attributes for the new text view

1. Open res/values/strings.xml. Add a new string resource with the following code:

```
<string name="thumbnail">Thumbnail</string>
```

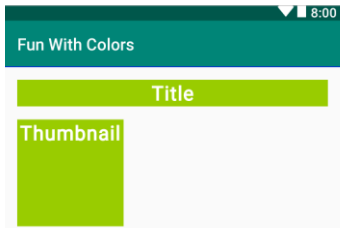
2. Open activity_main.xml and click the **Design** tab. Use the **Attributes** pane to set the following attributes on the new text view:

Attribute	Value
id	thumbnail_text
layout_height	130dp
layout_width	130dp
style	@style/greenBox
text	@string/thumbnail

In this case, you're assigning fixed sizes for the height and width of the text view. Assign fixed sizes for height and width only if your view should always have a fixed size on *all* devices and layouts.

Important: When developing real-world apps, use flexible constraints for the height and width of your UI elements, whenever possible. For example, use match_constraint or wrap_content. The more fixed-size UI elements you have in your app, the less adaptive your layout is for different screen configurations.

3. Run your app. You should see two green TextView views, one above the other, similar to the following screenshot:

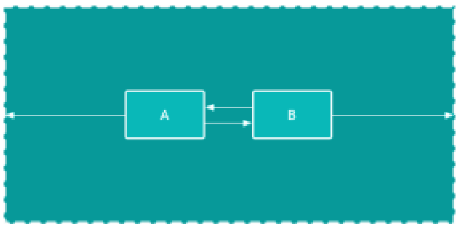


1.5 Create a chain of TextView views

In this task, you add three TextView views. The text views are vertically aligned with each other and horizontally aligned with the **Thumbnail** text view. The views will be in a *chain*.

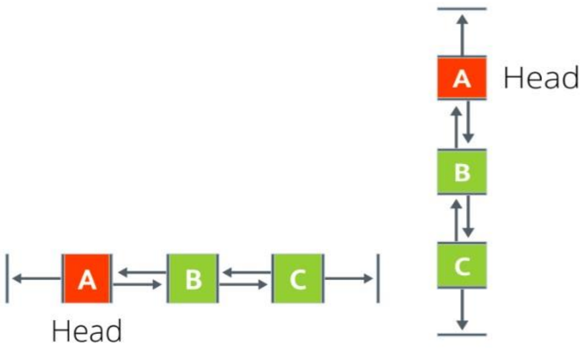
Chains

A chain is a group of views that are linked to each other with bidirectional constraints. The views within a chain can be distributed either vertically or horizontally. For example, the following diagram shows two views that are constrained to each other, which creates a horizontal chain.



Head of the chain

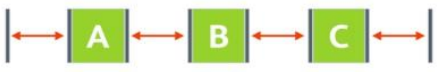
The first view in a chain is called the *head* of the chain. The attributes that are set on the head of the chain control, position, and distribute all the views in the chain. For horizontal chains, the head is the left-most view. For vertical chains, the head is the top-most view. In each of the two diagrams below, "A" is the head of the chain.



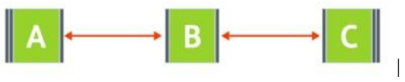
Chain styles

Chain styles define the way the chained views are spread out and aligned. You style a chain by assigning a chain style attribute, adding weight, or setting bias on the views.

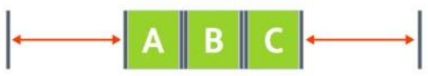
There are three chains styles:

- **Spread:** This is the default style. Views are evenly spread in the available space after margins are accounted for. 

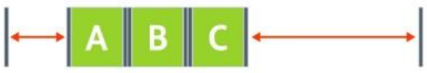
- **Spread inside:** The first and the last views are attached to the parent on each end of the chain. The rest of the views are evenly spread in the available space.



- **Packed:** The views are packed together after margins are accounted for. You can then adjust the position of the whole chain by changing the bias of the chain's head view.

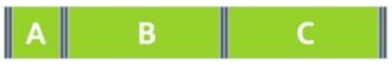


Packed chain



Packed chain with bias

- **Weighted:** The views are resized to fill up all the space, based on the values set in the layout_constraintHorizontal_weight or layout_constraintVertical_weight attributes. For example, imagine a chain containing three views, A, B, and C. View A uses a weight of 1. Views B and C each use a weight of 2. The space occupied by views B and C is twice that of view A, as shown below.



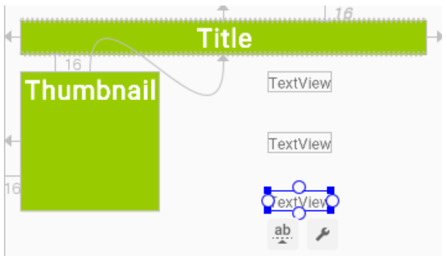
To add a chain style to a chain, set the layout_constraintHorizontal_chainStyle or the layout_constraintVertical_chainStyle attribute for the head of the chain. You can add chain styles in the Layout Editor, which you learn in this task.

Alternatively, you can add chain styles in the XML code. For example:

```
// Horizontal spread chain
app:layout_constraintHorizontal_chainStyle="spread"
// Vertical spread inside chain
app:layout_constraintVertical_chainStyle="spread_inside"
// Horizontal packed chain
app:layout_constraintHorizontal_chainStyle="packed"
```

1.5.1: Add three text views and create a vertical chain

1. Open the activity_main.xml file in the **Design** tab. Drag three TextView views from the **Palette** pane into the design editor. Put all three new text views to the right of the **Thumbnail** text view, as shown below.

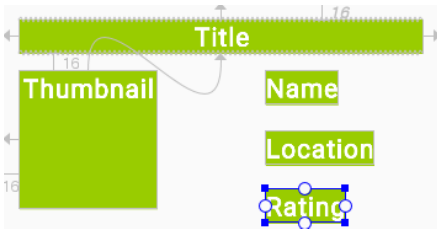


2. In the strings.xml file, add the following string resources for the names of the new text views:

```
<string name="name">Name</string>
<string name="location">Location</string>
<string name="rating">Rating</string>
```

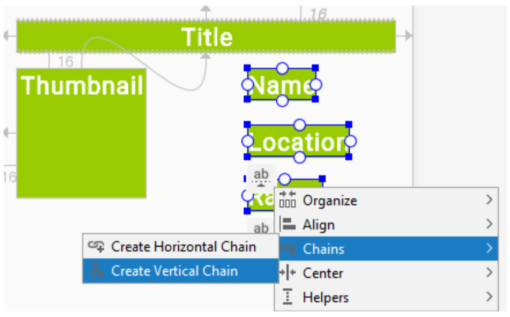
3. Set the following attributes for the new text views:

Attribute	Top text view	Middle text view	Bottom text view
ID	name_text	location_text	rating_text
text	@string/name	@string/location	@string/rating
style	@style/greenBox	@style/greenBox	*@style/greenBox*



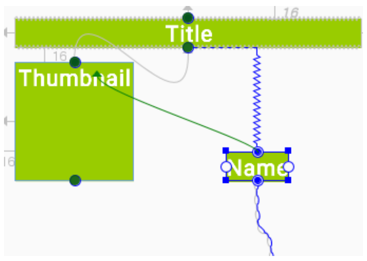
1.5.2: Create a chain and constrain it to the height of Thumbnail

1. Select all three new text views, right-click, and select **Chains** > **Create Vertical Chain**.

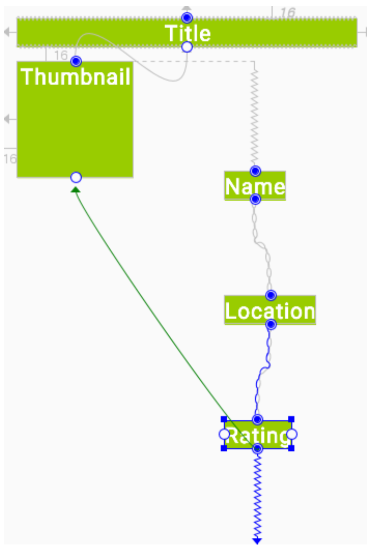


This creates a vertical chain that extends from **Title** to the end of the layout.

2. Add a constraint that extends from the top of **Name** to the top of **Thumbnail.** This removes the existing top constraint and replaces it with the new constraint. You don't have to delete the constraint explicitly.



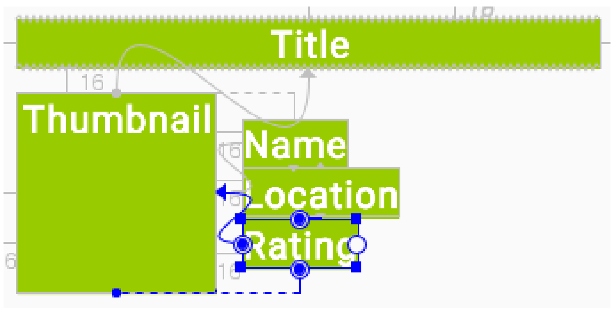
3. Add a constraint from the bottom of **Rating** to the bottom of **Thumbnail**.



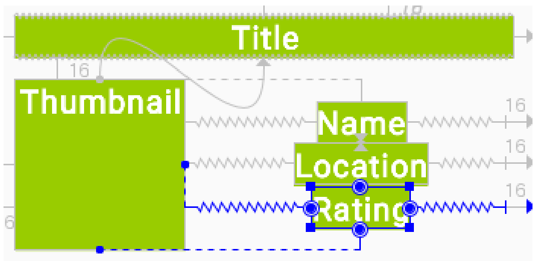
Observe that the three text views are now constrained to the top and bottom of **Thumbnail**.

1.5.3: Add left and right constraints

1. Constrain the left side of **Name** to the right side of **Thumbnail**. Repeat for **Location** and **Rating**, constraining the left side of each to the right side of **Thumbnail**.



2. Constrain the right side of each of the three text views to the right side of the layout.

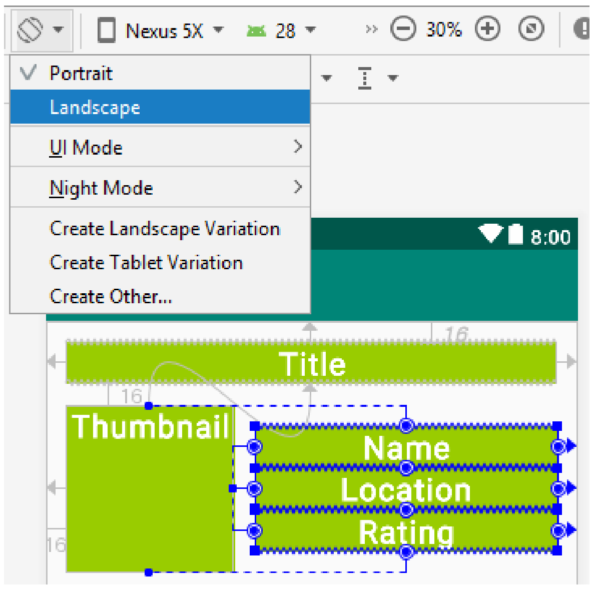


3. For each of the three text views, change the layout_width attribute 0dp, which is equivalent to changing the constraint type to **Match Constraints**.

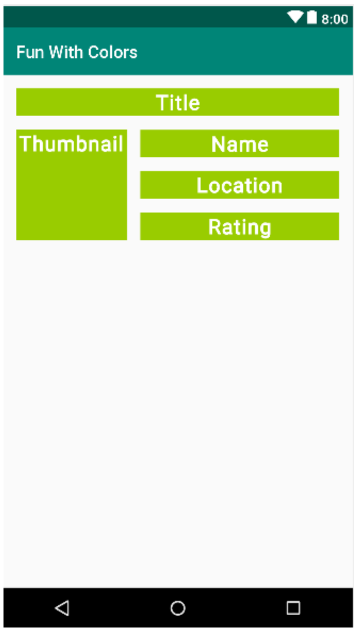
1.5.4: Add margin

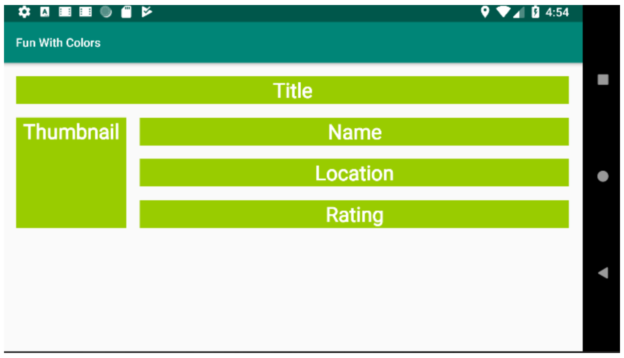
Use the **Attributes** pane to set **Layout_margin** attributes on the three text views to add spacing between them.

1. For **Name**, use @dimen/margin_wide for the **start (left)** and **end (right)** margins. Remove the other margins.
2. For **Location**, use @dimen/margin_wide for the **start**, **end**, **top**, and **bottom** margins. Remove the other margins.
3. For **Rating**, use @dimen/margin_wide for the **start** and **end** margins. Remove the other margins.
4. To see how the text views in your app adapt to device-configuration changes, change the orientation of the preview. To do this, click the **Orientation for Preview (O)** icon in the toolbar and select **Landscape**.



5. Run the app. You should see five styled TextView views. To see how the constraints behave on a wider screen, try running the app on a larger device or emulator, such as a Nexus 10.





1.5.5 Add clickHandlers to the text views

In this task, you make the FunWithColors app a little more colorful. First, you change the color of all the text views to white. Then you add a click handler that changes the view's color and the layout background color when the user taps it.

1. In styles.xml, inside the greenBox style, change the background color to white. The text views will start out white with white font, then change colors when the user taps them.

```
<item name="android:background">@android:color/white</item>
```

2. In MainActivity.kt, after the onCreate() function, add a function called makeColored() to start your fun journey. Use View as the function's parameter. This view is the one whose color will change.

```
private fun makeColored(view: View) {
}
```

3. Implement the makeColored() function: Add a when block (similar to *switch* in C/Java) to check the view's resource ID. Call the setBackgroundColor() function on each view's id to change the view's background color using the Color class constants. To fix the code indentation, choose **Code > Reformat code**.

```
private fun makeColored(view: View) {
    when (view.id) {
       // Boxes using Color class colors for the background
       R.id.title_text -> view.setBackgroundColor(Color.DKGRAY)
       R.id.thumbnail_text -> view.setBackgroundColor(Color.GRAY)
       R.id.name_text -> view.setBackgroundColor(Color.BLUE)
       R.id.location_text -> view.setBackgroundColor(Color.MAGENTA)
       R.id.rating_text -> view.setBackgroundColor(Color.BLUE)
   }
}
```

4. To run, the code that you just added needs the android.graphics.Color library. If Android Studio hasn't imported this library automatically, use an import statement to add the library before the MainActivity class definition.

5. If the user taps the background, you want the background color to change to light gray. A light background will reveal the outlines of the views and give the user a hint about where to tap next.

   If the id doesn't match any of the views, you know that the user has tapped the background. At the end of the when statement, add an else statement. Inside the else, set the background color to light gray.

```
else -> view.setBackgroundColor(Color.LTGRAY)
```

6. In activity_main.xml, add an id to the root constraint layout. The Android system needs an identifier in order to change its color.

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

7. In MainActivity.kt, add a function called setListeners() to set the click-listener function, makeColored(), on each view. Use findViewByID to get a reference for each text view, and for the root layout. Assign each reference to a variable.

```
private fun setListeners() {
    val titleText = findViewById<TextView>(R.id.title_text)
    val thumbnailText = findViewById<TextView>(R.id.thumbnail_text)
    val nameText = findViewById<TextView>(R.id.name_text)
    val locationText = findViewById<TextView>(R.id.location_text)
    val ratingText = findViewById<TextView>(R.id.rating_text)
    val rootConstraintLayout = findViewById<View>(R.id.constraint_layout)
}
```

For this code to run, it needs the android.widget.TextView library. If Android Studio doesn't import this library automatically, use an import statement to add the library before the MainActivity class definition.

8. At the end of setListeners() function, define a List of views. Name the list clickableViews and add all the view instances to the list.

```
fun setListeners() {
...
   val clickableViews: List<View> =
       listOf(titleText, thumbnailText, nameText, locationText, ratingText,
rootConstraintLayout)
}
```

9. At the end of the setListeners() function, set the listener for each view. Use a for loop and the setOnClickListener() function.

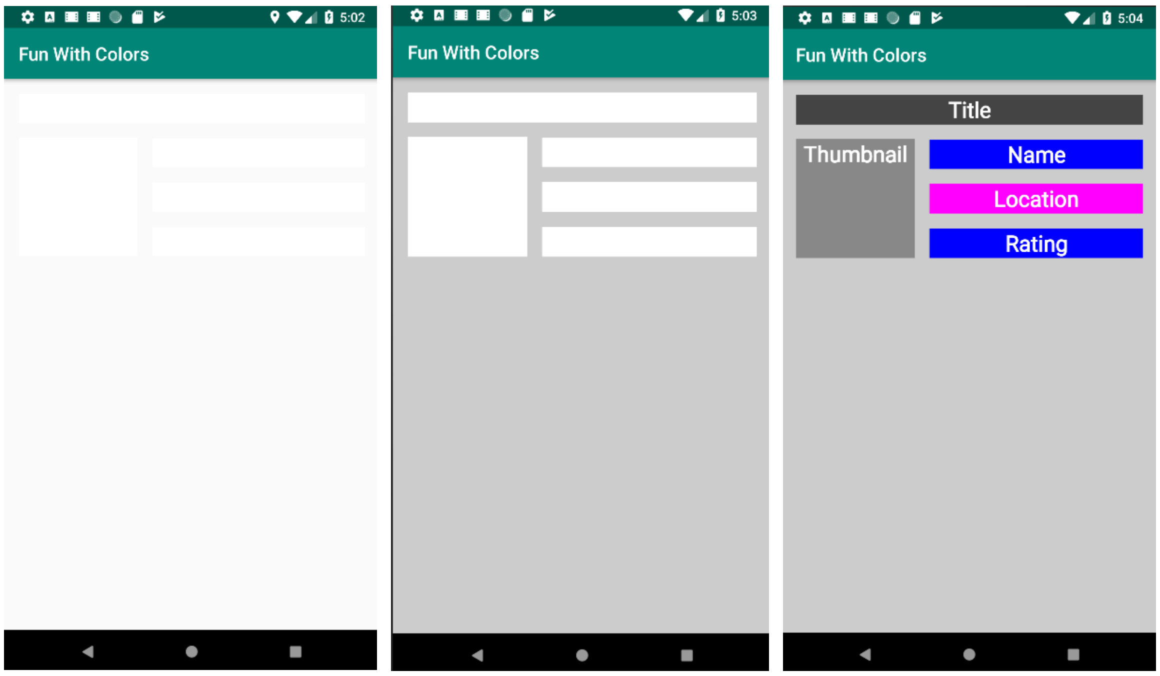
```
   for (item in clickableViews) {
        item.setOnClickListener {
            makeColored(it)
        }
    }
```

“it” keyword refers to the implicit name of a single parameter of a lamda expression.

10. In MainActivity.kt, at the end of the onCreate() function, make a call to setListeners().

```
override fun onCreate(savedInstanceState: Bundle?) {
...
   setListeners()
}
```

11. Run your app. At first, you see a blank screen. Tap the screen to reveal the boxes and the background. Go ahead and experiment more with more views and colors on your own.



1.5.6 (Optional Challenge) Go Further with Images

Use images instead of colors and text as backgrounds for the thumbnail view. The app can reveal the images when the user taps the thumbnail text view.

Hint: Add images to the app as drawable resources. Use the setBackgroundResource() function to set an image as a view's background.

Example:

```
R.id.thumbnail_text -> view.setBackgroundResource(R.drawable.image_cuisine)
```

2.1 Add an EditText for text input

In this task, you add input to let users input short comments. You create one EditText and one Button, one for input and one for confirm input. The button and EditText view have different font sizes, and you align their baselines. You also add one TextView view, to display the user input once it is done.

2.1.1: Add a style

3. Open res/values/dimens.xml, and add the following dimension resource for the font size.

```
<dimen name="input_text_size">18sp</dimen>
```

4. Open res/values/styles.xml and add the following style, which you will use for the text view.

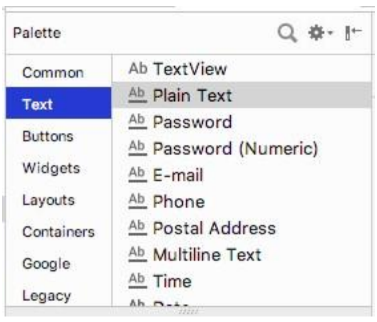
```
<style name="inputTextBox">
   <item name="android:background">@android:color/white</item>
   <item name="android:textAlignment">viewStart</item>
   <item name="android:textSize">@dimen/input_text_size </item>
   <item name="android:textStyle">normal</item>
   <item name="android:textColor">@android:color/black</item>
   <item name="android:fontFamily">@font/roboto</item>
</style>
```

In this style, the background color and the text color are set to default Android color resources. The font is set to Roboto. The text is left aligned, and the text size is set to input_text_size.

You can revert the style to green box in the following sections.

2.1.2: Add an EditText

1. In Android Studio, open the activity_main.xml layout file in the **Design** tab.
2. In the **Palette** pane, click **Text**.



Ab TextView, which is a TextView, shows at the top of the list of text elements in the **Palette** pane. Below **Ab TextView** are multiple EditText views.

In the **Palette** pane, notice how the icon for TextView shows the letters **Ab** with no underscoring. The EditText icons, however, show **Ab** underscored. The underscoring indicates that the view is editable.

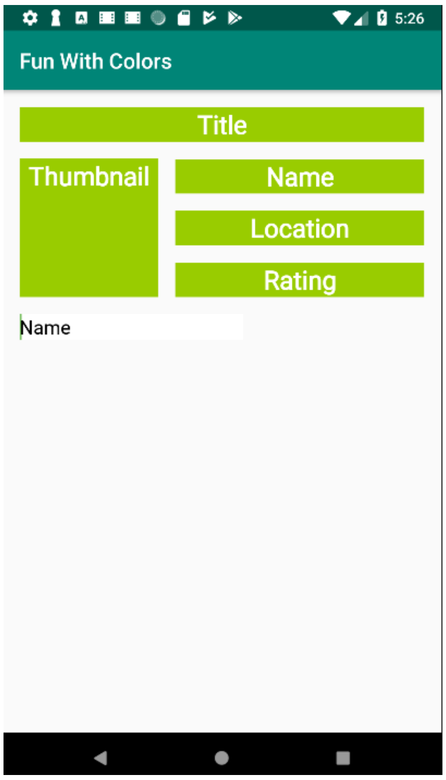
For each of the EditText views, Android sets different attributes, and the system displays the appropriate soft input method (such as an on-screen keyboard).

3. Drag a **PlainText** edit text into the **Component Tree** and place it to the bottom of the thumbnail text box.

4. Use the **Attributes** pane to set the following attributes on the EditText view.

Attribute	**Value**
id	comment_edit
layout_width	match_parent *(default)*
layout_height	wrap_content *(default)*

5. Run your app. Above the star image, you see an edit text with default text "Name".



2.1.2: Add hint text

In this task, you style your EditText view by adding a hint, changing the text alignment, changing the style to the inputTextBox, and setting the input type.

1. Add a new string resource for the hint in the string.xml file.

```
<string name="any_comment">Any Comment?</string>
```

Tip: It's a good practice to show a hint in each EditText view to help users figure out what input is expected for editable fields.

2. Use the **Attributes** pane to set the following attributes to the EditText view:

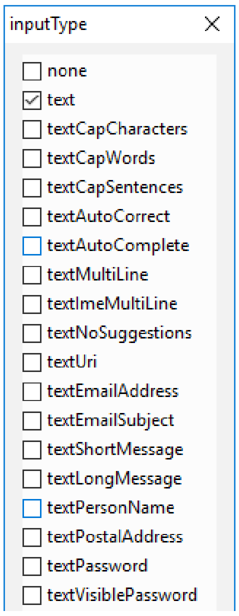
Attribute	**Value**
style	inputTextBox
textAlignment	*(viewStart)*
hint	@string/any_comment

3. In the **Attributes** pane, remove the Name value from the text attribute. The text attribute value needs to be empty so that the hint is displayed.

2.1.3: Set the inputType attribute

The inputType attribute specifies the type of input users can enter in the EditText view. The Android system displays the appropriate input field and on-screen keyboard, depending on the input type set.

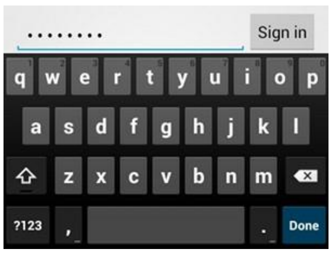
To see all the possible input types, in the **Attributes** pane, click the inputType field, or click the flag next to the field. A list opens that shows all the types of input you can use, with the currently active input type-checked. You can select more than one input type.



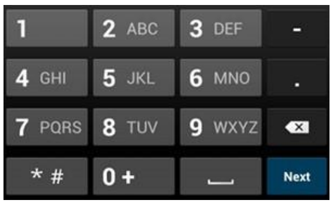
inputType

- ☐ none
- ☑ text
- ☐ textCapCharacters
- ☐ textCapWords
- ☐ textCapSentences
- ☐ textAutoCorrect
- ☐ textAutoComplete
- ☐ textMultiLine
- ☐ textImeMultiLine
- ☐ textNoSuggestions
- ☐ textUri
- ☐ textEmailAddress
- ☐ textEmailSubject
- ☐ textShortMessage
- ☐ textLongMessage
- ☐ textPersonName
- ☐ textPostalAddress
- ☐ textPassword
- ☐ textVisiblePassword

For example, for passwords, use the textPassword value. The text field hides the user's input.



For phone numbers, use the phone value. A number keypad is displayed, and the user can enter only numbers.



Set the input type for short comments field:

1. Set the inputType attribute to text for comment_edit edit text**.**
2. In the **Component Tree** pane, notice an autoFillHints warning. This warning does not apply to this app and is beyond the scope of this lab, so you can ignore it. (If you want to learn more about autofill, see Optimize your app for autofill.)
3. In the **Attributes** pane, verify the values for the following attributes of the EditText view:

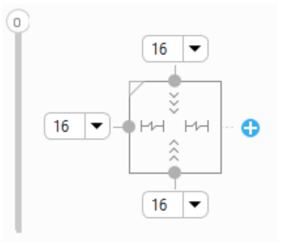
Attribute	**Value**
id	comment_edit
layout_width	match_parent *(default)*

layout_height	wrap_content *(default)*
style	@style/inputTextBox
inputType	text
hint	"@string/any_comment"
text	*(empty)*

2.1.4: Add vertical constraints to the edit text view

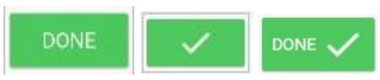
Without vertical constraints, views go to the top of the screen (vertical 0) at runtime. Adding vertical constraints will keep the two text views in place when you run the app.

1. Constrain the left of the comment_edit to the start of the layout.
2. Constrain the bottom of the comment_edit to the bottom of the layout.
3. Attach the top of the comment_edit to the bottom of the thumbnail.
4. In the view inspector, change the vertical bias of the comment_edit view to 0. This keeps the text views closer to the top constrained view, **Thumbnail**. (If the view inspector isn't visible in the **Attributes** pane when you click on the view in the design editor, close and reopen Android Studio.)



2.2: Add a DONE button

In this task, you add a **DONE** button, which the user taps after they enter a comment. The button swaps the EditText view with a TextView view that displays short comments. To update short comments, the user can tap the TextView view.



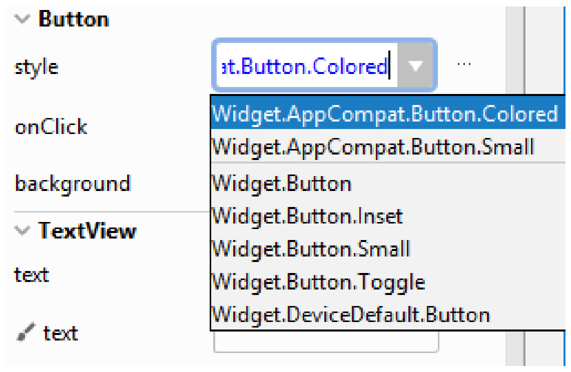
1. Drag a button from the **Palette** pane into the **Component Tree**. Place the button next to the comment_edit edit text.

2. Create a new string resource named done. Give the string value of Done,

```
<string name="done">Done</string>
```

3. Use the **Attributes** pane to set the following attributes on the newly added Button view:

Attribute	Values
id	done_button
text	@string/done
layout_height	wrap_content
layout_width	wrap_content



4. Change the style to Widget.AppCompat.Button.Colored, which is one of the predefined styles that Android provides. You can select the style from either the drop-down or from the **Resources** window.
   This style changes the button color to the accent color, colorAccent. The accent color is defined in the res/values/colors.xml file.

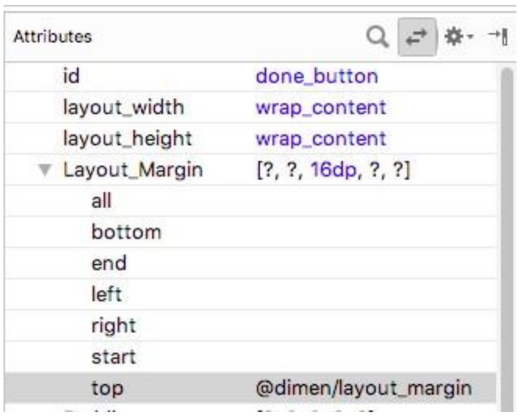
The colors.xml file contains the default colors for your app. You can add new color resources or change the existing color resources in your project, based on your app's requirements.

Sample colors.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="colorPrimary">#008577</color>
   <color name="colorPrimaryDark">#00574B</color>
   <color name="colorAccent">#D81B60</color>
</resources>
```

2.2.1: Style the DONE button

1. In the **Attributes** pane, add a top margin by selecting **Layout_Margin > Top**. Set the top margin to layout_margin, which is defined in the dimens.xml file.



Attributes	
id	done_button
layout_width	wrap_content
layout_height	wrap_content
Layout_Margin	[?, ?, 16dp, ?, ?]
all	
bottom	
end	
left	
right	
start	
top	@dimen/layout_margin

2. Set the fontFamily attribute to roboto from the drop-down menu.
3. Switch to the **Text** tab and verify the generated XML code for the newly added button.

```
<Button
    android:id="@+id/done_button"
    style="@style/Widget.AppCompat.Button.Colored"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:fontFamily="@font/roboto"
    android:text="@string/done" />
```

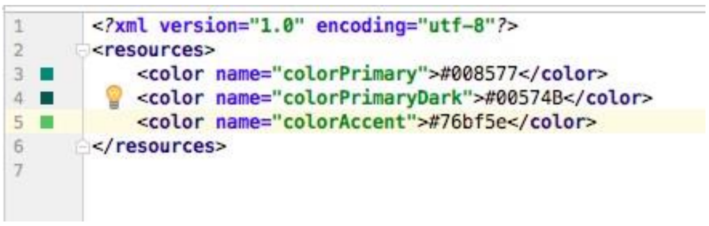
2.2.2: Change the color resource

In this step, you change the button's accent color to match your activity's app bar.

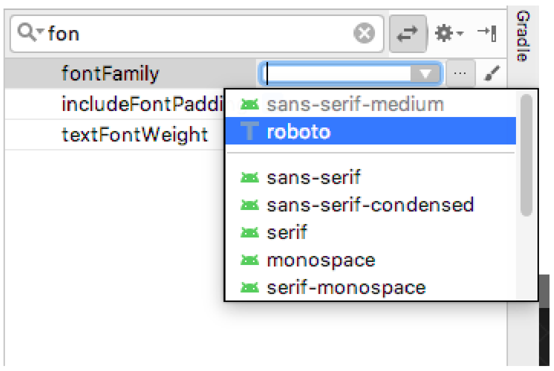
1. Open res/values/colors.xml and change the value of the colorAccent to #76bf5e.

```
<color name="colorAccent">#76bf5e</color>
```

You can see the color corresponding to the HEX code, in the left margin of the file editor.



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#008577</color>
    <color name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#76bf5e</color>
</resources>
```

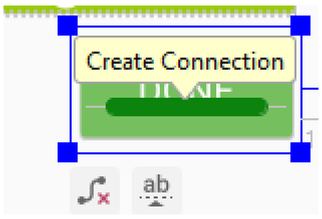


Notice the change in the button color in the design editor.

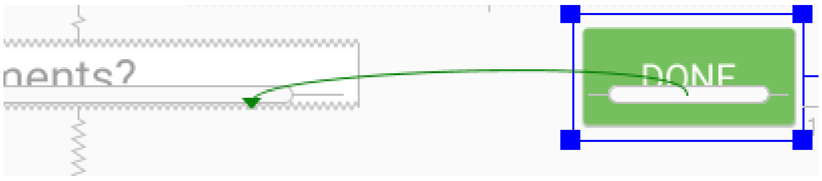
2.2.3: Align the baselines of the edit text view and button

1. Click the button. Right click and select **Show Baseline** then the **Baseline** appears below the view.

2. Hold the pointer over the label text view until the green baseline blinks, as shown below.

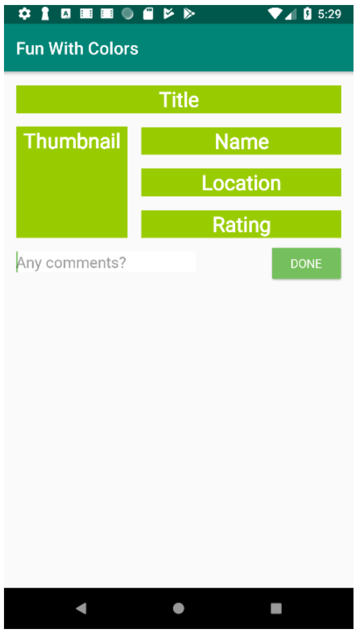


3. Click the green baseline and drag it. Connect the baseline to the bottom of the green blinking baseline on the edit text view.



4. Constrain the right of the done_button to the end of the layout.

Run your app. Your screen should look like the screenshot below. You should see a styled **DONE** button below the edit text.



2.3: Add a TextView to display a comment

In this task, after the user enters short comments and taps the **DONE** button, short comments display in a TextView view.

1. Drag a text view from the **Palette** pane into the **Component Tree**. Place the text view to the top of the done_button.
2. Use the **Attributes** pane to set the following attributes for the new TextView view:

Attribute	Value
id	comment_text
style	inputTextBox
layout_width	match_parent *(default)*
layout_height	wrap_content *(default)*
text	*(empty)*

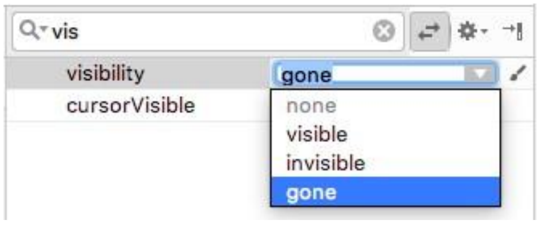
3. Constrain the left of the comment_text to the start of the layout.
4. Attach the top of the comment_text to the bottom of the thumbnail.
5. Constrain the bottom of the comment_text to the bottom of the layout.
6. In the view inspector, change the vertical bias of the comment_text view to 0.

2.3.2: Change the visibility of the TextView

You can show or hide views in your app using the visibility attribute. This attribute takes one of three values:

- visible: The view is visible.
- Invisible: Hides the view, but the view still takes up space in the layout.
- gone: Hides the view, and the view does not take up any space in the layout.

1. In the **Attributes** pane, set the visibility of comment_text text view to gone**,** because you don't want your app to show this text view at first.



   Notice that as you change the attribute in the **Attributes** pane, comment_text view disappears from the design editor. The view is hidden in the layout preview.

2. Change the text attribute value of comment_text view to an empty string.

Your generated XML code for this TextView should look similar to this:

```
<TextView
    android:id="@+id/comment_text"
    style="@style/inputTextBox"
    android:layout_width="match_parent"
   android:layout_height="wrap_content"
    android:textAlignment=" viewStart"
    android:visibility="gone"
    android:text="" />
```

Your layout preview should look something like the following:

A click handler on the Button object (or on any view) specifies the action to be performed when the button (view) is tapped. The function that handles the click event should be implemented in the Activity that hosts the layout with the button (view).

The click listener has this format, where the passed in view is the view that received the click or tap.

```
private fun clickHandlerFunction(viewThatIsClicked: View) {
   // Add code to perform the button click event
}
```

You can attach the click-listener function to button click events two ways:

- In the XML layout, you can add the android:onClick attribute to the <Button> element. For example:

```
<Button     android:id="@+id/done_button"
    android:text="@string/done"
   ...
   android:onClick="clickHandlerFunction"/>
```

OR

- You can do it programmatically at runtime, in onCreate() of the Activity, by calling setOnClickListener. For example:

```
myButton.setOnClickListener {
    clickHanderFunction(it)
}
```

In this task, you add a click listener for the done_button programmatically. You add the click listener in the corresponding activity, which is MainActivity.kt.

Your click-listener function, called addComment, will do the following:

- Get the text from comment_edit edit text.
- Set the text in the comment_text text view.
- Hide the edit text and the button.
- Display a short comment in comment_text.

2.3.3: Add a click listener

1. In MainActivity.kt, inside the MainActivity class, add a function called addComment. Include an input parameter called view of type View. The view parameter is the View on which the function is called. In this case, view will be an instance of your **DONE** button.

```
private fun addComment(view: View) {
}
```

2. Inside the addComment function, use findViewById() to get a reference to comment_edit edit text and comment_text text view.

```
val editText = findViewById<EditText>(R.id.comment_edit)
val commentTextView = findViewById<TextView>(R.id.comment_text)
```

3. Set the text in commentTextView text view to the text that the user entered in the editText, getting it from the text property.

```
commentTextView.text = editText.text
```

4. Hide short comments editText view by setting the visibility property of editText to View.GONE.

In a previous task, you changed the visibility property using the Layout Editor. Here you do the same thing programmatically.

```
editText.visibility = View.GONE
```

5. Hide the **DONE** button by setting the visibility property to View.GONE. You already have the button's reference as the function's input parameter, view.

```
view.visibility = View.GONE
```

6. At the end of the addComment function, make short comments TextView view visible by setting its visibility property to View.VISIBLE.

```
commentTextView.visibility = View.VISIBLE
```

2.3.4: Attach the click listener to the DONE Button

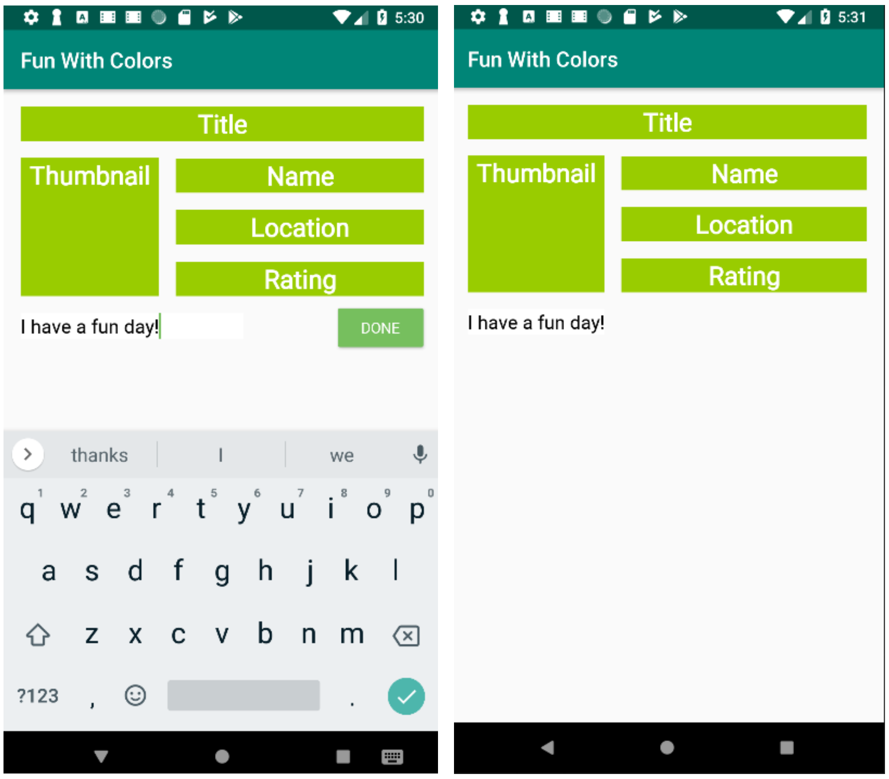
Now that you have a function that defines the action to be performed when the **DONE** button is tapped, you need to attach the function to the Button view.

1. In MainActivity.kt, at the end of the onCreate() function, get a reference to the **DONE** Button view. Use the findViewById() function and call setOnClickListener. Pass in a reference to the click-listener function, addComment().

```
findViewById<Button>(R.id.done_button).setOnClickListener {
             addComment(it)
}
```

In the above code, it refers to the done_button, which is the view passed as the argument.

2. Run your app, enter a comment, and tap the **DONE** button. Notice how the edit text and the button are replaced by a short comment text view.



Notice that even after the user taps the **DONE** button, the keyboard is still visible. This behavior is the default.

2.3.5: Hide the keyboard

In this step, you add code to hide the keyboard after the user taps the **DONE** button.

1. In MainActivity.kt, at the end of addComment() function, add the following code. If you'd like more information on how this code works, see the hideSoftInputFromWindow documentation.

```
// Hide the keyboard.
val inputMethodManager = getSystemService(Context.INPUT_METHOD_SERVICE) as
InputMethodManager
inputMethodManager.hideSoftInputFromWindow(view.windowToken, 0)
```

2. Run your app again. Notice that after you tap **DONE,** the keyboard is hidden.

There's no way for the user to change a short comment after they tap the **DONE** button. So, you add extra interactivity to the text view next.

2.4: Add a click listener to the comment TextView

In this task, you add a click listener to the short comments text view. The click listener hides short comments text view, shows the edit text, and shows the **DONE** button.

2.4.1: Add a click listener

1. In MainActivity, add a click-listener function called updateComment(view: View) for short comments text view.

```
private fun updateComment (view: View) {
}
```

2. Inside the updateComment function, get a reference to comment_edit edit text, and get a reference to the **DONE** button. To do this, use the findViewById() method.

```
val editText = findViewById<EditText>(R.id.comment_edit)
val doneButton = findViewById<Button>(R.id.done_button)
```

3. At the end of the updateComment function, add code to show the edit text, show the **DONE** button, and hide the text view.

```
editText.visibility = View.VISIBLE
doneButton.visibility = View.VISIBLE
view.visibility = View.GONE
```

4. In MainActivity.kt, at the end of the onCreate() function, call setOnClickListener on comment_text text view. Pass in a reference to the clicklistener function, which is updateComment().

```
findViewById<TextView>(R.id.comment_text).setOnClickListener {
    updateComment(it)
}
```

5. Run your app. Enter a comment, tap the **DONE** button, then tap short comments comment_text view. The comment view disappears, and the edit text and the **DONE** button become visible.

Notice that by default, the EditText view does not have focus and the keyboard is not visible. It's difficult for the user to figure out that the short comments text view is clickable. In the next task, you add focus and a style to short comments text view.

2.4.2: Set the focus to the EditText view and show the keyboard

1. At the end of the updateComment function, set the focus to the EditText view. Use the requestFocus() method.

```
// Set the focus to the edit text.
editText.requestFocus()
```

2. At the end of the updateComment function, add code to make the keyboard visible.

```
// Show the keyboard.
```

```
val imm = getSystemService(Context.INPUT_METHOD_SERVICE) as
InputMethodManager
imm.showSoftInput(editText, 0)
```

References

- Build a Responsive UI with ConstraintLayout
- ConstraintLayout
- Build a UI with Layout Editor
- Understanding the performance benefits of ConstraintLayout

Videos:

- ConstraintLayout Deep Dive (Android Dev Summit '18)