Android Kotlin Fundamentals: LinearLayout using the Layout Editor

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Introduction

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

What you should already know

- Creating a basic Android app in Kotlin.
- Running an Android app on an emulator or on a device.
- The basics of `LinearLayout`.
- Creating a simple app that uses `LinearLayout` and a `TextView`.

What you'll learn

How to work with `View` and `ViewGroup`.

How to arrange views in an `Activity`, using `LinearLayout`.

How to use `ScrollView` for displaying the scrollable content.

How to change the visibility of a `View`.

How to create and use string and dimension resources.

How to create a LinearLayout using Android Studio's Layout Editor.

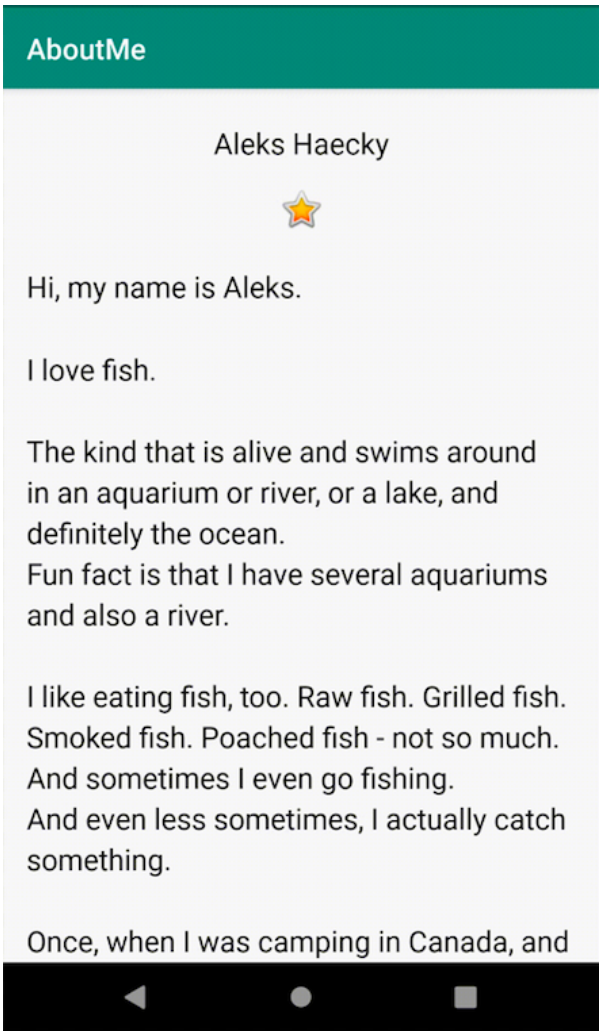
What you'll do

- Create the AboutMe app.
- Add a `TextView` to the layout to display your name.
- Add an `ImageView`.

- Add a `ScrollView` to display scrollable text.

2. App overview

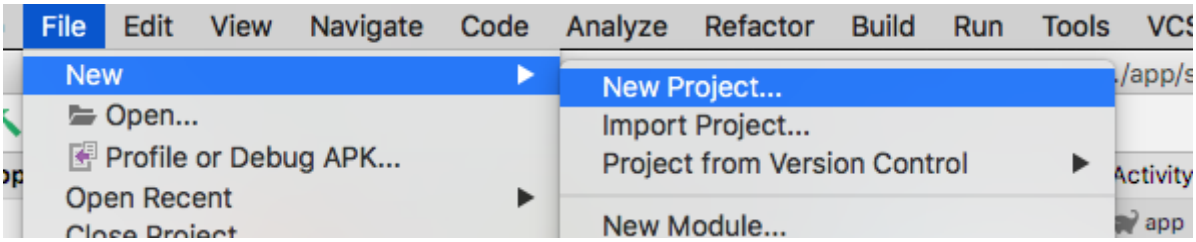
In the AboutMe app, you can showcase interesting facts about yourself, or you can customize the app for a friend, family member, or pet. The app displays a name, a **DONE** button, a star image, and some scrollable text.



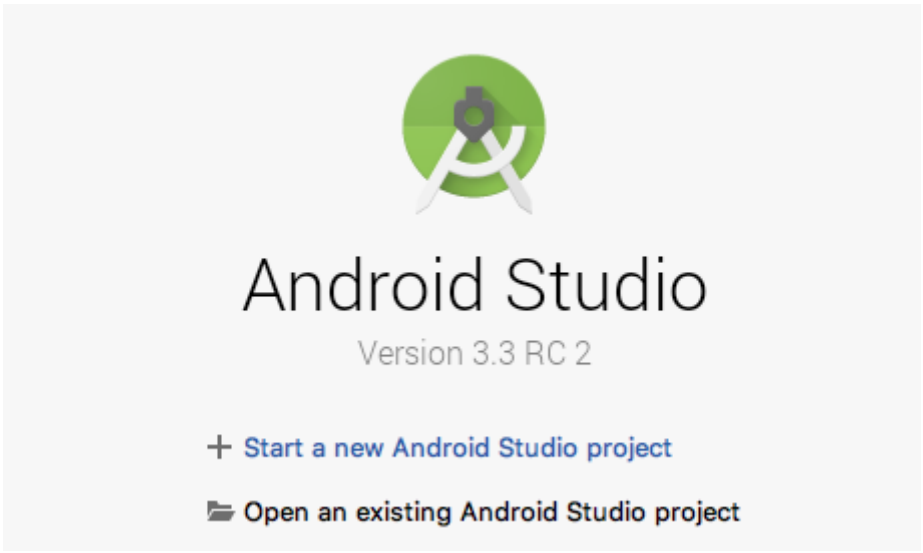
3. Task: Create the AboutMe Project

In this task, you create the AboutMe Android Studio project.

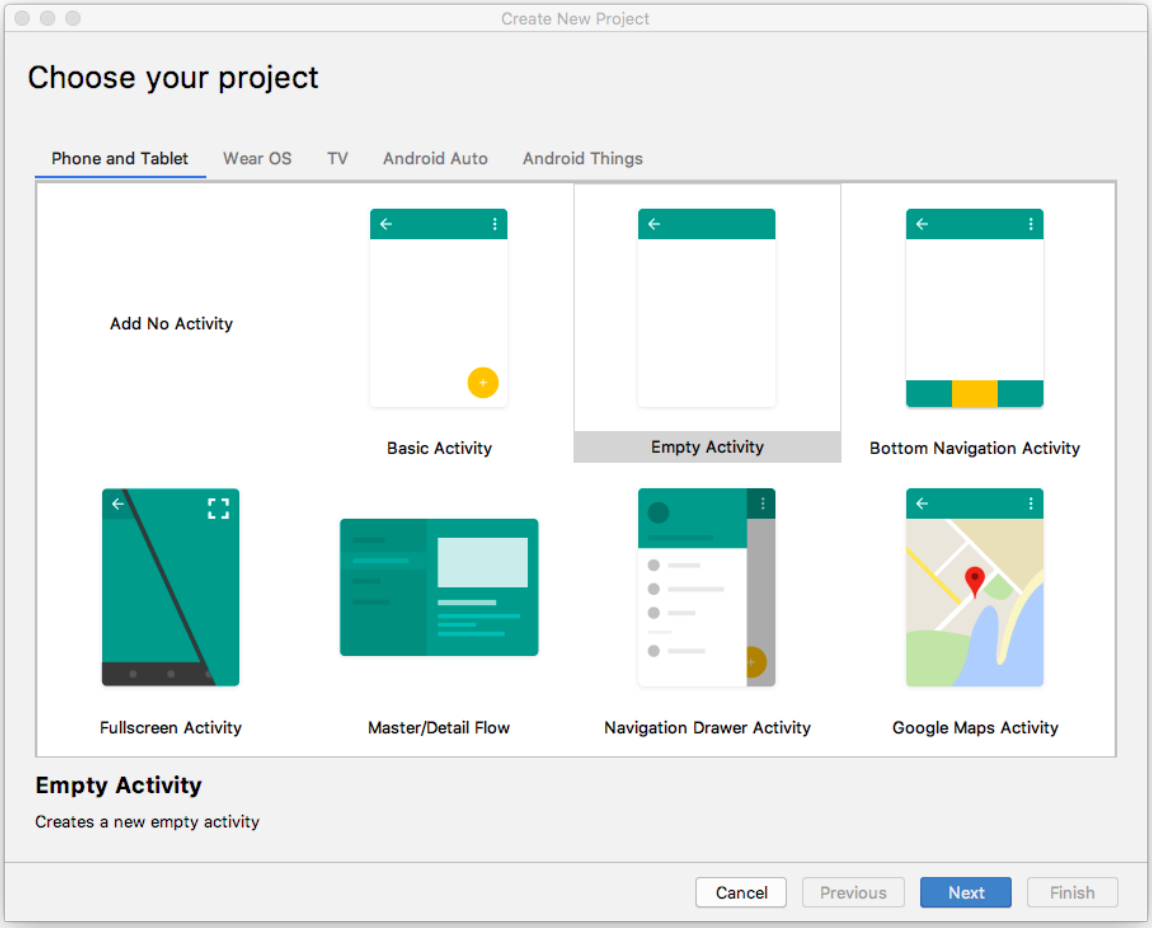
1. Open Android Studio, if it's not already open.
2. If a project is already open in Android Studio, select **File > New > New Project**.



3. If a project is not already open, select **+ Start a new Android Studio project** in the **Welcome to Android Studio** dialog.

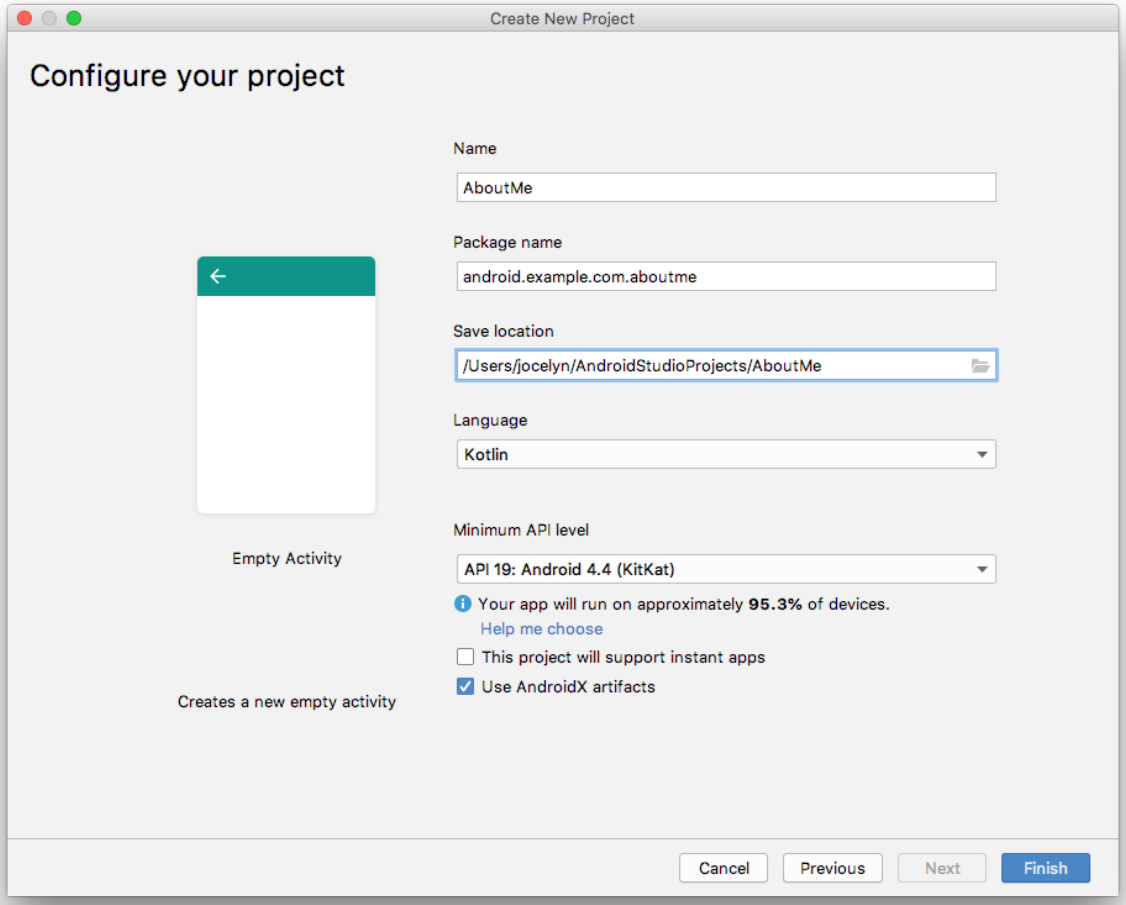


4. In the **Create New Project** dialog, in the **Phone and Tablet** tab, select the **Empty Activity** template. Click **Next**.



5. In the next **Create New Project** dialog, set the following parameters and click **Finish**.

Attribute	**Value**
Application Name	AboutMe
Company Name android	`com.android.example.AboutMe` *(or your own domain)*
Save location	*Leave the default location, or change it to your preferred directory.*
Language	Kotlin
Minimum API level	API 19: Android 4.4 (KitKat)
This project will support instant apps	*Leave this checkbox cleared.*
Use AndroidX artifacts	*Select this checkbox.*



Android Studio will take a moment to generate the project files.

6. Run your app. You will see the string "Hello World" on the blank screen.

The Empty Activity template creates a single empty activity, `Mainactivity.kt`. The template also creates a layout file called `activity_main.xml`. The layout file has `ConstraintLayout` as its root `ViewGroup`, and it has a single `TextView` as its content.

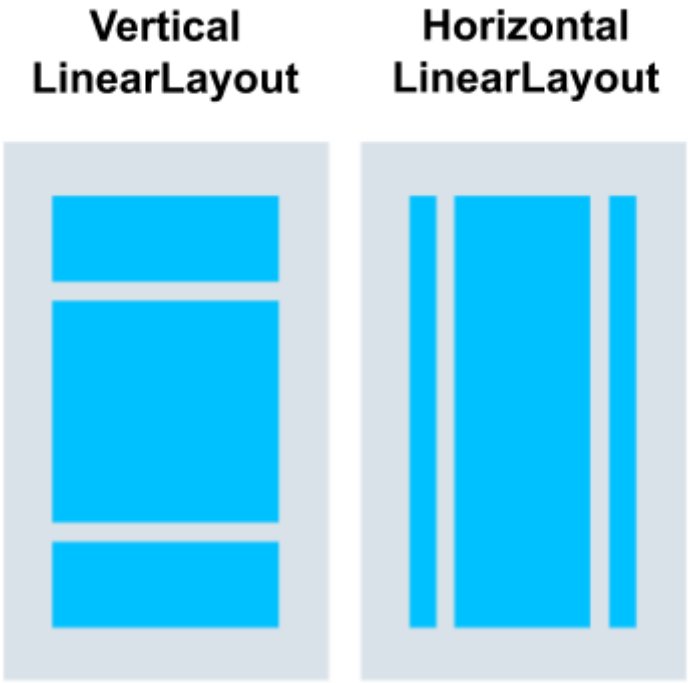
4. Task: Change the root layout to use LinearLayout

In this task, you change the generated root `ViewGroup` to a `LinearLayout`. You also arrange the UI elements vertically.

View groups

A `ViewGroup` is a view that can contain *child* views, which are other views and view groups. Views that make up a layout are organized as a hierarchy of views with a view group as the root.

In a `LinearLayout` view group, the UI elements are arranged either horizontally or vertically.



Change the root layout so that it uses a `LinearLayout` view group:

1. Select the **Project > Android** pane. In the `app/res/layout` folder, open the `activity_main.xml` file.
2. Select the **Text** tab and change the root view group from `ConstraintLayout` to `LinearLayout`.
3. Remove the `TextView`. In the `LinearLayout` element, add the `android:orientation` attribute and set it to `vertical`.

Before:

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.c
    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>
```

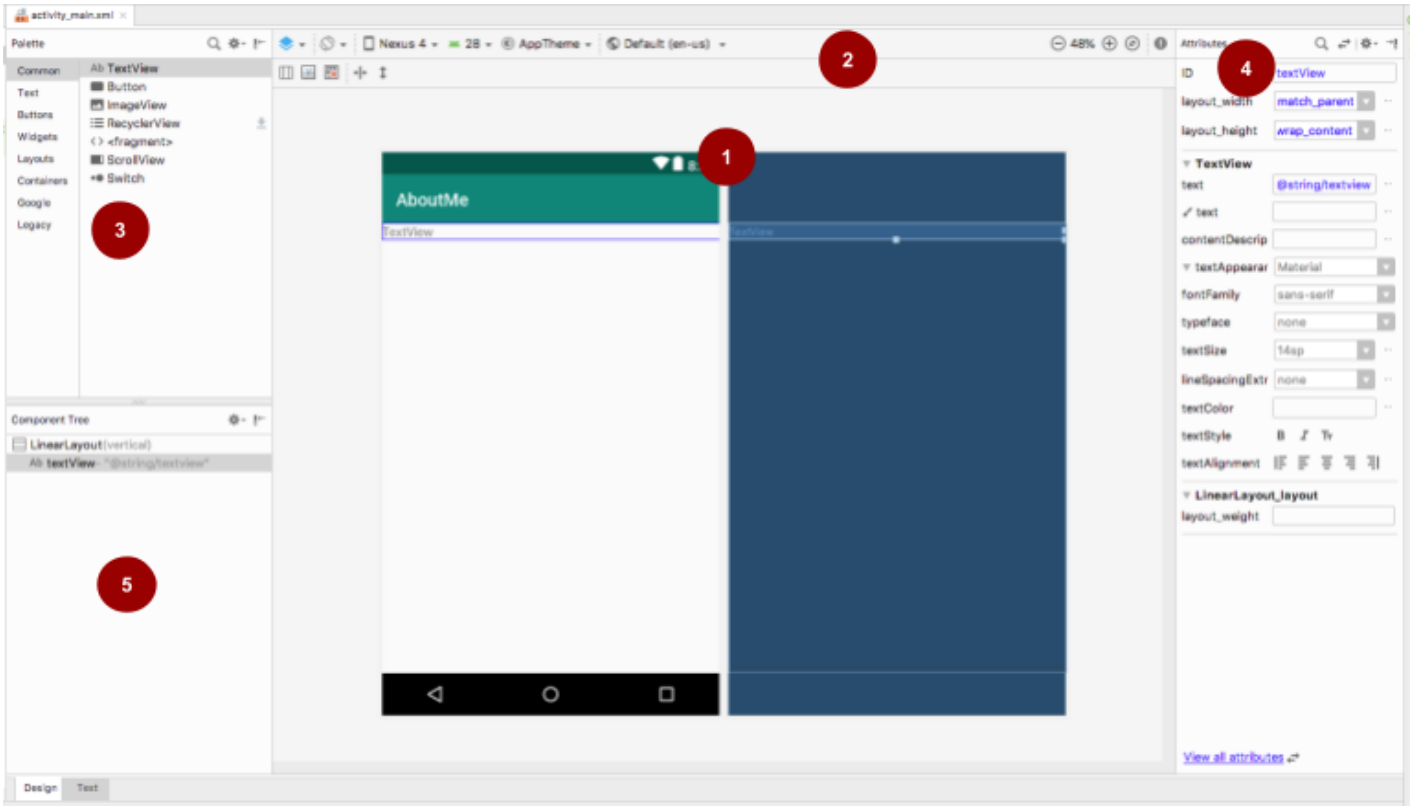
After:

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        tools:context=".MainActivity">

</LinearLayout>
```

5. Task: Add a TextView using Layout Editor

The Layout Editor is a visual-design tool inside Android Studio. Instead of writing XML code by hand to build your app's layout, you can use the Layout Editor to drag UI elements into the design editor.

To see the Layout Editor, click the **Design** tab. The screenshot below shows the parts of the Layout Editor.



1

Design editor: Displays a visual representation of your screen layout in design view, blueprint view, or both. The design editor is the main part of the Layout Editor.

2

Toolbar: Provides buttons to configure your layout's appearance in the design editor, and to change some layout attributes. For example, to change the display of your layout in the design editor, use the **Select Design Surface**

drop-down menu:

- Use **Design** for a real-world preview of your layout.
- Use **Blueprint** to see only outlines for each view.
- Use **Design + Blueprint** to see both displays side by side.

3

Palette: Provides a list of views and view groups that you can drag into your layout or into the **Component Tree** pane.

4

Attributes: Shows attributes for the currently selected view or view group. To toggle between a complete list of attributes and commonly used attributes, use the

icon at the top of the pane.

5

Component Tree: Displays the layout hierarchy as a tree of views. The **Component Tree** is useful when you have small, hidden, or overlapping views that you could not otherwise select in the design editor.

Step 1: Add a TextView

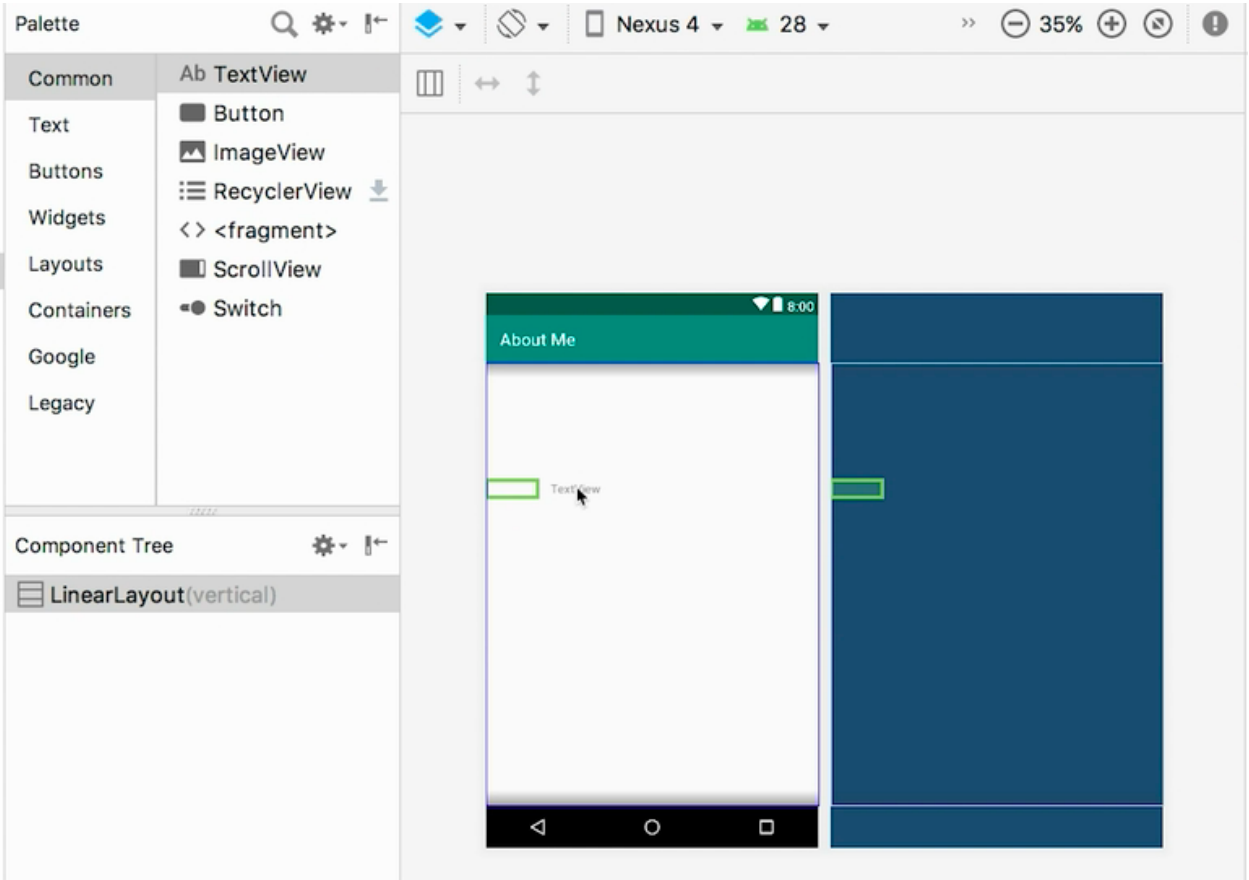
1. Open the `res/layout/activity_main.xml` file, if it's not already open.
2. Switch to the **Text** tab

and inspect the code. The code has a `LinearLayout` as its root view group. (*View groups* are views that contain other views.)

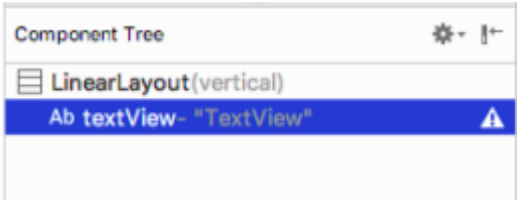
The `LinearLayout` has the required attributes `layout_height`, `layout_width`, and `orientation`, which is `vertical` by default. 3. Switch to the **Design** tab to open the Layout Editor.

> **Note:** The **Design** tab and the **Text** tab shows the same layout, just in a different way. Changes you make in one tab are reflected in the other.

4. Drag a text view from the **Palette** pane onto the design editor.



5. Notice the **Component Tree** pane. The new text view is placed as a child element of the parent view group, which is the `LinearLayout`.



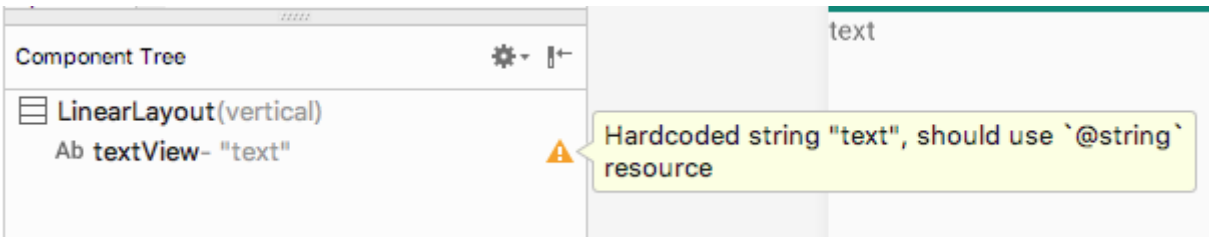
6. Open the **Attributes** pane, if it's not open already. (To open the pane, double-click the newly added `TextView` in the design editor.) 7. Set the following attributes in the **Attributes** pane:

Attribute	Value
ID	`name_text`
text	Set it to your name. (One of the **text** fields shows a wrench icon to indicate that it's for the `tools` namespace. The one without the wrench is for the `android` namespace—this is the **text** field you want.)
textAppearance > textSize	`20sp`
textAppearance > textColor	`@android:color/black`
textAppearance > textAlignment	*Center*

Step 2: Create a string resource

In the **Component Tree**, next to the `TextView`, you will notice a warning icon

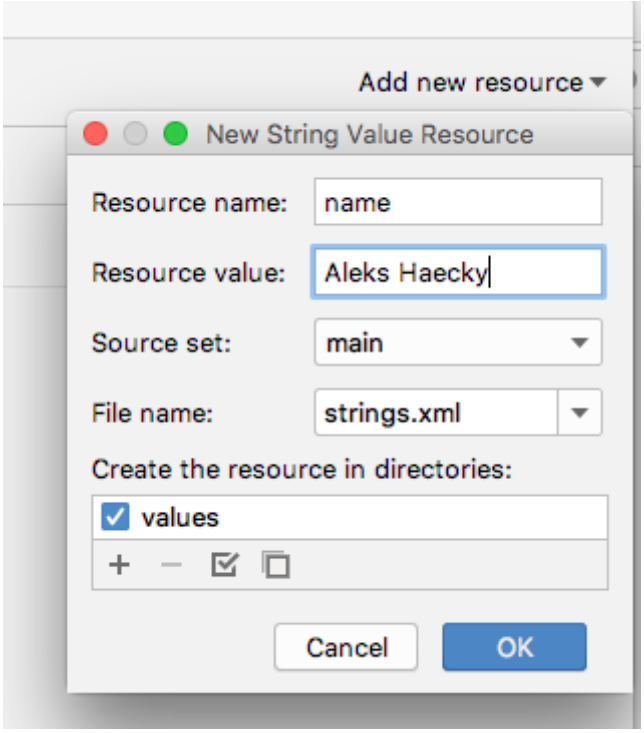
. To see the warning text, click the icon or point to it, as shown in the screenshot below.



To resolve the warning, create a string resource:

1. In the **Attributes** pane, click the three dots next to the **text** attribute that you set to your name. The resource editor opens.

2. In the **Resources** dialog, select **Add new resource > New string Value**.
3. In the **New String Value Resource** dialog, set the **Resource name** field to `name`. Set the **Resource value** field to your own name. Click **OK**. Notice that the warning is gone.



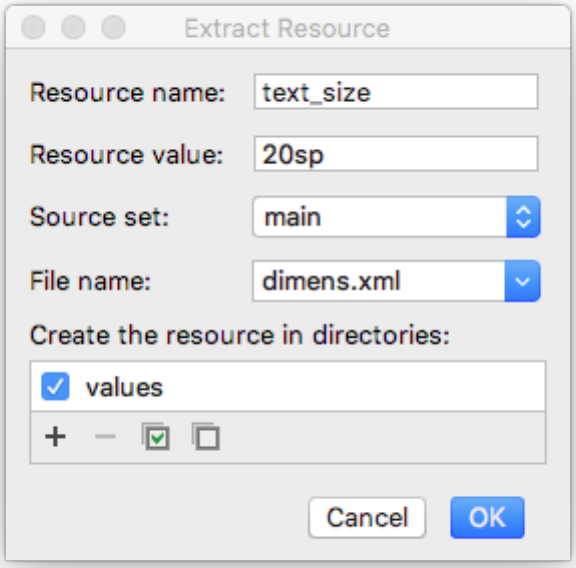
4. Open the `res/values/strings.xml` file and look for the newly created string resource called `name`.

```
<string name="name">Aleks Haecky</string>
```

Step 3: Create a dimension resource

You just added a resource using the resource editor. You can also extract resources in the XML code editor to create new resources:

1. In the `activity_main.xml` file, switch to the **Text** tab.
2. On the `textSize` line, click on the number (`20sp`) and type `Alt+Enter` (`Option+Enter` on a Mac). Select **Extract dimension resource** from the popup menu.
3. In the **Extract Resource** dialog, enter `text_size` in the **Resource name** field. Click **OK**.



4. Open the `res/values/dimens.xml` file to see the following generated code:

```
<dimen name="text_size">20sp</dimen>
```

> **Note:** If the **dimens.xml** file was not already present inside your **res/values** folder, Android Studio creates it.

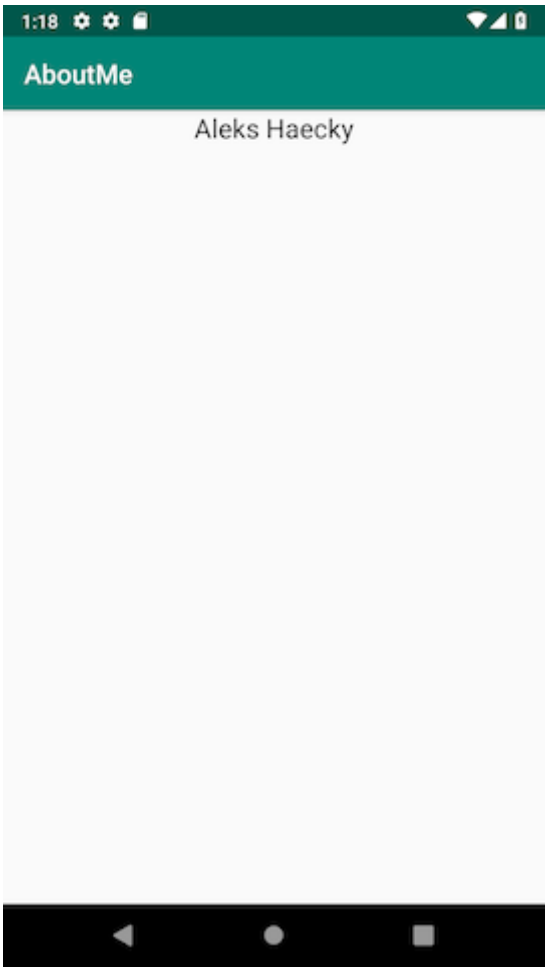
5. Open `MainActivity.kt` file, and look for the following code at the end of the `onCreate()` function:

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

The `setContentView()` function connects the layout file with the `Activity`. The specified layout resource file is `R.layout.activity_main`:

- `R` is a reference to the resource. It is an auto-generated class with definitions for all the resources in your app.
- `layout.activity_main` indicates that the resource is a layout named `activity_main`.

6. Run your app. A `TextView` with your name is displayed.

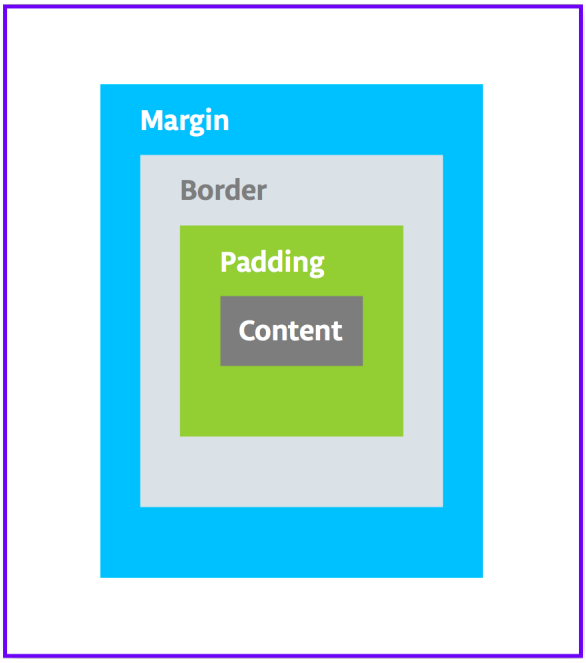


6. Task: Style your TextView

When you look at your app screen, your name is pushed up against the top of the screen, so now you add padding and a margin.

Padding versus margin

Padding is the space inside the boundaries of a view or element. It is the space between the edges of the view and the view's content, as shown in the figure below.



A view's size includes its padding. The following are commonly used padding attributes:

- `android:padding` specifies padding for all four edges of the view.
- `android:paddingTop` specifies padding for the top edge.
- `android:paddingBottom` specifies padding for the bottom edge.
- `android:paddingStart` specifies padding for the "starting" edge of the view.
- `android:paddingEnd` specifies padding for the "ending" edge of the view.
- `android:paddingLeft` specifies padding for the left edge.
- `android:paddingRight` specifies padding for the right edge.

Margin is the space added outside of the view's borders. It is the space from the edge of the view to its parent, as shown in the figure above. The following are commonly used margin attributes:

- `android:layout_margin` specifies a margin for all four sides of the view.
- `android:layout_marginTop` specifies extra space on the top side of this view
- `android:layout_marginBottom` specifies space outside the bottom side of this view.
- `android:layout_marginStart` specifies space outside the "starting" side of this view.
- `android:layout_marginEnd` specifies space on the end side of this view.
- `android:layout_marginLeft` specifies space on the left side of this view.
- `android:layout_marginRight` specifies space on the right side of this view.

> **Right/left versus start/end**
>
> "Right" and "left" always refer to the right and left sides of the screen, whether your app uses a left-to-right (LTR) flow or a right-to-left (RTL) flow. "Start" and "end" always refer to the start and end of the flow:

- For a LTR flow, start = left and end=right.
- For a RTL flow, start=right and end=left.

If your app targets API level 17 (Android 4.2) or higher:

- Use "start" and "end" instead of "left" and "right".
- For example, `android:layout_marginLeft` should become `android:layout_marginStart` to support RTL languages.

If you want your app to work with versions lower than Android 4.2; that is, if the app's `targetSdkVersion` or `minSdkVersion` is 16 or lower:

- Add "start" and end" in addition to "left" and "right".
- For example, use both `android:paddingLeft` and `android:paddingStart`.

Step 1: Add padding

To put space between your name and the top edge of the `name` text view, add top padding.

1. Open `activity_main.xml` file in the **Design** tab.
2. In the **Component Tree** or in the design editor, click the text view to open its **Attributes** pane.
3. At the top of the **Attributes** pane, click the double-arrow icon  to see all the available attributes.
4. Search for **Padding,** expand it, and click the three dots **...** next to the **top** attribute. The **Resources**dialog appears.
5. In the **Resources** dialog, select **Add new resource > New dimen Value**.
6. In the **New Dimension Value Resource** dialog, create a new `dimen` resource called `small_padding` with a value of `8dp` .

The *dp* abbreviation stands for *density-independent*. If you want a UI element to look the same size on screens with different densities, use dp as your unit of measurement. When specifying text size, however, always use sp (scalable pixels). 7. Click **OK**.

Step 2: Add a margin

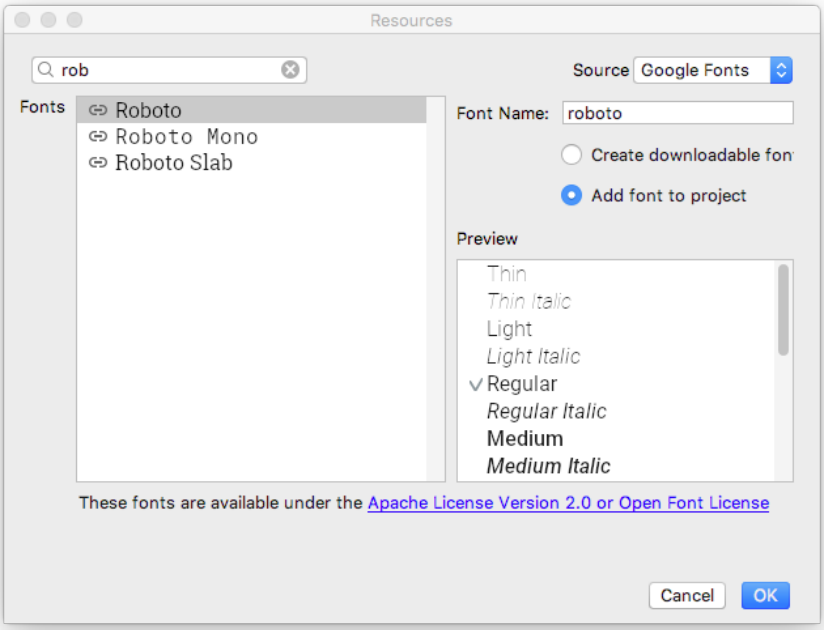
To move the `name` text view away from the edge of the parent element, add a top margin.

1. In the **Attributes** pane, search for "margin" to find **Layout_Margin**.
2. Expand **Layout_Margin**, and click the three dots **...** next to the **top** attribute.
3. Create a new `dimen` resource called `layout_margin` and make it `16dp` . Click **OK**.

Step 3: Add a font

To make the `name` text view look better, use the Android Roboto font. This font is part of the support library, and you add the font as a resource.

1. In the **Attributes** pane, search for "fontFamily".
2. In the **fontFamily** field, click the drop-down arrow**,** scroll to the bottom of the list, and select **More Fonts**.
3. In the **Resources** dialog, search for `rob` and choose **Roboto**. In the **Preview** list, select **Regular**.
4. Select the **Add font to project** radio button.
5. Click **OK**.



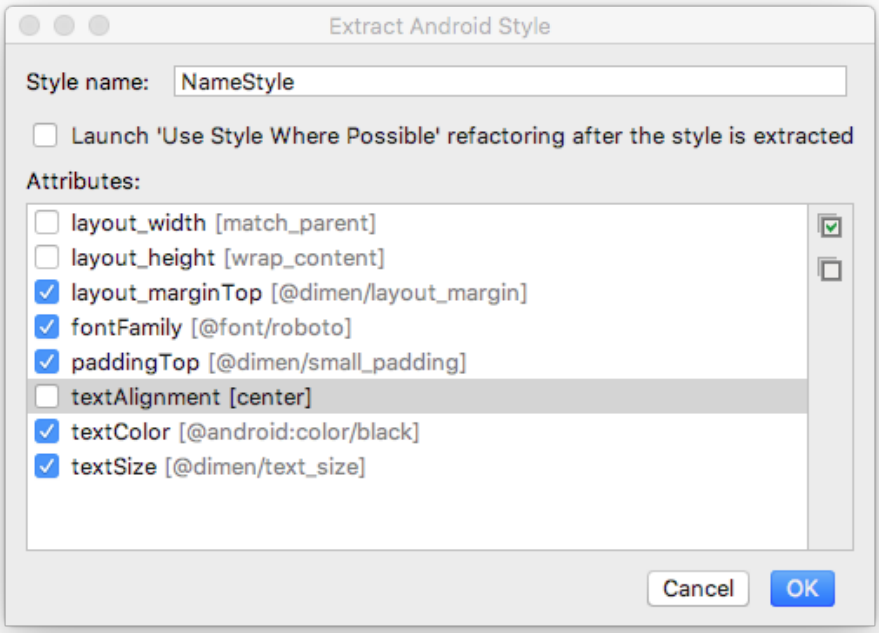
The `res` folder now has a `font` folder that contains a `roboto.ttf` font file. The `@font/roboto` attribute is added to your `TextView`.

Step 4: Extract the style

A *style* is a collection of attributes that specify the appearance and format for a view. A style can include font color, font size, background color, padding, margin, and other common attributes.

You can extract the `name` text view's formatting into a style and reuse the style for any number of views in your app. Reusing a style gives your app a consistent look when you have multiple views. Using styles also allows you to keep these common attributes in one location.

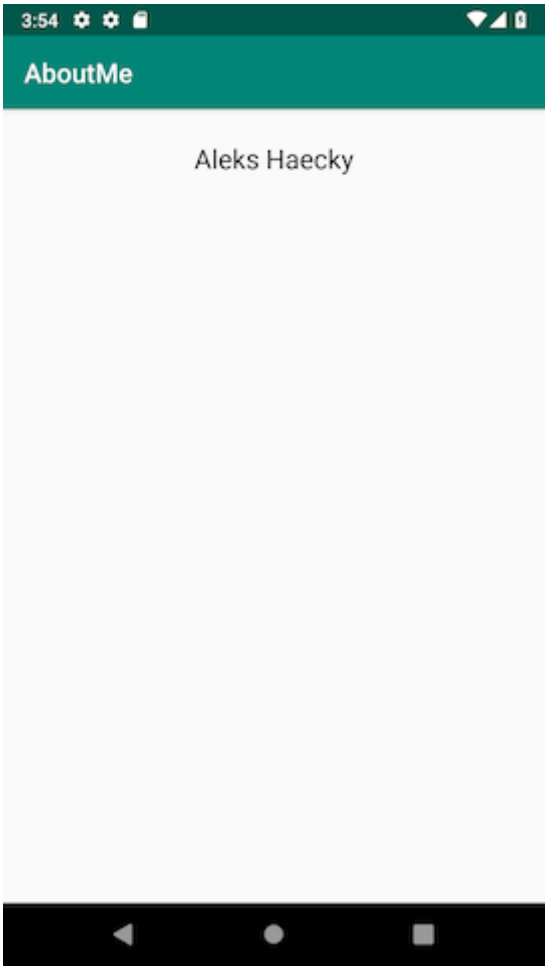
1. Right-click the `TextView` in the **Component Tree** and select **Refactor > Extract Style**.
2. In the **Extract Android Style** dialog, clear the `layout_width` checkbox, the `layout_height` checkbox, and the `textAlignment` checkbox. These attributes are usually different for each view, so you don't want them to be part of the style.
3. In the **Style name** field, enter `NameStyle`.
4. Click **OK**.



5. A style is also a resource, so the style is saved in the `res/values/` folder in a `styles.xml` file. Open `styles.xml` and examine the generated code for the `NameStyle` style, which will look similar to this:

```
<style name="NameStyle">
    <item name="android:layout_marginTop">@dimen/layout_margin</item>
    <item name="android:fontFamily">@font/roboto</item>
    <item name="android:paddingTop">@dimen/small_padding</item>
    <item name="android:textColor">@android:color/black</item>
    <item name="android:textSize">@dimen/text_size</item>
</style>
```

6. Open `activity_main.xml` and switch to the **Text** tab. Notice that the generated style is being used in the text view as `style="@style/NameStyle"` .
7. Run the app and notice the changes in the font and the padding around your `TextView` .


3:54
AboutMe
Aleks Haecky

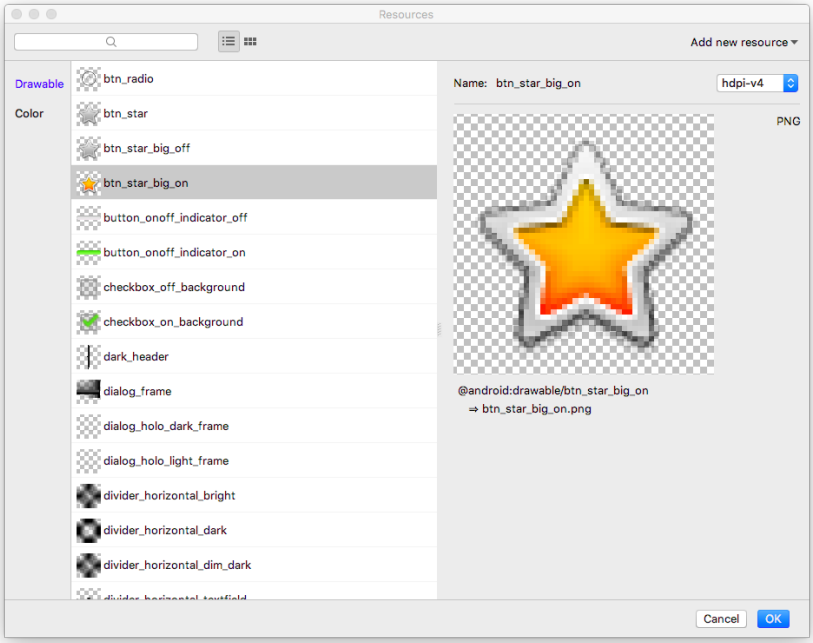
7. Task: Add an ImageView

Most real-world Android apps consist of a combination of views to display images, display text, and accept input from the user in the form of text or click events. In this task, you add a view to display an image.

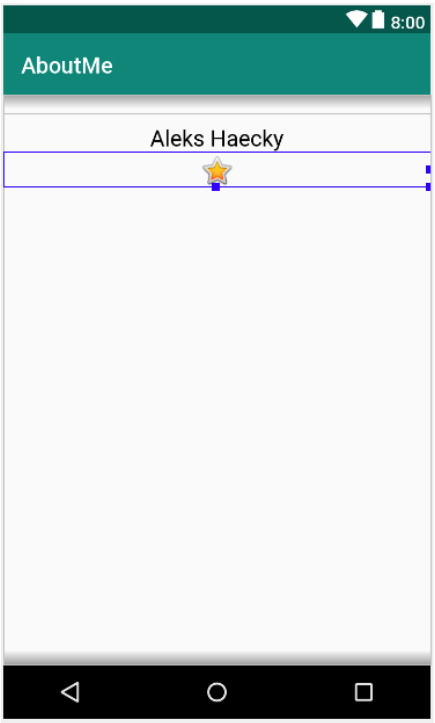
An `ImageView` is a view for displaying image resources. For example, an `ImageView` can display `Bitmap` resources such as PNG, JPG, GIF, or WebP files, or it can display a `Drawable` resource such as a vector drawing.

There are image resources that come with Android, such as sample icons, avatars, and backgrounds. You will add one of these resources to your app.

1. Display the layout file In the **Design** tab, then drag an **ImageView** from the **Palette** pane to below `name_text` in the **Component Tree.** The **Resources** dialog opens.
2. Select **Drawable** if it's not already selected.
3. Expand **android**, scroll, and select **btn_star_big_on**. It's the yellow star  .
4. Click **OK**.



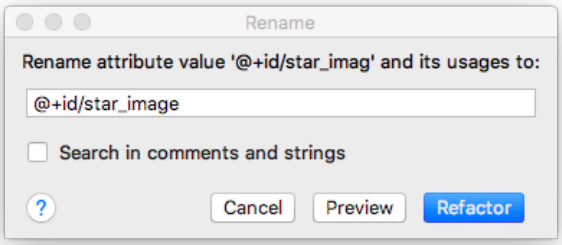
The star image is added to the layout below your name. Because you have a vertical `LinearLayout`, views you add are vertically aligned.



5. Switch to the **Text** tab and look at the generated `ImageView` code. The width is set to `match_parent`, so the view will be as wide as its parent element. The height is set to `wrap_content`, so the view will be as tall as its content. The `ImageView` references the `btn_star_big_on` drawable.

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:srcCompat="@android:drawable/btn_star_big_on" />
```

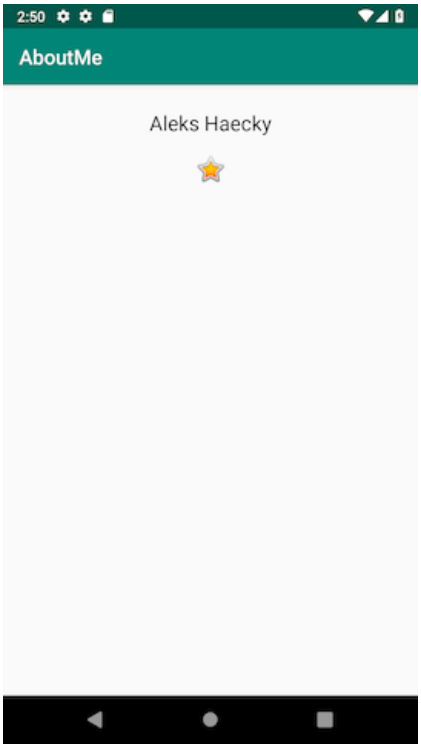
6. To rename the `id` of the `ImageView`, right-click on `"@+id/imageView"` and select **Refactor > Rename**.
7. In the **Rename** dialog, set the `id` to `@+id/star_image`. Click **Refactor**.



> **Tip: Refactor > Rename** renames all the occurrences of an attribute or variable name in your app project.

8. In the **Design** tab, in the **Component Tree**, click the warning icon  next to `star_image`. The warning is for a missing `contentDescription`, which screen readers use to describe images to the user.
9. In the **Attributes** pane, click the three dots **...** next to the `contentDescription` attribute. The **Resources** dialog opens.
10. In the **Resources** dialog, select **Add new resource > New string Value**. Set the **Resource name**field to `yellow_star`, and set the **Resource value** field to `Yellow star`. Click **OK**.

11. Use the **Attributes** pane to add a top margin of `16dp` (which is `@dimen/layout_margin`) to the `yellow_star`**,** to separate the star image from the name.
12. Run your app. Your name and the star image are displayed in your app's UI.

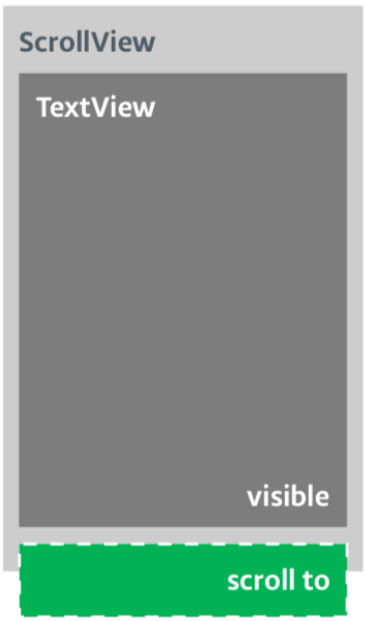


8. Task: Add a ScrollView

A `ScrollView` is a view group that allows the view hierarchy placed within it to be scrolled. A scroll view can contain only one other view, or view group, as a child. The child view is commonly a `LinearLayout`. Inside a `LinearLayout`, you can add other views.

The following image shows an example of a `ScrollView` that contains a `LinearLayout` that contains several other views.

In this task, you will add a `ScrollView` that allows the user to scroll a text view that displays a brief biography. If you are only making one view scrollable, you can put the view directly into the `ScrollView`, which is what you do in this task.



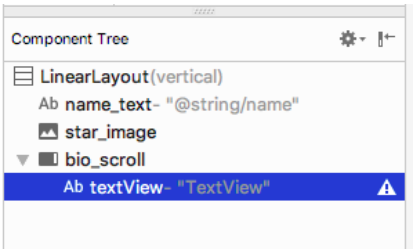
Step 1: Add a ScrollView that contains a TextView

1. Open the `activity_main.xml` file in the **Design** tab.
2. Drag a scroll view into the layout by dragging it into the design editor, or into the **Component Tree**. Put the scroll view below the star image.
3. Switch to the **Text** tab to inspect the generated code.

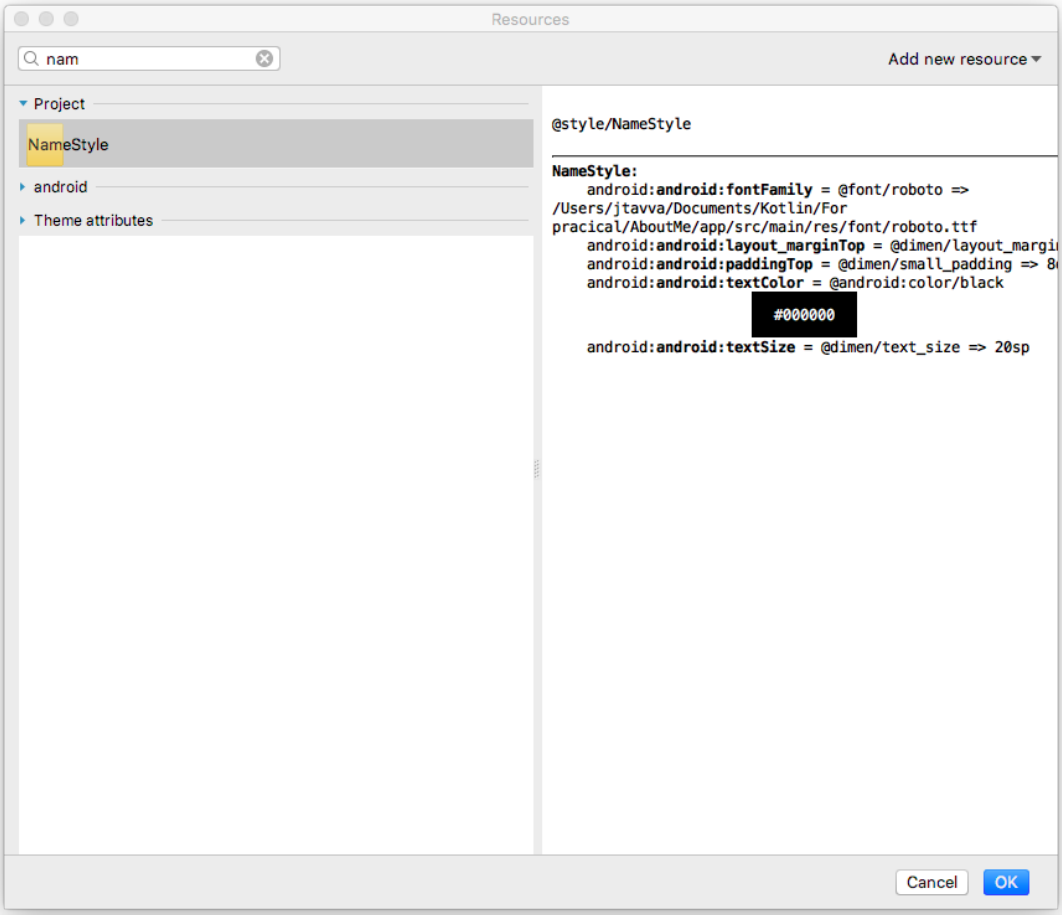
```
// Auto generated code
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

The height and width of the `ScrollView` match the parent element. Once the `name_text` text view and the `star_image` image view have used enough vertical space to display their contents, the Android system lays out the `ScrollView` to fill the rest of the available space on the screen.

4. Add an `id` to the `ScrollView` and call it `bio_scroll`. Adding an `id` to the `ScrollView` gives the Android system a handle for the view so that when the user rotates the device, the system preserves the scroll position.
5. Inside the `ScrollView`, remove the `LinearLayout` code, because your app will only have one view that's scrollable—a `TextView`.
6. Drag a `TextView` from the **Palette** to the **Component Tree**. Put the `TextView` under the `bio_scroll`, as a child element of `bio_scroll`.



7. Set the **id** of the new text view to `bio_text`. 8. Next you add a style for the new text view. In the **Attributes** pane, click the three dots **...** next to the **style** attribute to open the **Resources** dialog. 9. In the **Resources** dialog, search for `NameStyle`. Select `NameStyle` from the list, and click **OK**. The text view now uses the `NameStyle` style, which you created in a prior task.



Step 2: Add your biography to the new TextView

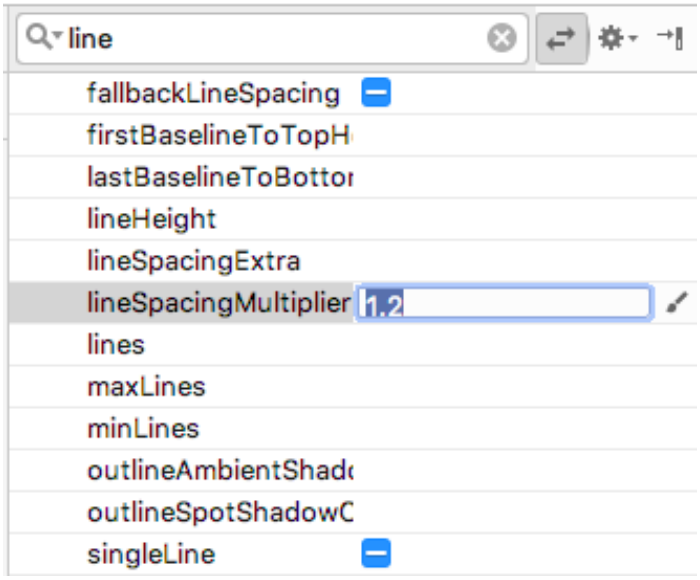
1. Open `strings.xml`, create a string resource called `bio`, and put in some long text about yourself, or about anything that you want.

> **Note:**
>
> - Use `\n` to indicate a line break.
> - If you use an apostrophe, you must escape it with a backslash. For example: "`You mustn\'t forget the backslash.`"
> - For bold text use `<b>...</b>`, and for italicized text use `<i>...</i>`. For example: "`This text is <b>bold</b> and this text is <i>italics</i>.`"

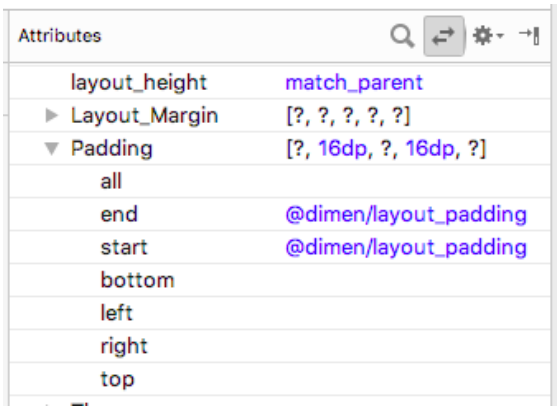
Here is a sample biography:

```
<string name="bio">Hi, my name is Aleks.
\n\nI love fish.
\n\nThe kind that is alive and swims around in an aquarium or river, or a lake, and definit
\nFun fact is that I have several aquariums and also a river.
\n\nI like eating fish, too. Raw fish. Grilled fish. Smoked fish. Poached fish - not so muc
\nAnd sometimes I even go fishing.
\nAnd even less sometimes, I actually catch something.
\n\nOnce, when I was camping in Canada, and very hungry, I even caught a large salmon with
\n\nI\'ll be happy to teach you how to make your own aquarium.
\nYou should ask someone else about fishing, though.\n\n</string>
```

2. In the `bio_text` text view, set the value of the `text` attribute to the `bio` string resource that contains your biography.
3. To make the `bio_text` text easier to read, add spacing between the lines. Use the `lineSpacingMultiplier` attribute, and give it a value of `1.2`.

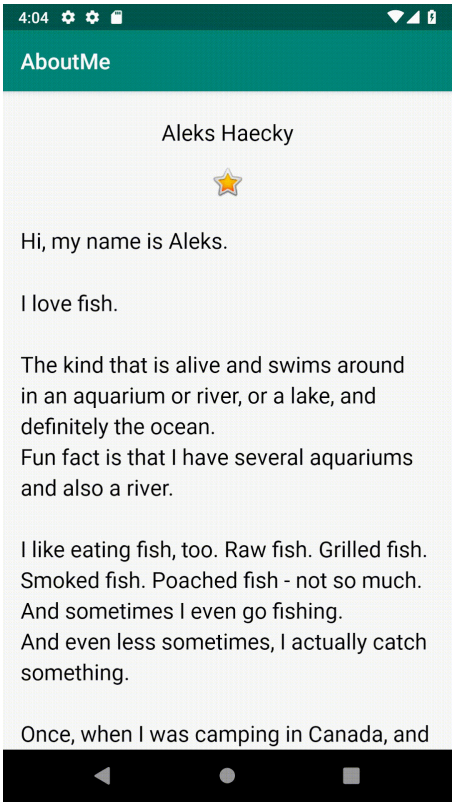


Notice how in the design editor, the `bio` text runs all the way to the side edges of the screen. To fix this problem, you can add left, start, right, and end padding attributes to the root `LinearLayout`. You do not need to add bottom padding, because text that runs right up to the bottom signals to the user that the text is scrollable. 4. Add start and end padding of `16dp` to the root `LinearLayout`. 5. Switch to the **Text**tab, extract the dimension resource, and name it `layout_padding`.



> **Note:** Starting from API level 17, use "start" and "end" instead of "left" and "right" for padding and margin to adapt your app for RTL languages like Arabic.

6. Run your app and scroll through the text.



Congratulations!

You have created a complete app from scratch, and it looks great!

Android Kotlin Fundamentals: Add user interactivity

About this codelab

subject Last updated Jun 24, 2020

account_circle Written by Google Developers Training team

1. Introduction

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

What you should already know

- Creating a basic Android app in Kotlin.
- Running an Android app on an emulator or on a device.
- Creating a linear layout using Android Studio's Layout Editor .
- Creating a simple app that uses `LinearLayout` , `TextView` , `ScrollView` , and a button with a click handler.

What you'll learn

How to get user input using an `EditText` view.

How to set text to a `TextView` view using the text from the `EditText` view.

How to work with `View` and `ViewGroup` .

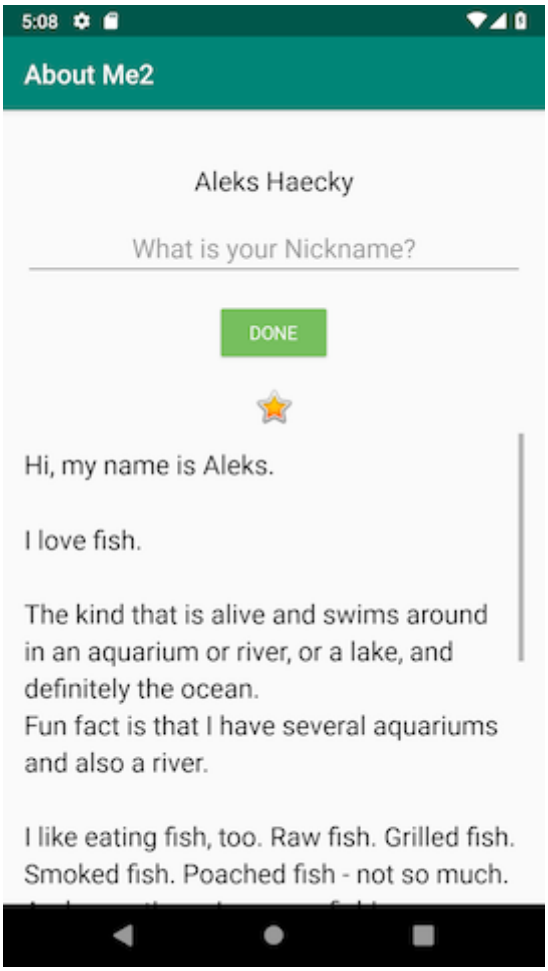
How to change the visibility of a `View` .

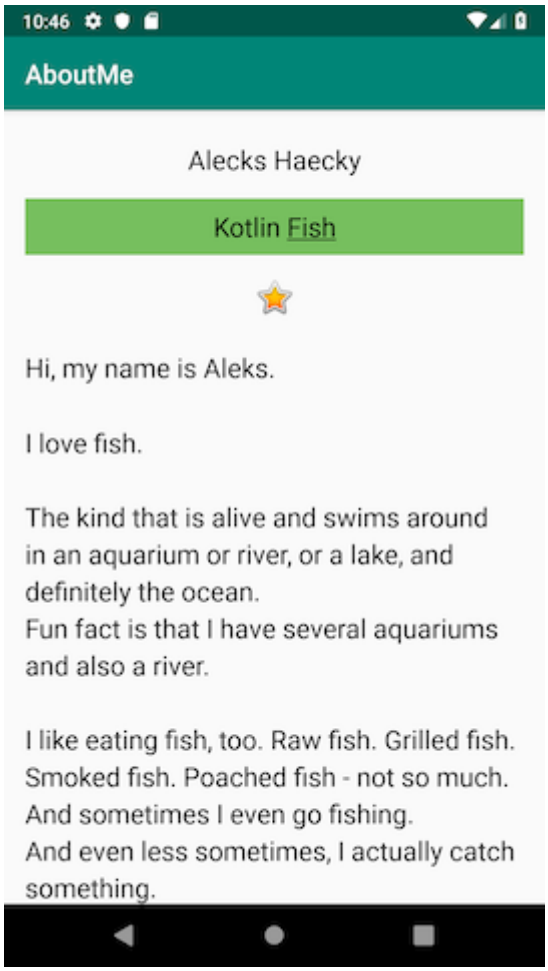
What you'll do

- Add interactivity to the AboutMe app, which is from a previous codelab.
- Add an `EditText` so the user can enter text.
- Add a `Button` and implement its click handler.

2. App overview

In this codelab, you extend the AboutMe app to add user interaction. You add a nickname field, a **DONE**button, and a text view to display the nickname. Once the user enters a nickname and taps the **DONE** button, the text view updates to show the entered nickname. The user can update the nickname again by tapping the text view.



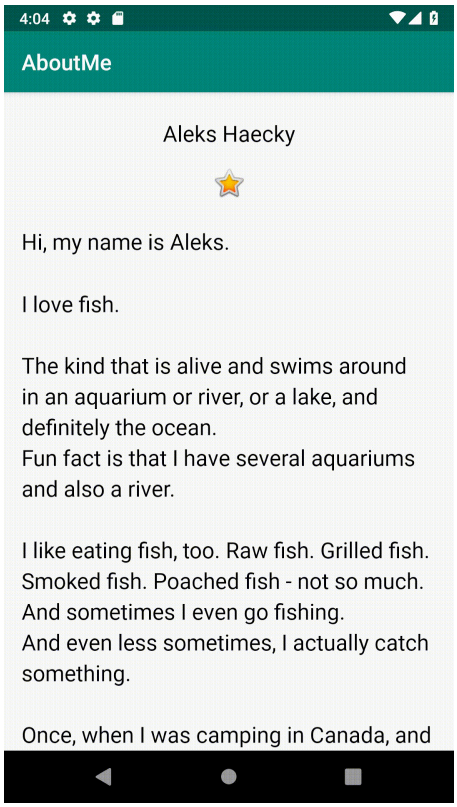


3. Task: Add an EditText for text input

In this task, you add an `EditText` input field to allow the user to enter a nickname.

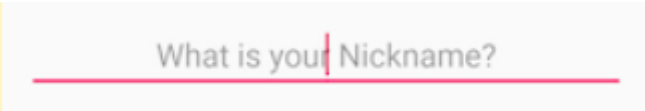
Step 1: Get started

1. If you do not already have the AboutMe app from a previous codelab, download the starter code, AboutMeInteractive-Starter. This the same code you finished in a previous codelab.
2. Open the AboutMeInteractive-Starter project in Android Studio.
3. Run the app. You see a name text view, a star image, and a long segment of text in a scroll view.



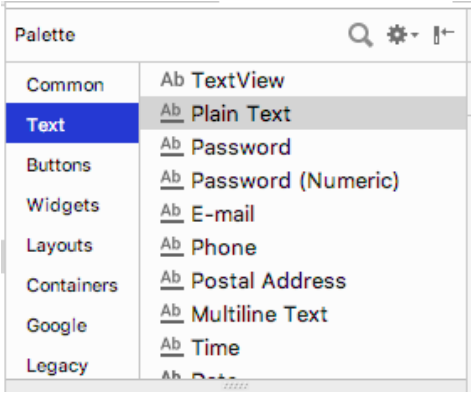
Notice that the user can't change any of the text.

Apps are more interesting if the user can interact with the app, for example if the user can enter text. To accept text input, Android provides a user interface (UI) widget called an *edit text*. You define an edit text using `EditText`, a subclass of `TextView`. An edit text allows the user to enter and modify text input, as shown in the screenshot below.



Step 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**.

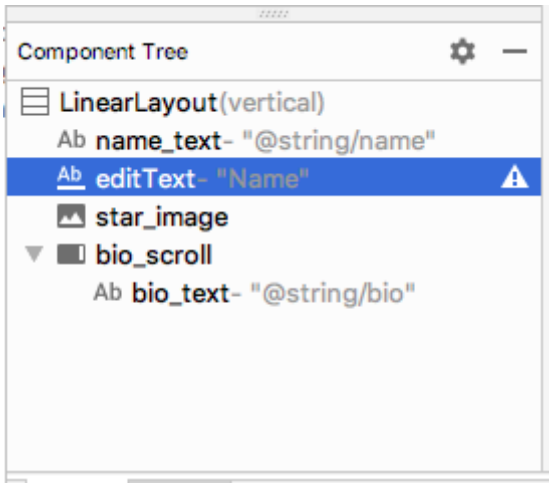


Ab TextView, which is a `TextView`, shows at the top of the list of text elements in the **Palette**pane. Below **Ab TextView** are multiple `EditText` views.

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).

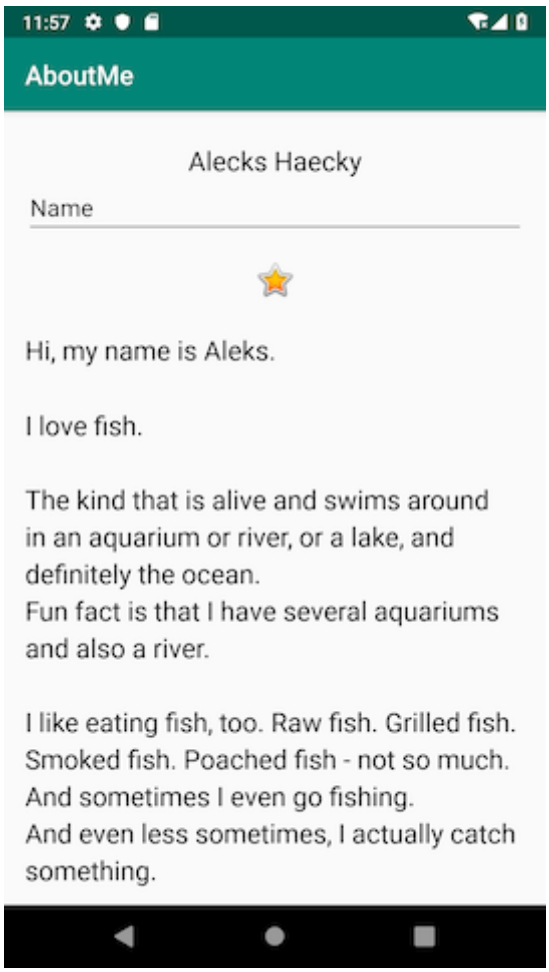
3. Drag a **PlainText** edit text into the **Component Tree** and place it below the `name_text` and above the `star_image`.



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

Attribute	Value
`id`	`nickname_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".


11:57
AboutMe
Alecks Haecky
Name
Hi, my name is Aleks.
I love fish.
The kind that is alive and swims around in an aquarium or river, or a lake, and definitely the ocean.
Fun fact is that I have several aquariums and also a river.
I like eating fish, too. Raw fish. Grilled fish. Smoked fish. Poached fish - not so much.
And sometimes I even go fishing.
And even less sometimes, I actually catch something.

4. Task: Style your EditText

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

Step 1: Add hint text

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

```
<string name="what_is_your_nickname">What is your Nickname?</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:

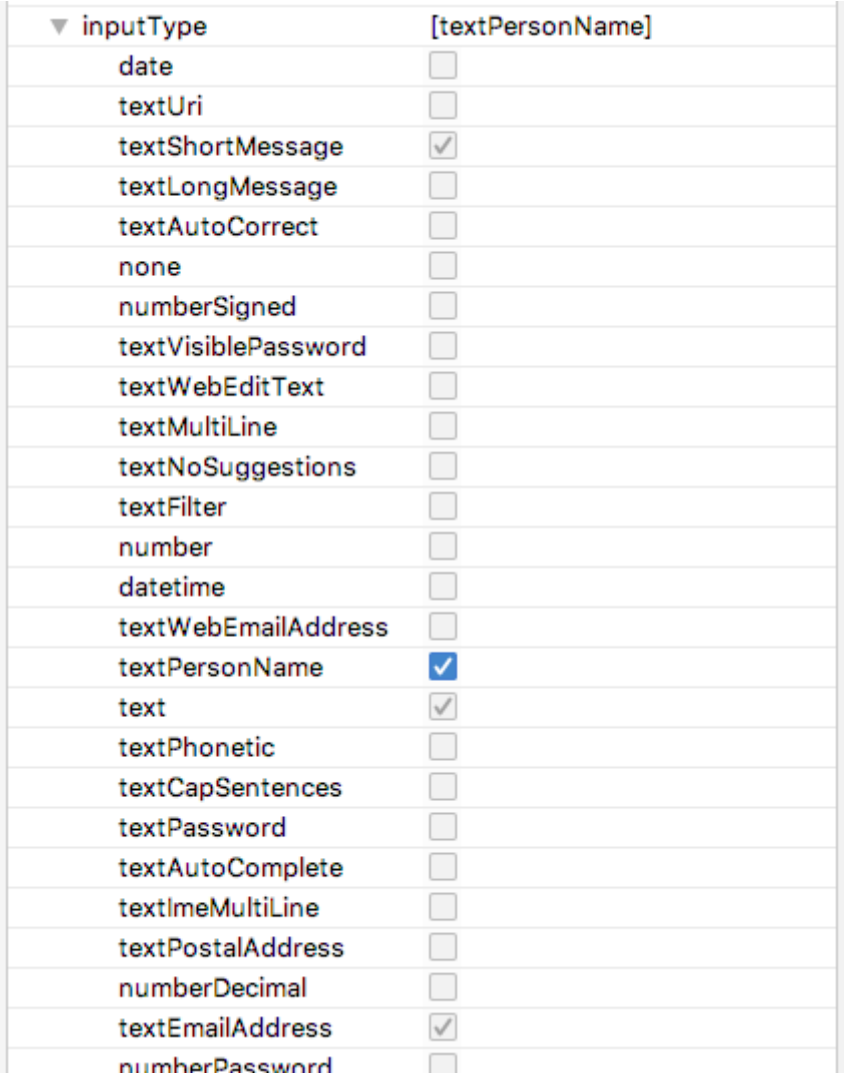
Attribute	Value
`style`	`NameStyle`
`textAlignment`	 *(center)*
`hint`	`@string/what_is_your_nickname`

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.

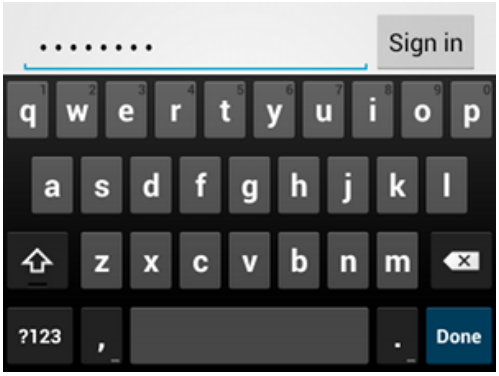
Step 2: Set the inputType attribute

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 three dots **...** 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.



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 the nickname field:

1. Set the `inputType` attribute to `textPersonName` for the `nickname_edit` edit text.

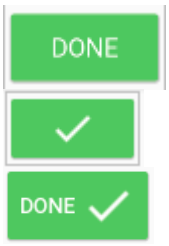
2. In the **Component Tree** pane, notice an `autoFillHints` warning. This warning does not apply to this app and is beyond the scope of this codelab, so you can ignore it. (If you want to learn more about autofill, see Optimize your app for autofill.)

3. In the **Attributes** pane, verify the values for the following attributes of the `EditText` view:

Attribute	Value
`id`	`nickname_edit`
`layout_width`	`match_parent` *(default)*
`layout_height`	`wrap_content` *(default)*
`style`	`@style/NameStyle`
`inputType`	`textPersonName`
`hint`	`"@string/what_is_your_nickname"`
`text`	*(empty)*

5. Task: Add a button and style it

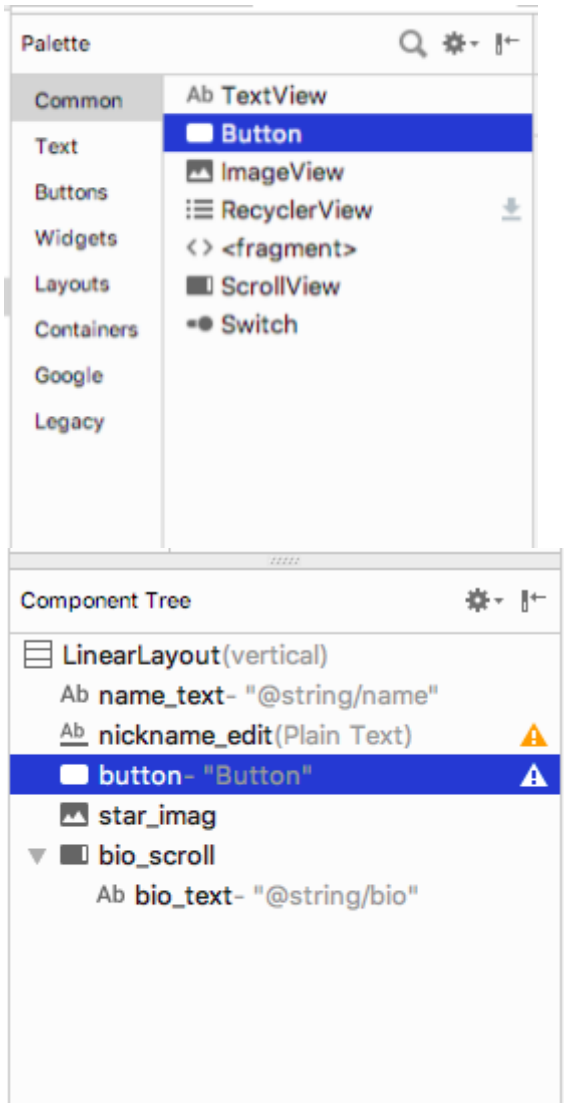
A `Button` is a UI element that the user can tap to perform an action. A button can consist of text, an icon, or both text and an icon.



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

Step 1: Add a DONE button

1. Drag a button from the **Palette** pane into the **Component Tree**. Place the button below the `nickname_edit` edit text.



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

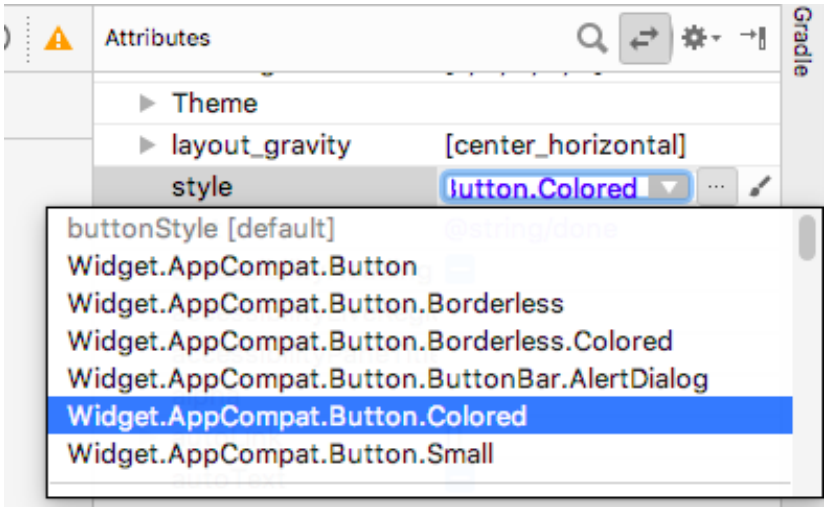
```
<string name="done">Done</string>
```

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

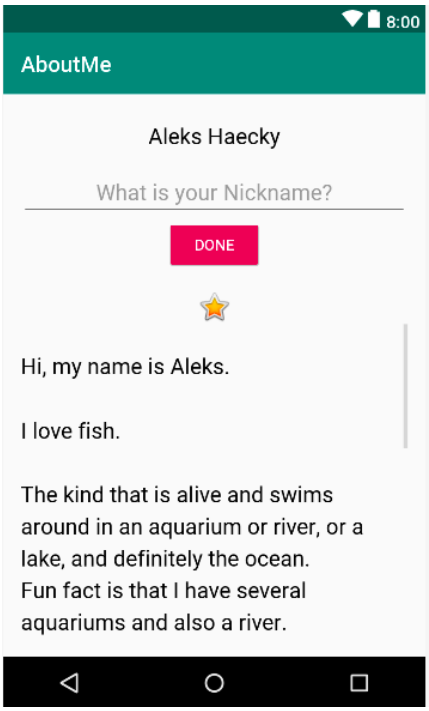
Attribute	Values
`id`	`done_button`
`text`	`@string/done`
`layout_gravity`	`center_horizontal`
`layout_width`	`wrap_content`

The `layout_gravity` attribute centers the view in its parent layout, `LinearLayout`.

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.



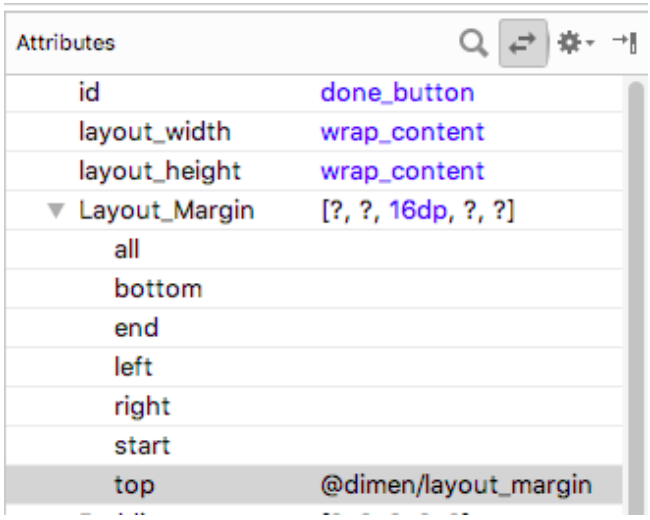
The `colors.xml` file contains the default colors for your app. You can add new color resources or change the existing color resources in your project, based on your app's requirements.

Sample `colors.xml` file:

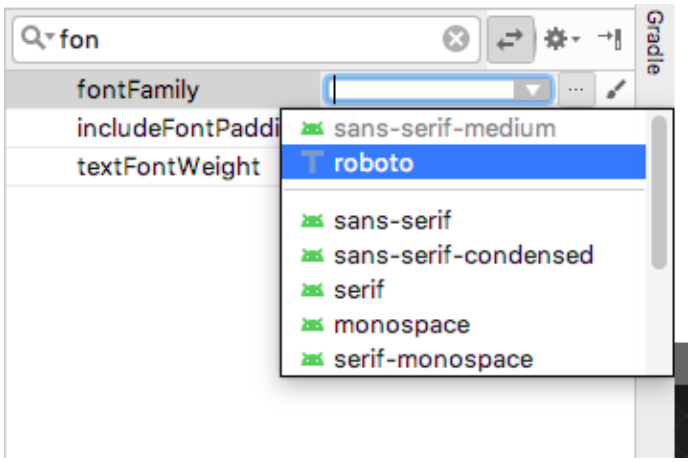
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#008577</color>
    <color name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>
</resources>
```

Step 2: 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.



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:layout_gravity="center_horizontal"
    android:layout_marginTop="@dimen/layout_margin"
    android:fontFamily="@font/roboto"
    android:text="@string/done" />
```

Step 3: 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`.

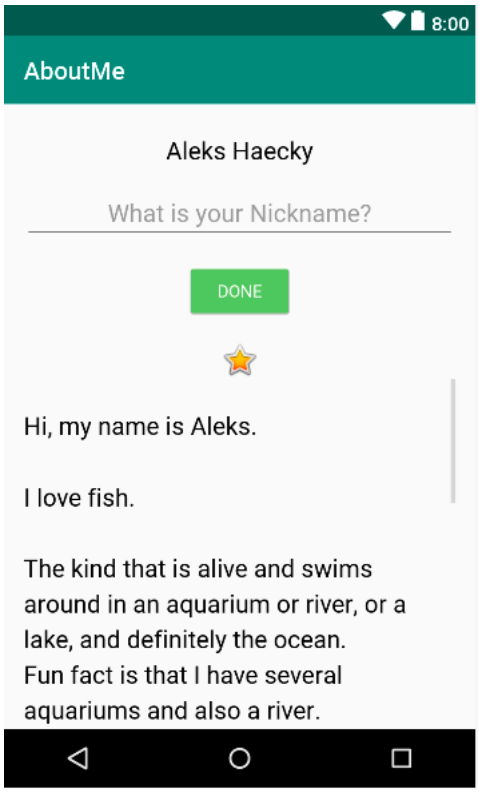
```
<color name="colorAccent">#76bf5e</color>
```

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

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <color name="colorPrimary">#008577</color>
4      <color name="colorPrimaryDark">#00574B</color>
5      <color name="colorAccent">#76bf5e</color>
6  </resources>
7
```

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

2. Run your app. You should see a styled **DONE** button below the edit text.

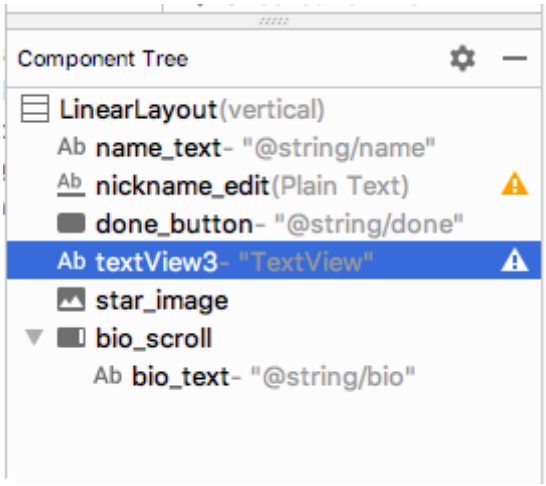


6. Task: Add a TextView to display the nickname

After the user enters a nickname and taps the **DONE** button, the nickname displays in a `TextView` view. In this task, you add a text view with a colored background. The text view displays the user's nickname above the `star_image`.

Step 1: Add a TextView for the nickname

1. Drag a text view from the **Palette** pane into the **Component Tree**. Place the text view below the `done_button` and above the `star_image`.



2. Use the **Attributes** pane to set the following attributes for the new `TextView` view:

Attribute	Value
`id`	`nickname_text`
`style`	`NameStyle`
`textAlignment`	*(center)*

Step 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 the `nickname_text` text view to `gone`, because you don't want your app to show this text view at first.

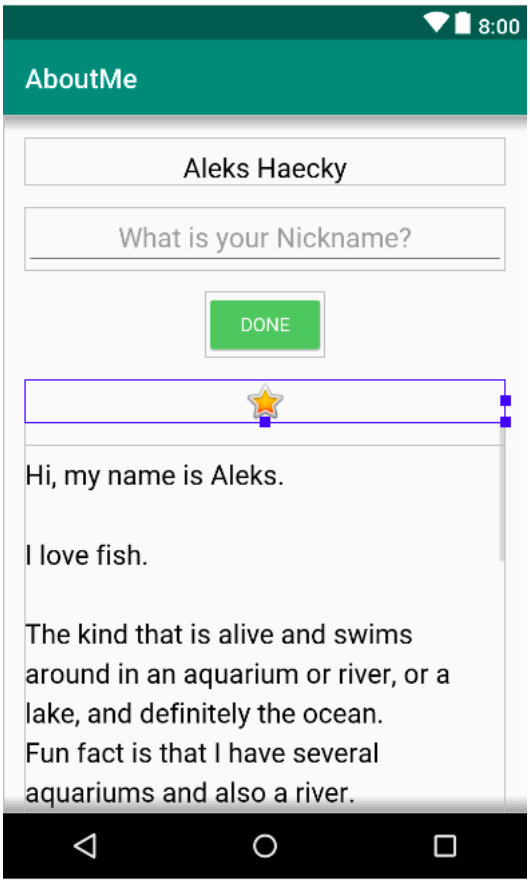
Notice that as you change the attribute in the **Attributes** pane, the `nickname_text` view disappears from the design editor. The view is hidden in the layout preview.

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

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

```
<TextView
    android:id="@+id/nickname_text"
    style="@style/NameStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAlignment="center"
    android:visibility="gone"
    android:text="" />
```

Your layout preview should look something like the following:



7. Task: Add a click listener to the DONE button

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 generically 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 `addNickname`, will do the following:

- Get the text from the `nickname_edit` edit text.
- Set the text in the `nickname_text` text view.
- Hide the edit text and the button.
- Display the nickname `TextView`.

Step 1: Add a click listener

1. In Android Studio, in the `java` folder, open the `MainActivity.kt` file.
2. In `MainActivity.kt`, inside the `MainActivity` class, add a function called `addNickname`. 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 addNickname(view: View) {
}
```

3. Inside the `addNickname` function, use `findViewById()` to get a reference to the `nickname_edit` edit text and the `nickname_text` text view.

```
val editText = findViewById<EditText>(R.id.nickname_edit)
val nicknameTextView = findViewById<TextView>(R.id.nickname_text)
```

4. Set the text in the `nicknameTextView` text view to the text that the user entered in the `editText`, getting it from the `text` property.

```
nicknameTextView.text = editText.text
```

5. Hide the nickname `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
```

6. 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
```

7. At the end of the `addNickname` function, make the nickname `TextView` view visible by setting its `visibility` property to `View.VISIBLE`.

```
nicknameTextView.visibility = View.VISIBLE
```

Step 2: Attach the click listener to the DONE Button

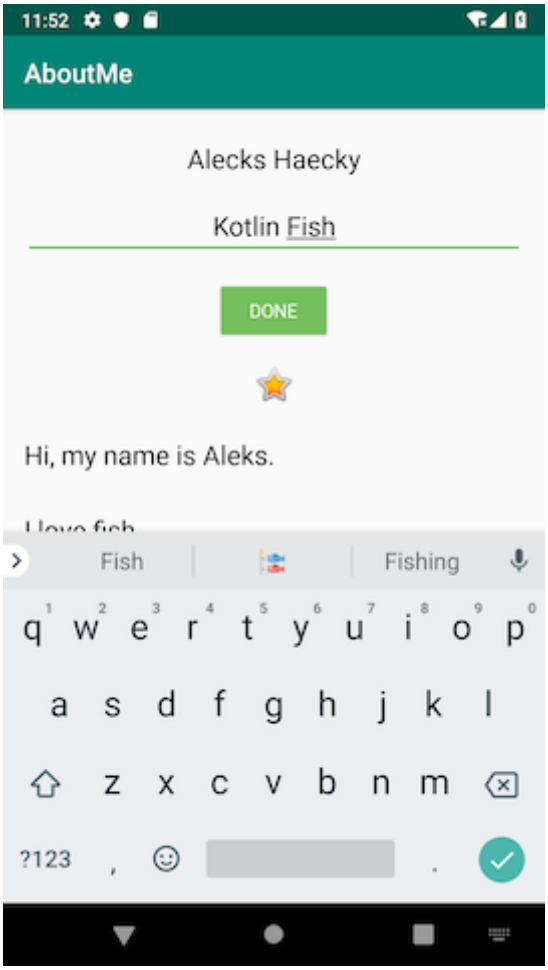
Now that you have a function that defines the action to be performed when the **DONE** button is tapped, you need to attach the function to the `Button` view.

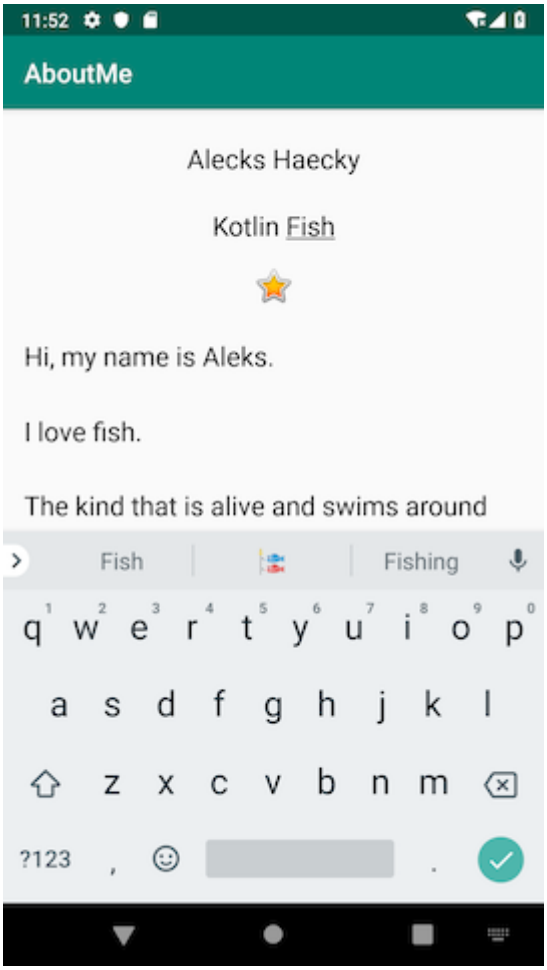
1. In `MainActivity.kt`, at the end of the `onCreate()` function, get a reference to the **DONE** `Button` view. Use the `findViewById()` function and call `setOnClickListener`. Pass in a reference to the click-listener function, `addNickname()`.

```
findViewById<Button>(R.id.done_button).setOnClickListener {
            addNickname(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 nickname, and tap the **DONE** button. Notice how the edit text and the button are replaced by the nickname text view.





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

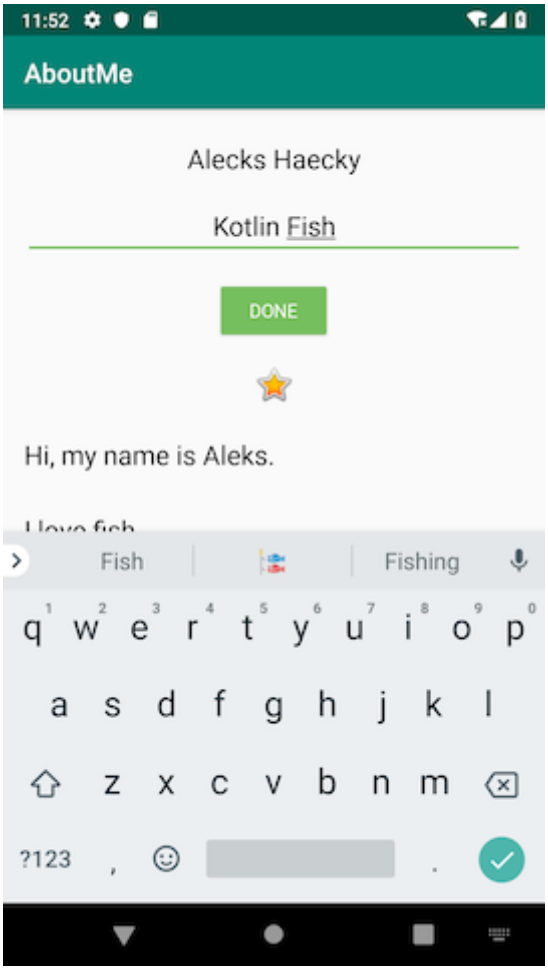
Step 3: Hide the keyboard

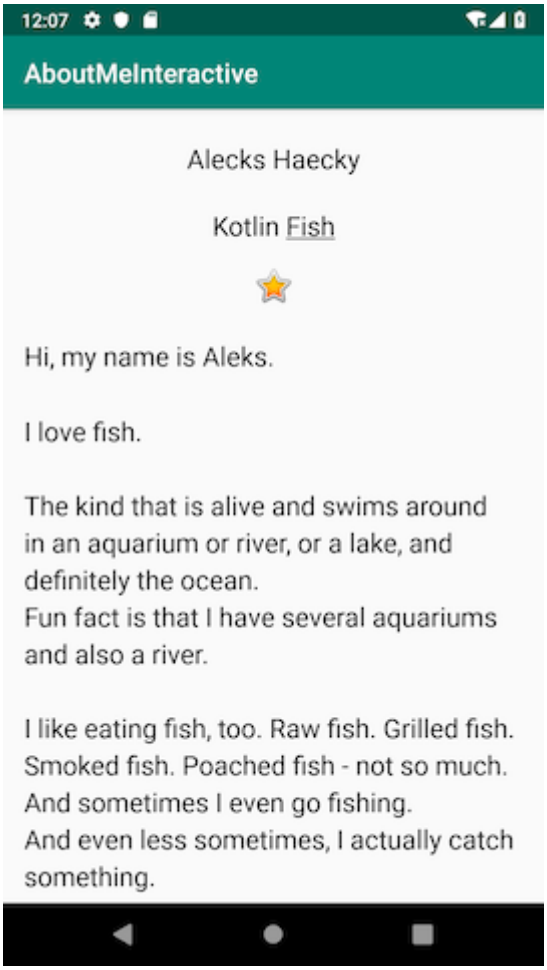
In this step, you add code to hide the keyboard after the user taps the **DONE** button.

1. In `MainActivity.kt`, at the end of `addNickname()` 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 InputMethodManag
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 the nickname after they tap the **DONE** button. In the next task, you make the app more interactive and add functionality so that the user can update the nickname.

8. Task: Add a click listener to the nickname TextView

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

Step 1: Add a click listener

1. In `MainActivity`, add a click-listener function called `updateNickname(view: View)` for the nickname text view.

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

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

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

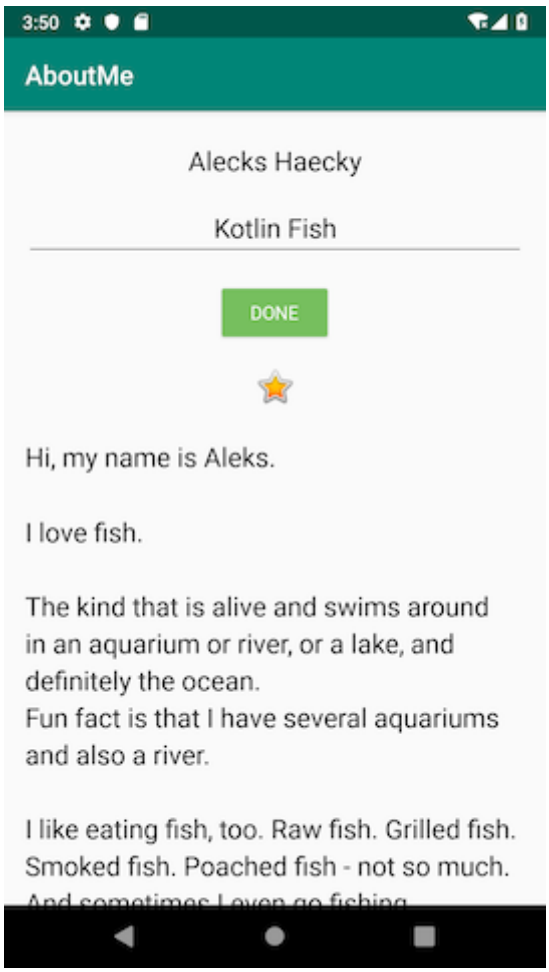
3. At the end of the `updateNickname` function, add code to show the edit text, show the **DONE**button, and hide the text view.

```
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 the `nickname_text` text view. Pass in a reference to the click-listener function, which is `updateNickname()`.

```
findViewById<TextView>(R.id.nickname_text).setOnClickListener {
   updateNickname(it)
}
```

5. Run your app. Enter a nickname, tap the **DONE** button, then tap the nickname `TextView` view. The nickname 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 nickname text view is clickable. In the next task, you add focus and a style to the nickname text view.

Step 2: Set the focus to the EditText view and show the keyboard

1. At the end of the `updateNickname` 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 `updateNickname` function, add code to make the keyboard visible.

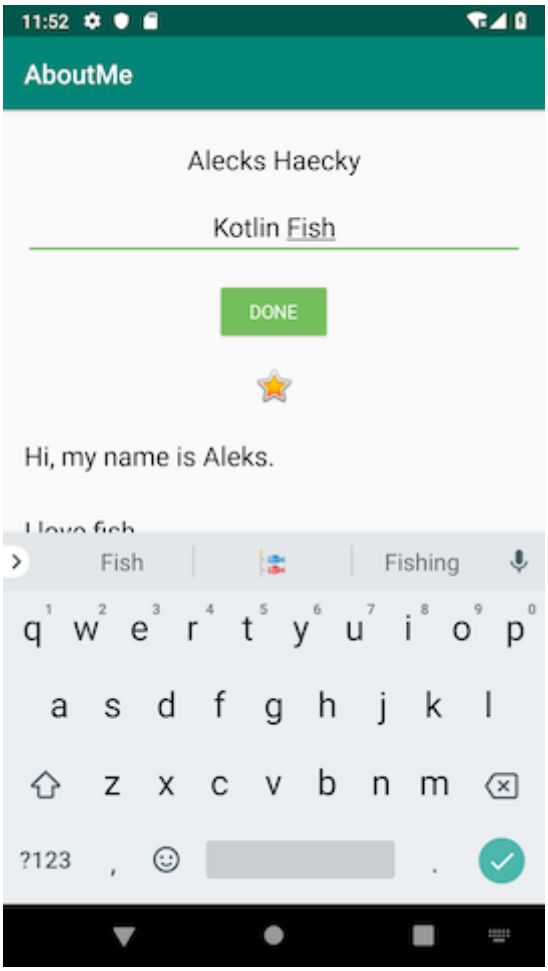
```
// Show the keyboard.
val imm = getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
imm.showSoftInput(editText, 0)
```

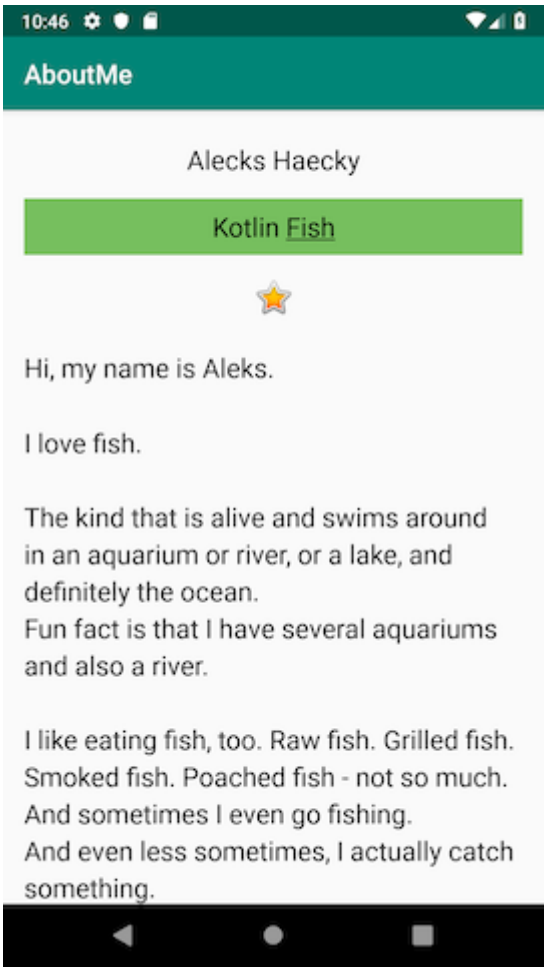
Step 3: Add a background color to the nickname TextView view

1. Set the background color of the `nickname_text` text view to `@color/colorAccent` , and add a bottom padding of `@dimen/small_padding` . These changes will serve as a hint to the user that the nickname text view is clickable.

```
android:background="@color/colorAccent"
android:paddingBottom="@dimen/small_padding"
```

2. Run your final app. The edit text has focus, the nickname is displayed in the edit text, and the nickname text view is styled.





Now go show your interactive AboutMe app to a friend!

Android Kotlin Fundamentals: 02.3 ConstraintLayout using the Layout Editor

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Introduction

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

What you should already know

You should be familiar with:

- Creating a basic Android app in Kotlin.
- Running an Android app on an emulator or on a device.
- Using `LinearLayout` in your app.
- The basics of Android Studio's Layout Editor.

What you'll learn

How to use `ConstraintLayout` in your app to arrange views.

How to change a text view's background color.

How to align a view with text using a baseline constraint.

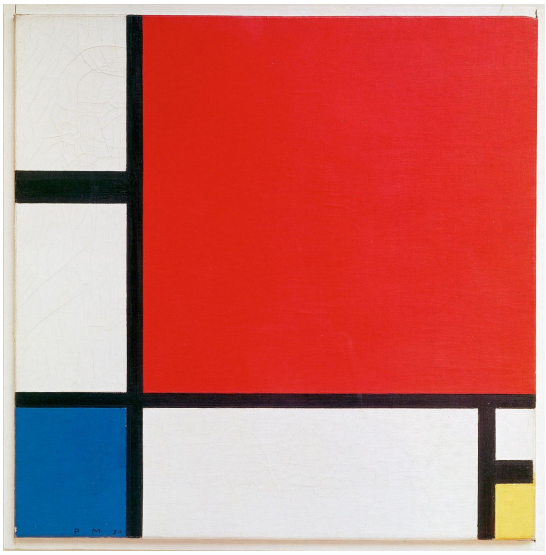
How to create vertical and horizontal chains from a group of views.

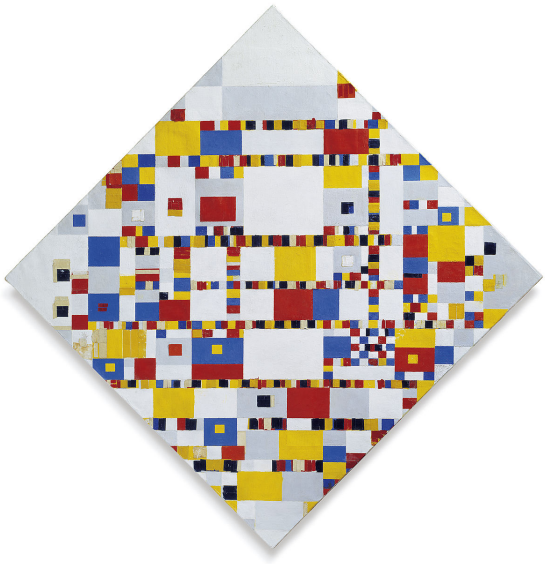
What you'll do

- Create the ColorMyViews 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.
- Add a chain of three buttons and constrain the chain to the bottom of the layout.

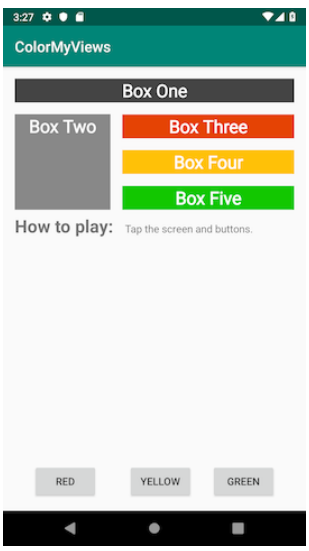
2. App overview

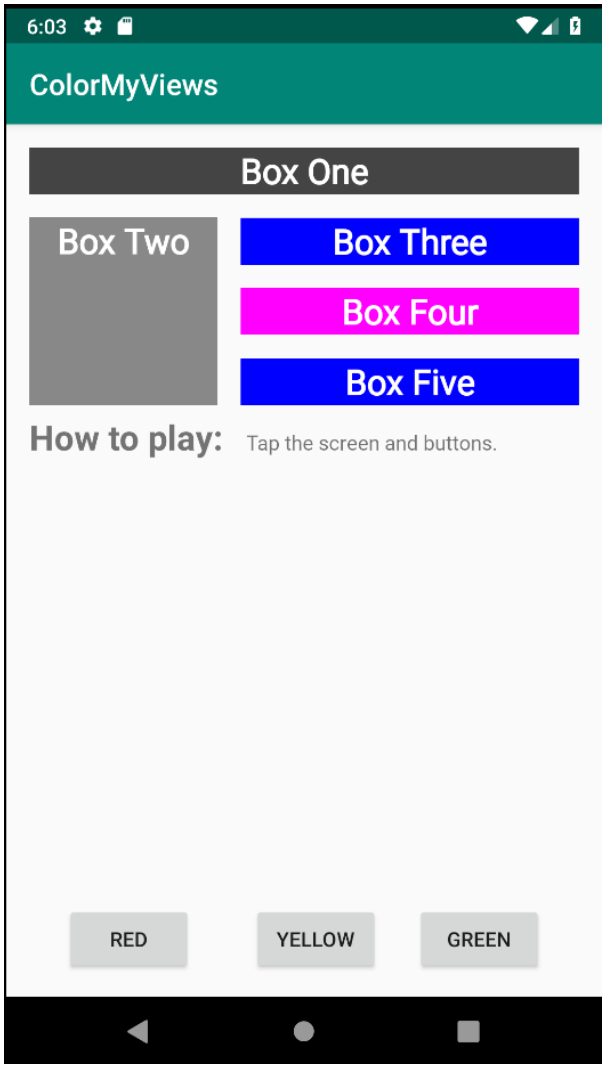
The ColorMyViews app is inspired by the Dutch artist, Piet Mondrian. He invented a style of painting style called *neoplasticism*, which uses only vertical and horizontal lines and rectangular shapes in black, white, gray, and primary colors.





Although paintings are static, your app will be interactive! The app consists of clickable text views that change color when tapped, and button views in a `constraintLayout` .





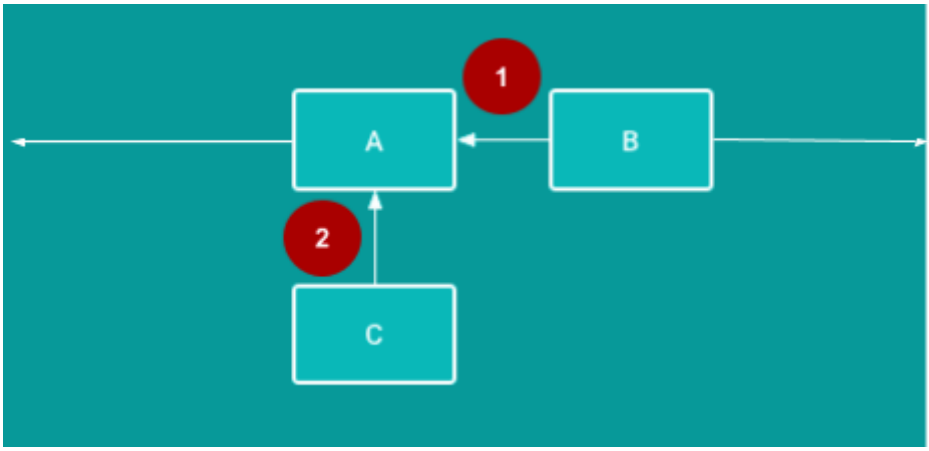
ConstraintLayout

A `ConstraintLayout` is a `ViewGroup` that allows you to position and size child views in a flexible way. A ConstraintLayout allows you to create large, complex layouts with flat view hierarchies (no nested view groups). To build a ConstraintLayout, you can use the Layout Editor to add constraints, and to drag-and-drop views. You don't need to edit the XML.

> **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 ConstraintLayout, 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.)

3. Task: Create the ColorMyViews project

1. Open Android Studio, if it's not already open, and create a new project with the following parameters:

Attribute	**Value**
Template	**Empty Activity** in the **Phone and Tablet** tab
Application Name	ColorMyViews
Company Name android	`com.android.example.colormyviews` (*or your own domain*)
Language	Kotlin
Minimum API level	API 19: Android 4.4 (KitKat)
This project will support instant apps	(*Leave this box cleared*)
Use AndroidX artifacts	Select this box.

The Empty Activity template creates a single empty activity in the `Mainactivity.kt` file. The template also creates a layout file called `activity_main.xml`. The layout uses `ConstraintLayout` as its root view group, with a single `TextView` as the layout's content.

2. Wait for Android Studio to finish the Gradle build. If you see any errors, select **Build** > **Rebuild Project**.
3. Run the app and wait for a few seconds for the build to complete. You should see a screen with "Hello World!" in the middle of it.

4. Task: Use Layout Editor to build a ConstraintLayout

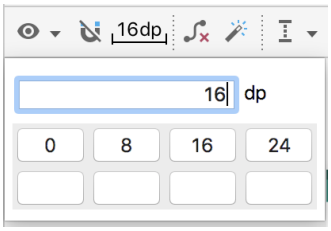
In this task, you use the Android Studio Layout Editor to build a ConstraintLayout for your app.

Step 1: Set up your Android Studio work area

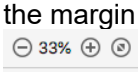
1. Open the `activity_main.xml` file and click the **Design** tab.
2. You'll add constraints manually, so you want autoconnect turned off. In the toolbar, locate the **Turn Off/On Autoconnect** toggle button, which is shown below. (If you can't see the toolbar, click inside the design editor area of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

	Autoconnect is on.
	Autoconnect is off—this is what you want for this codelab.

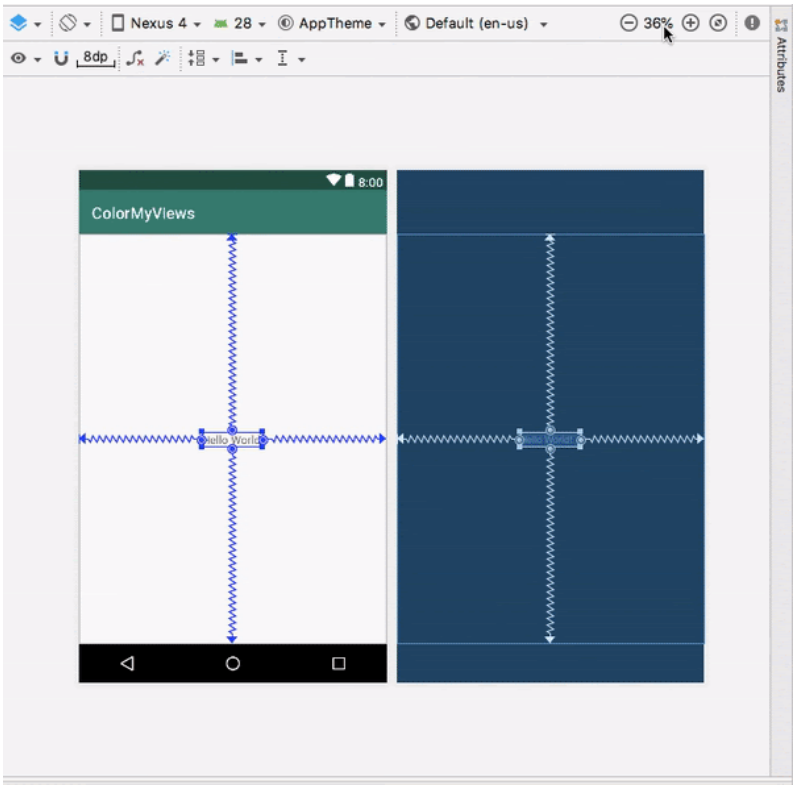
3. Use the toolbar to set the default margins to `16dp`. (The default is `8dp`.)



When you set the default margin to `16dp`, new constraints are created with this margin, so you don't have to add the margin each time you add a constraint. 4. Zoom in using the **+** icon

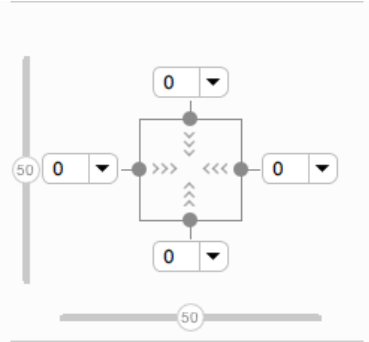


on the right side of the toolbar, until the **Hello World** text is visible inside its text view. 5. Double-click on the **Hello World** text view to open the **Attributes** pane.



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

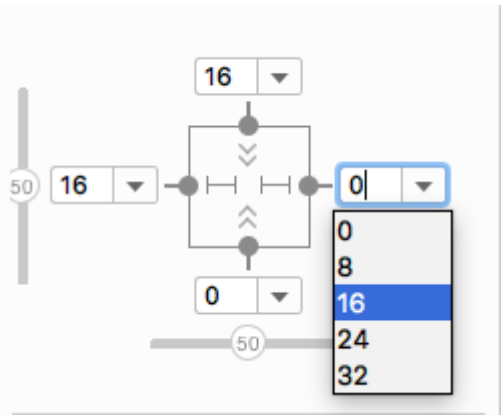
Constraint bias positions the view element along the horizontal and vertical axes. By default, the view is centered between the two constraints with a bias of 50%.

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.

Step 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`).



Step 3: Adjust constraints and margins for the text view

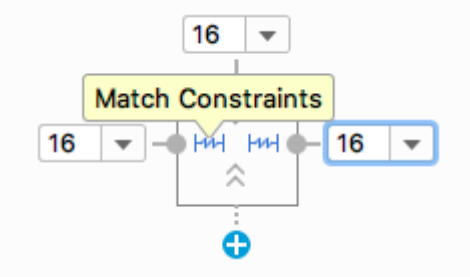
In the view inspector, the arrows

inside the square represents the type of the constraint:

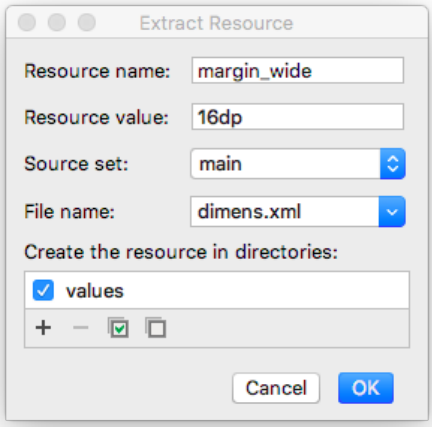
- 
  Wrap Content: The view expands only as much as needed to contain its contents.
- 
  Fixed: You can specify a dimension as the view margin in the text box next to the fixed-constraint arrows.
- 

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 **Text** tab. Extract the dimension resource for `layout_marginStart`, and set the **Resource name** to `margin_wide`.



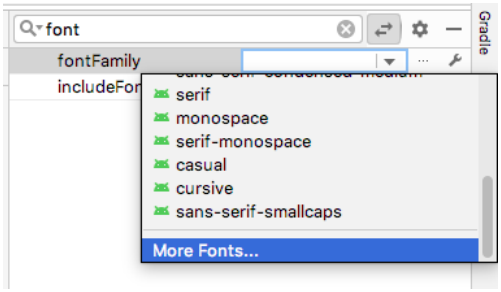
4. Set the same dimension resource, `@dimen/margin_wide`, for the top and end margins.

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

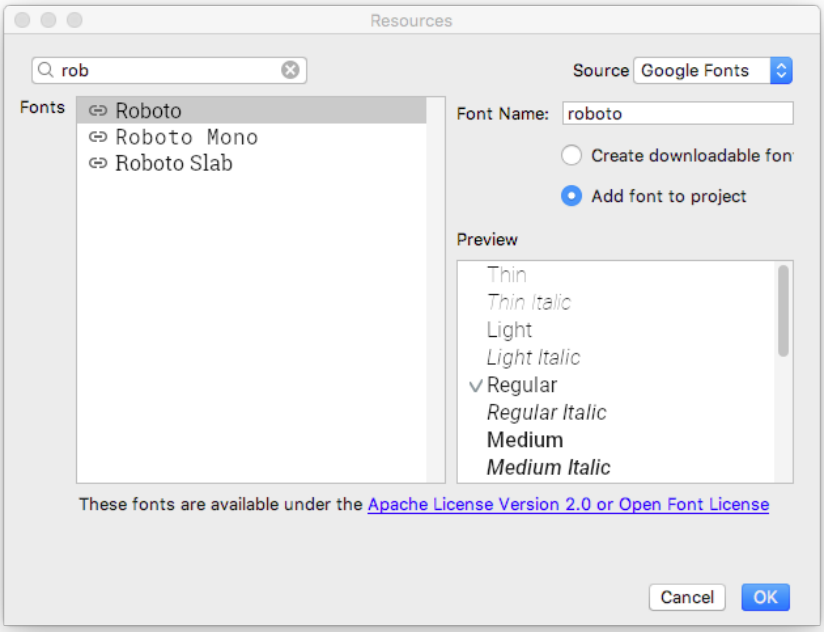
5. Task: Style the TextView

Step 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.

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

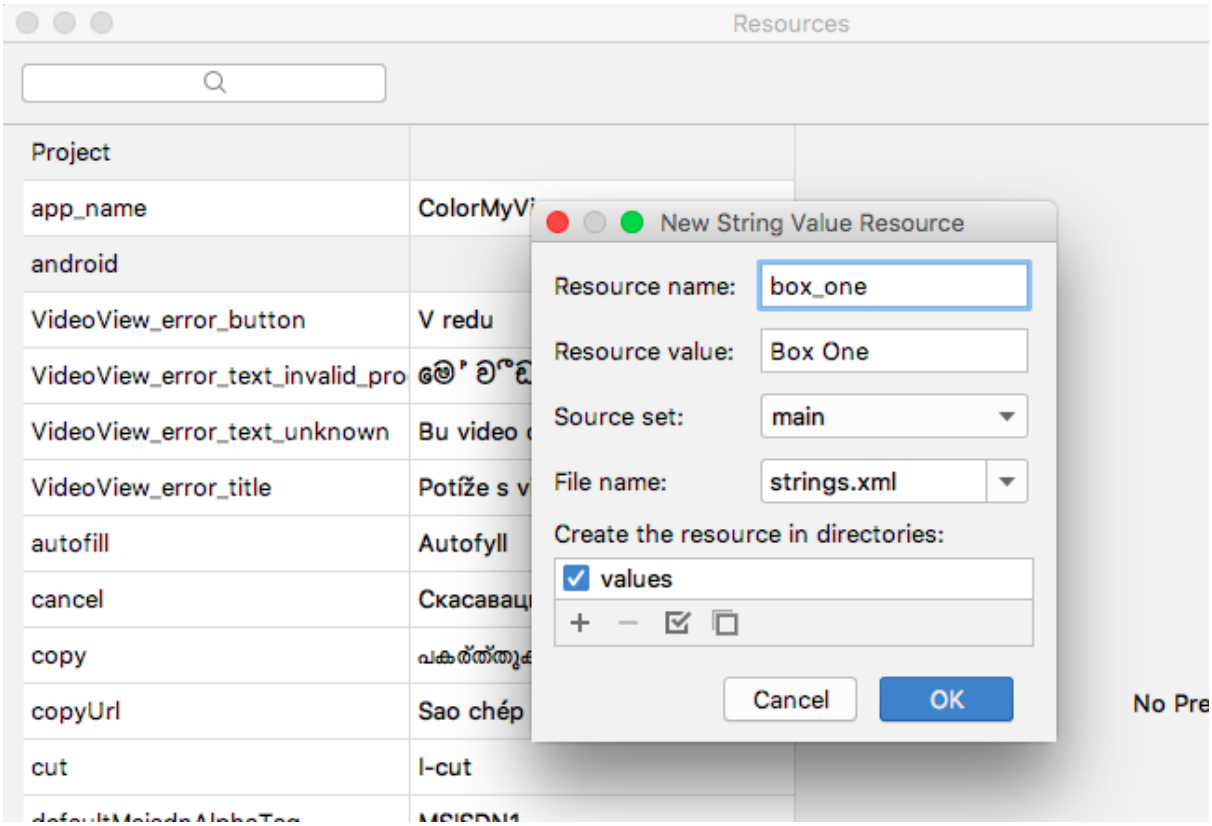
```
<style name="whiteBox">
    <item name="android:background">@android:color/holo_green_light</item>
```

```
    <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>
```

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`.

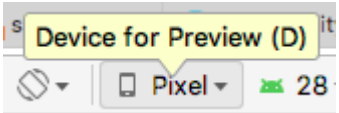
Step 3: Add a string resource for the text view

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, select **Add new resource > New string Value**. Set the resource name to `box_one`, and set the value to `Box One`.
4. Click **OK**.



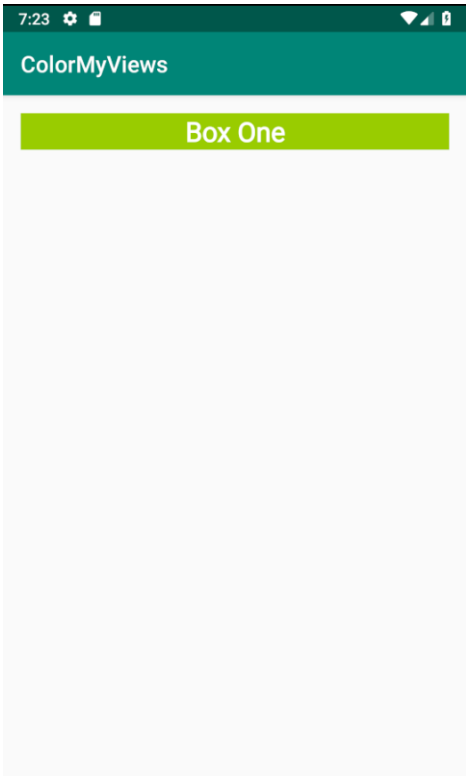
Step 4: Finish setting attributes for the text view

1. In the **Attributes** pane, set the `id` of the text view to `box_one_text`.
2. Set the `style` to `@style/whiteBox`.
3. To clean up the code, switch to the **Text** tab and remove the `android:fontFamily="@font/roboto"` attribute, because this font is present in the `whiteBox` 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 "Box One".

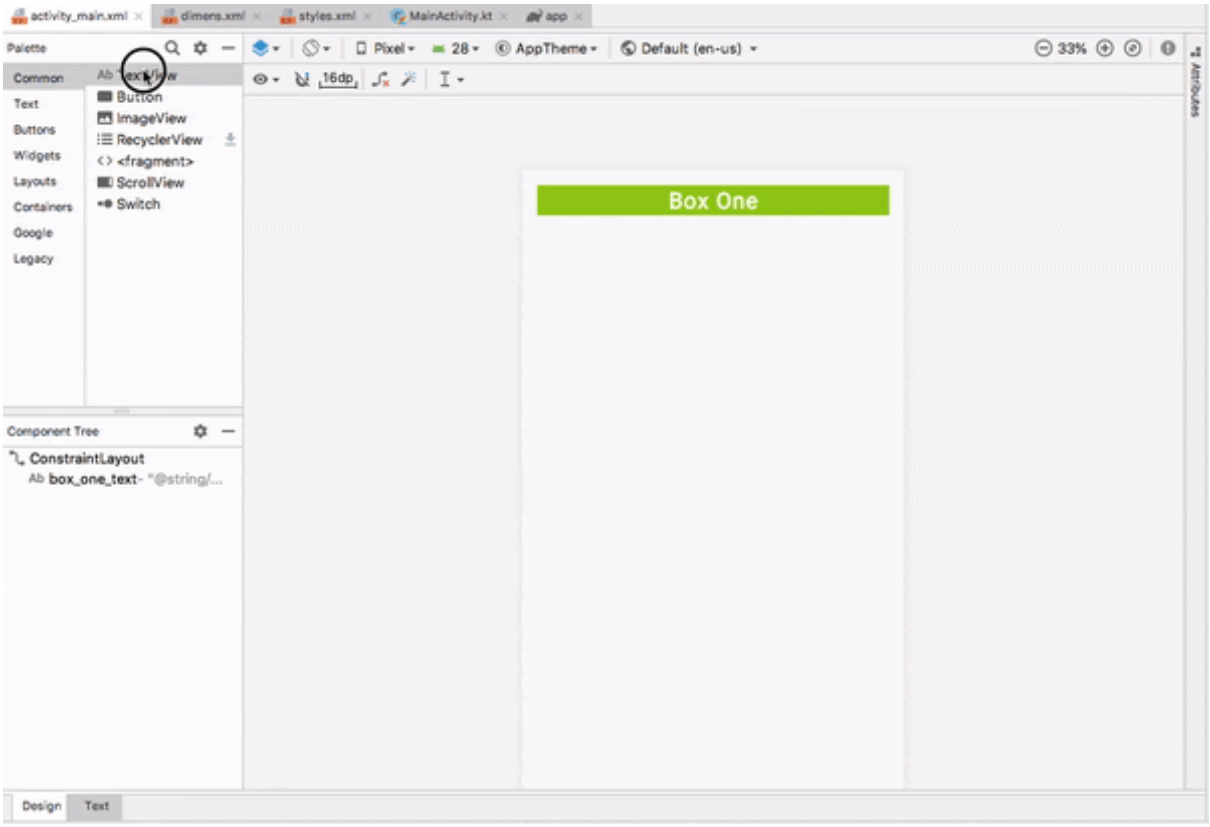


6. Task: Add a second TextView and add constraints

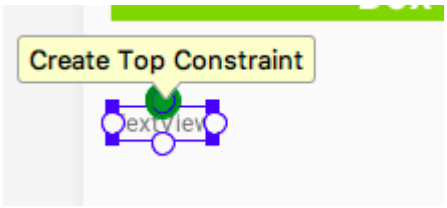
In this task, you add another text view below the `box_one_text` . You constrain the new text view to `box_one_text` and the layout's parent element.

Step 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 `box_one_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*.

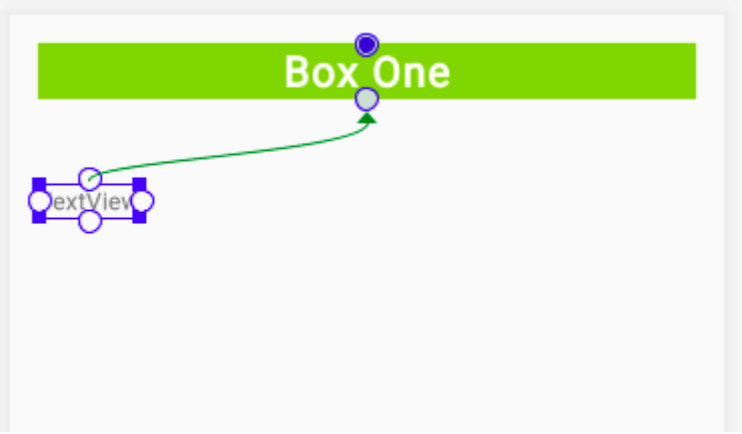


Notice that when you hold the pointer over the constraint handle, the handle turns green and blinks.

Step 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 **Box One** text view:

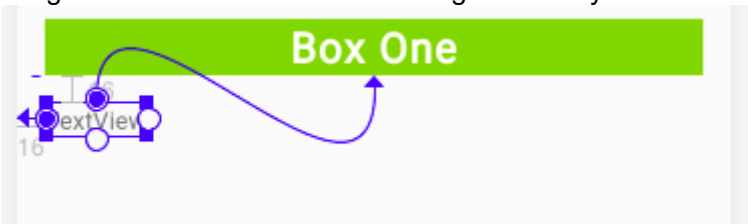
1. Hold the pointer over the top constraint handle on the new text view.
2. Click the top constraint handle of the view and drag it up. A constraint line appears. Connect the constraint line to the bottom of the **Box One** textview, as shown below.



As you release the click, the constraint is created, and the new text view jumps to within 16dp of the bottom of the **Box One**. (The new text view has a top margin of 16dp because that's the default you set earlier.)

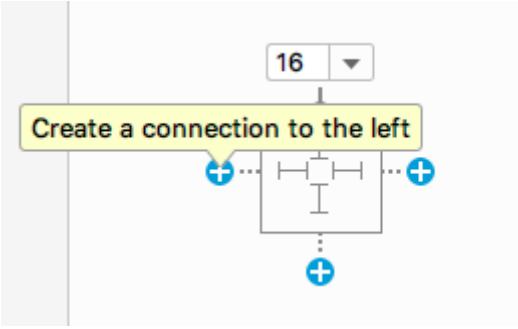
Now create a left constraint:

1. Click the constraint handle on the left side of the new view.
2. Drag the constraint line to the left edge of the layout.



Tip: You can also create constraints using the view inspector. For example, to create a left constraint on the new text box:

1. In the preview, click the new text box to select it.
2. In the view inspector, click the **+** icon  on the left side of the box, as shown below.



When you create a constraint this way, the constraint is attached to the parent or to a view closer to it.

Step 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="box_two">Box Two</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:

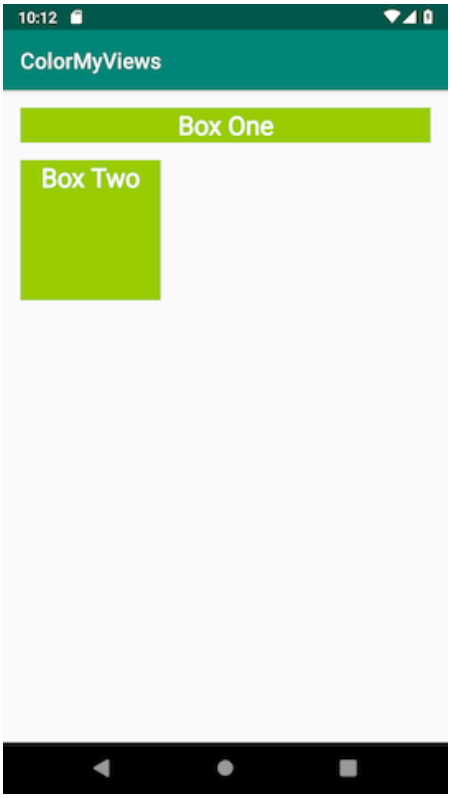
Attribute	value

`id`	`box_two_text`
`layout_height`	`130dp`
`layout_width`	`130dp`
`style`	`@style/whiteBox`
`text`	`@string/box_two`

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:

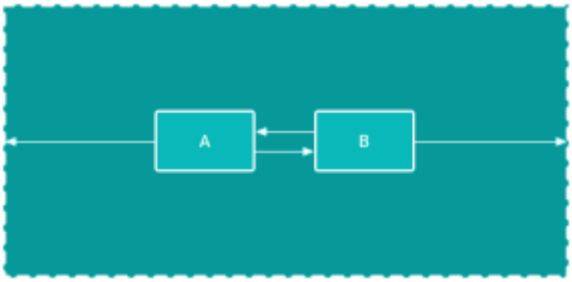


7. Task: Create a chain of TextView views

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

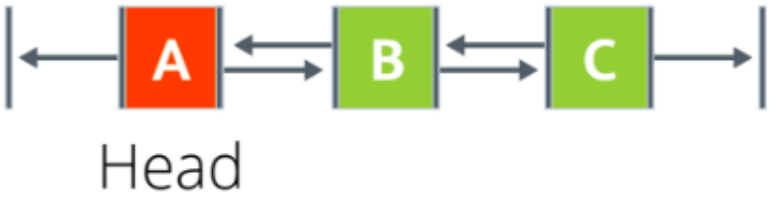
Chains

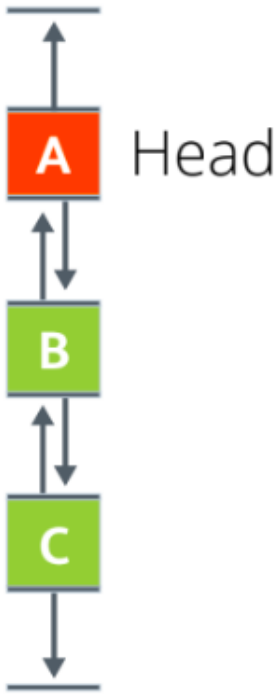
A chain is a group of views that are linked to each other with bidirectional constraints. The views within a chain can be distributed either vertically or horizontally. For example, the following diagram shows two views that are constrained to each other, which creates a horizontal chain.



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.

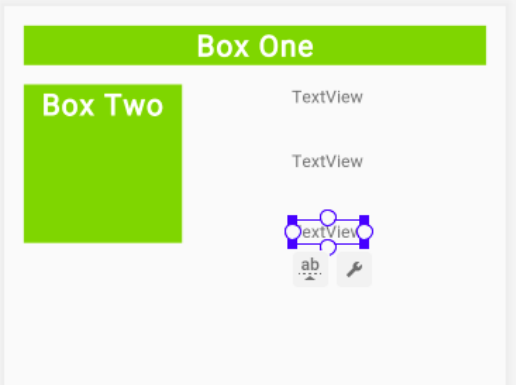
To add a chain style to a chain, set the `layout_constraintHorizontal_chainStyle` or the `layout_constraintVertical_chainStyle` attribute for the head of the chain. You can add chain styles in the Layout Editor, which you learn in this task.

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"
```

Step 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 **Box Two** text view, as shown below.

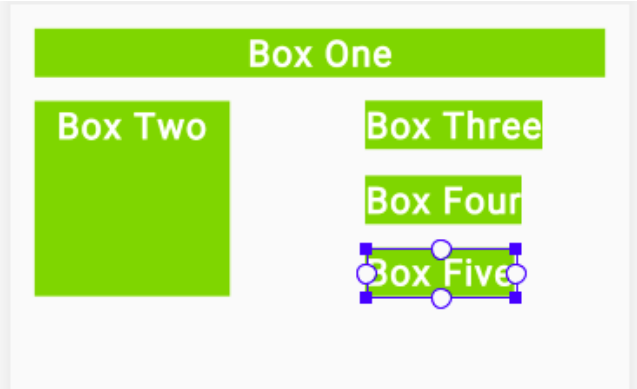


2. In the `strings.xml` file, add the following string resources for the names of the new text views:

```
<string name="box_three">Box Three</string>
<string name="box_four">Box Four</string>
<string name="box_five">Box Five</string>
```

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

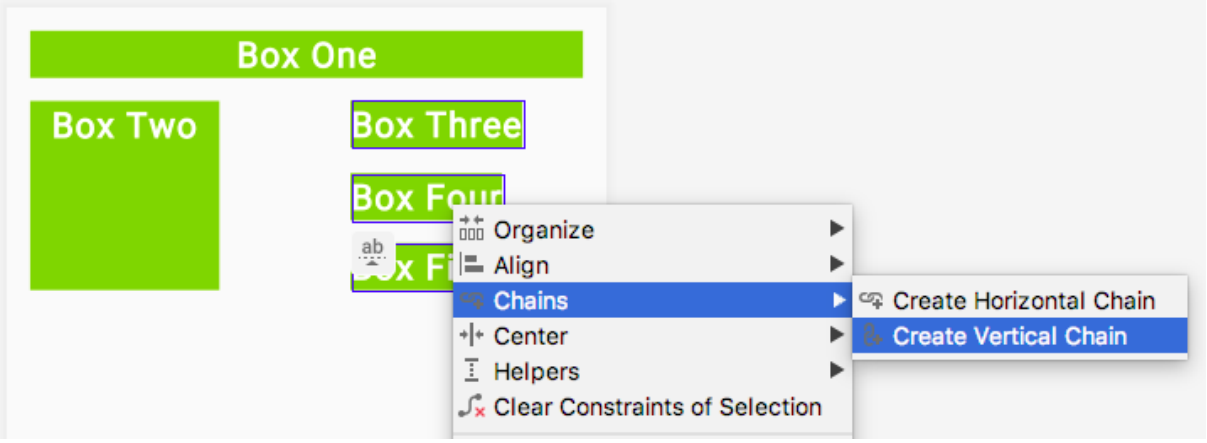
Attribute	Top text view	Middle text view	Bottom text view
ID	box_three_text	box_four_text	box_five_text
text	@string/box_three	@string/box_four	@string/box_five
style	@style/whiteBox	@style/whiteBox	@style/whiteBox



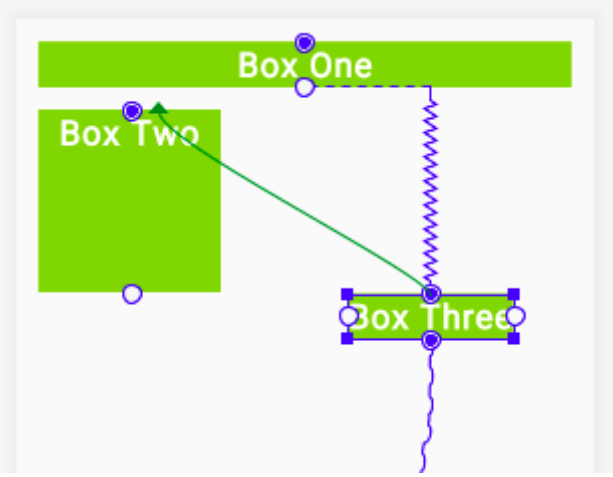
In the **Component Tree,** you see errors about missing constraints. You fix these errors later.

Step 2: Create a chain and constrain it to the height of Box Two

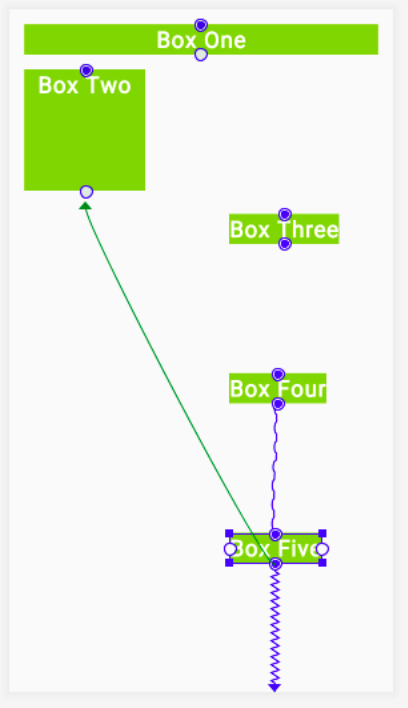
1. Select all three new text views, right-click, and select **Chains** > **Create Vertical Chain**.



This creates a vertical chain that extends from **Box One** to the end of the layout. 2. Add a constraint that extends from the top of **Box Three** to the top of **Box Two.** This removes the existing top constraint and replaces it with the new constraint. You don't have to delete the constraint explicitly.



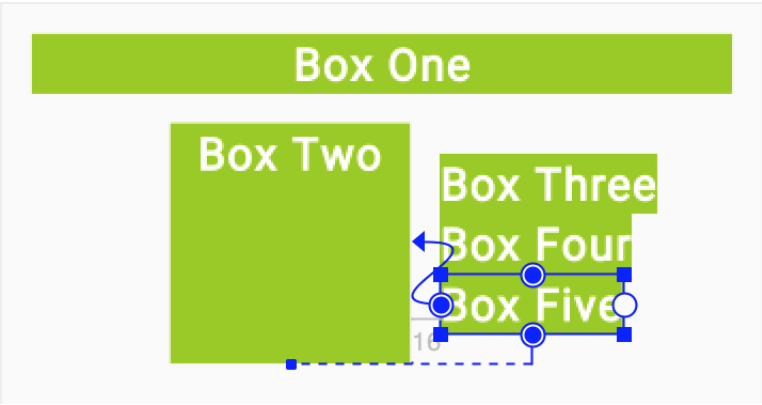
3. Add a constraint from the bottom of **Box Five** to the bottom of **Box Two**.



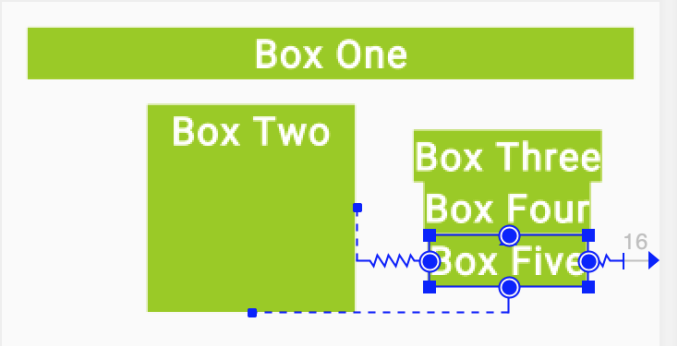
Observe that the three text views are now constrained to the top and bottom of **Box Two**.

Step 3: Add left and right constraints

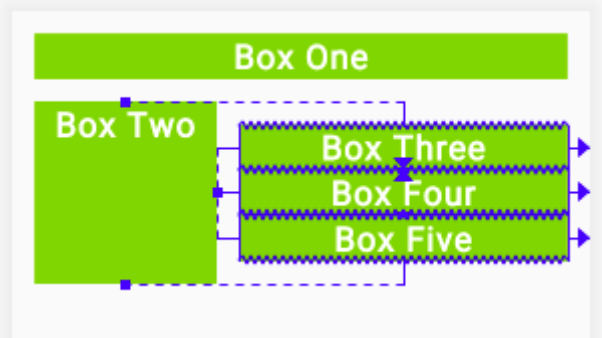
1. Constrain the left side of **Box Three** to the right side of **Box Two**. Repeat for **Box Four** and **Box Five**, constraining the left side of each to the right side of **Box Two**.



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



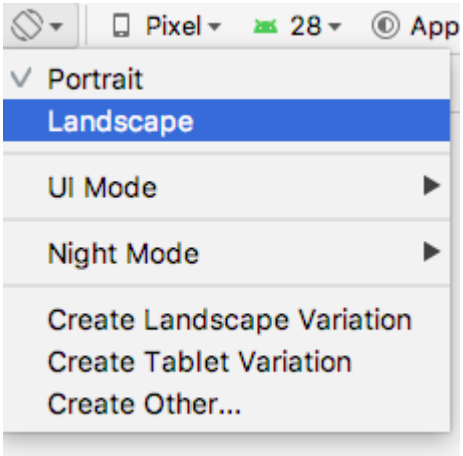
3. For each of the three text views, change the `layout_width` attribute `0dp`, which is equivalent to changing the constraint type to **Match Constraints**.



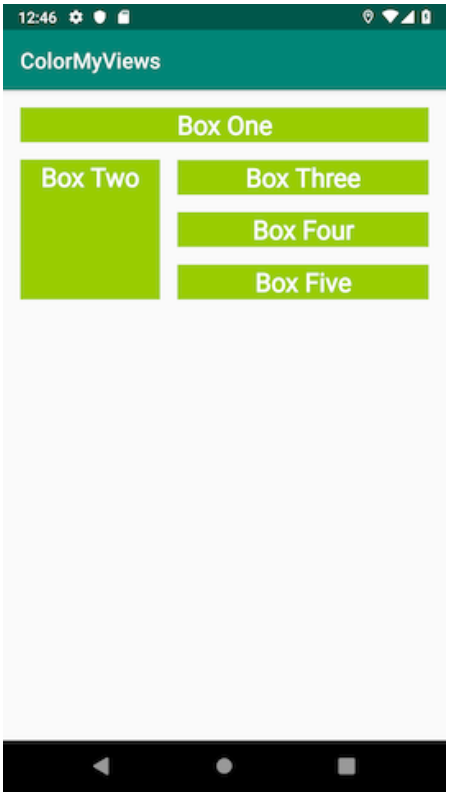
Step 4: Add margin

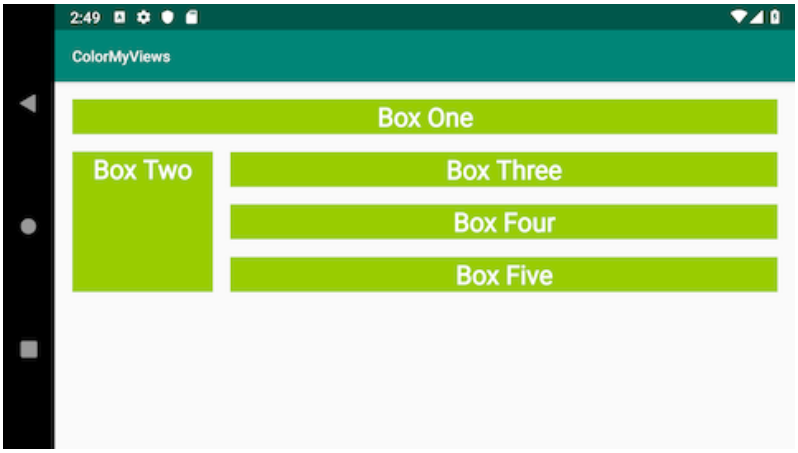
Use the **Attributes** pane to set **Layout_margin** attributes on the three text views to add spacing between them.

1. For **Box Three**, use `@dimen/margin_wide` for the **start** and **end** margins. Remove the other margins.
2. For **Box Four**, use `@dimen/margin_wide` for the **start**, **end**, **top**, and **bottom** margins. Remove the other margins.
3. For **Box Five**, 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.


12:46
ColorMyViews
Box One
Box Two
Box Three
Box Four
Box Five


2:49
ColorMyViews
Box One
Box Two
Box Three
Box Four
Box Five

8. Task: Add clickHandlers to the text views

In this task, you make the ColorMyViews 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 `whiteBox` 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()`. Use `View` as the function's parameter. This view is the one whose color will change.

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

Every view has a resource ID. The resource ID is the value assigned to the view's `id` attribute in the layout file, `activity_main.xml`. To set a color, the code will switch using a `when` statement on the view's resource ID. It's a common pattern to use one click-handler function for many views when the click action is the same.

3. Implement the `makeColored()` function: Add a `when` block 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.box_one_text -> view.setBackgroundColor(Color.DKGRAY)
        R.id.box_two_text -> view.setBackgroundColor(Color.GRAY)
        R.id.box_three_text -> view.setBackgroundColor(Color.BLUE)
        R.id.box_four_text -> view.setBackgroundColor(Color.MAGENTA)
        R.id.box_five_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 ConstraintLayout. 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 boxOneText = findViewById<TextView>(R.id.box_one_text)
    val boxTwoText = findViewById<TextView>(R.id.box_two_text)
    val boxThreeText = findViewById<TextView>(R.id.box_three_text)
    val boxFourText = findViewById<TextView>(R.id.box_four_text)
    val boxFiveText = findViewById<TextView>(R.id.box_five_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(boxOneText, boxTwoText, boxThreeText,
               boxFourText, boxFiveText, rootConstraintLayout)
  }
```

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

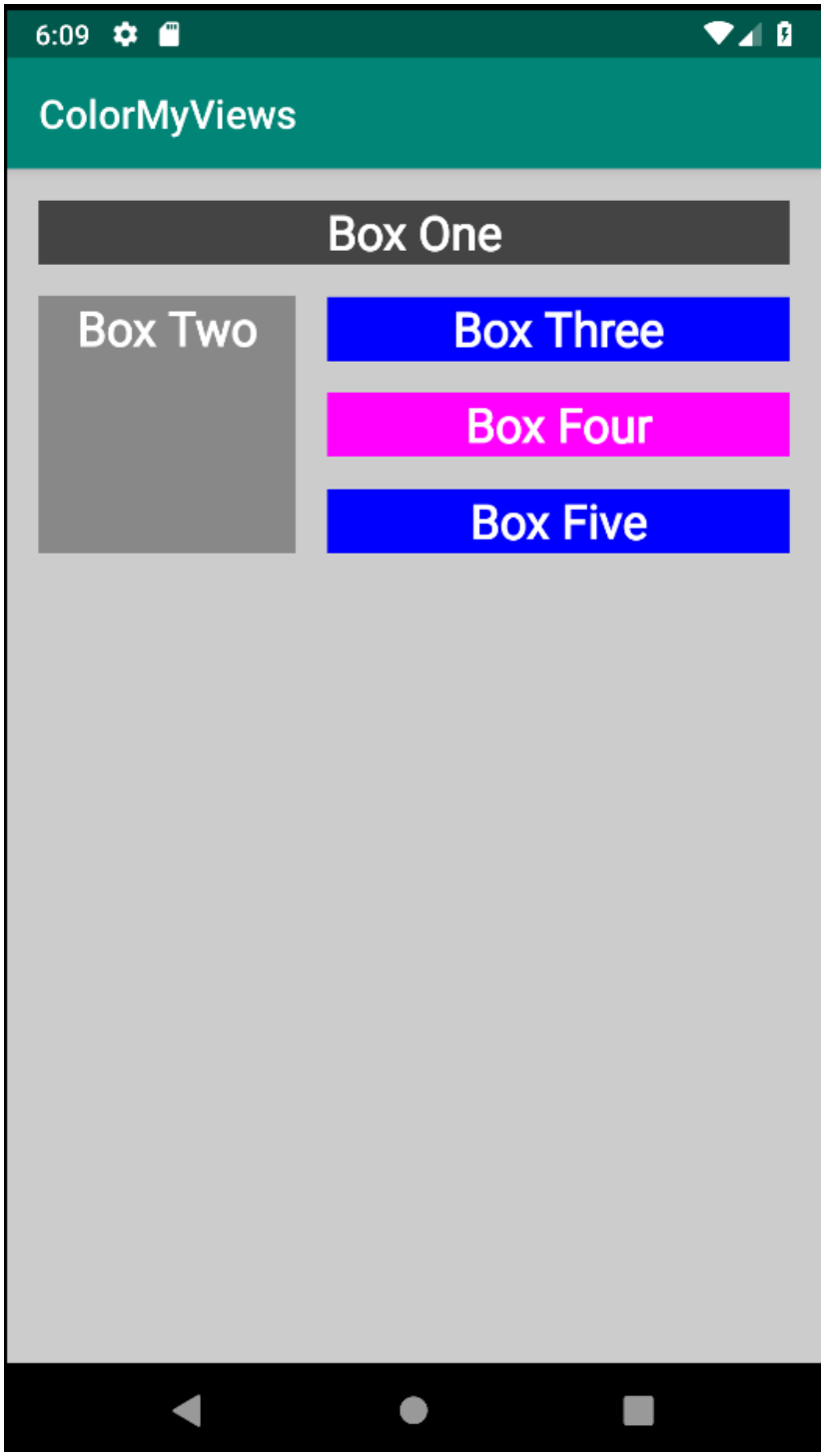
```
    for (item in clickableViews) {
        item.setOnClickListener { makeColored(it) }
```

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.


6:49
ColorMyViews


6:09
ColorMyViews
Box One
Box Two
Box Three
Box Four
Box Five

9. Coding challenge

Use images instead of colors and text as backgrounds for all the views. The app should reveal the images when the user taps the text views.

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

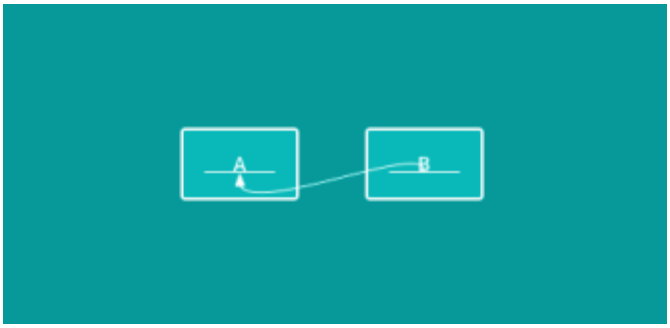
Example:

```
R.id.box_two_text -> view.setBackgroundResource(R.drawable.image_two)
```

10. Task: Add a baseline constraint

Baseline constraint

The baseline constraint aligns the baseline of a view's text with the baseline of another view's text. Aligning views that contain text can be a challenge, especially if the fonts are differently sized. Baseline constraint does the alignment work for you.



You can create baseline constraints in the Layout Editor by using the **Edit Baseline**  icon, which is displayed below the view when you hold the pointer over it. The equivalent XML code for the baseline constraint has the ConstraintLayout attribute, `layout_constraintBaseline_toBaselineOf` .

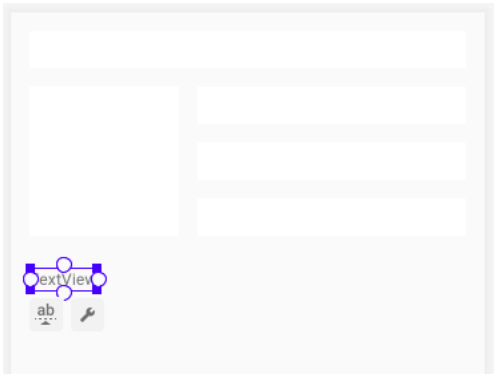
Sample XML code for the baseline constraint:

```
<Button
    android:id="@+id/buttonB"
    ...
    android:text="B"
    app:layout_constraintBaseline_toBaselineOf="@+id/buttonA" />
```

In this task, you add instructions that tell the user how to use the app. You create two `TextView` views, one for a label and one for the instructions information. The text views have different font sizes, and you align their baselines.

Step 1: Add a text view for the label

1. Open `activity_main.xml` in **Design** tab and drag a text view from the **Palette** pane into the design editor. Put the text view below **Box Two**. This text view will hold the text that labels the instructions.

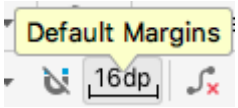


2. In `strings.xml` , create a string resource for the label `TextView` view.

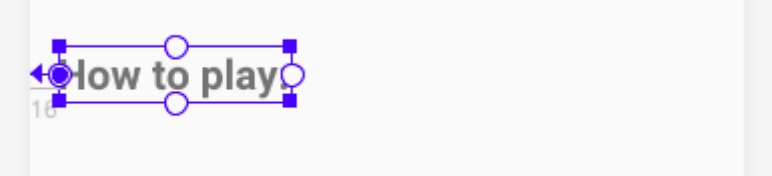
```
<string name="how_to_play">How to play:</string>
```

3. Use the **Attributes** pane to set the following attributes to the newly added label `TextView` :

Attribute	Value
ID	label_text
fontFamily	roboto
text	@string/how_to_play
textSize	24sp
textStyle	**B** *(bold)*

4. Use the **Default Margins**


icon in the toolbar to set the default margins to `16dp`.
5. Constrain the `label_text` view's left side to the layout's parent element.



6. In the `activity_main.xml` file, the Layout Editor adds the `layout_marginStart` attribute with a hard-coded value of `16dp`. Replace `16dp` with `@dimen/margin_wide`. The XML code now looks similar to this:

```
<TextView
    android:id="@+id/label_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_wide"
    android:fontFamily="@font/roboto"
    android:text="@string/how_to_play"
    android:textSize="24sp"
    android:textStyle="bold"
    app:layout_constraintStart_toStartOf="parent"
    tools:layout_editor_absoluteY="287dp"/> <!--Designtime attribute-->
```

Design-time attributes

Design-time attributes are used and applied only during the layout design, not at runtime. When you run the app, the design-time attributes are ignored.

Design-time attributes are prefixed with the `tools` namespace, for example, `tools:layout_editor_absoluteY` in the generated code snippet shown above. This line of code is added because you haven't yet added a vertical constraint.

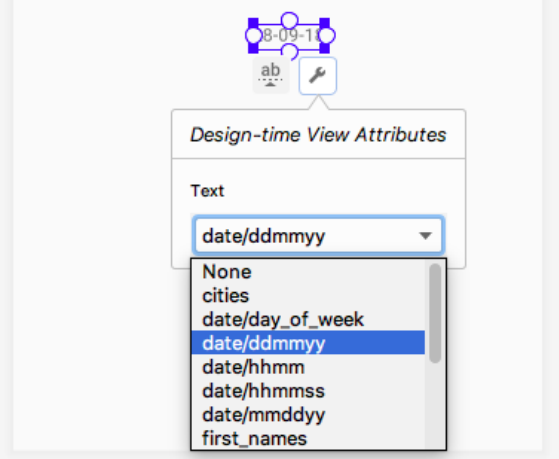
All views in a `ConstraintLayout` need to be constrained horizontally and vertically, or else the views jump to an edge of the parent when you run the app. This is why the Layout Editor adds `tools:layout_editor_absoluteX` if the view is not horizontally constrained. Layout Editor gives the design-time attribute the value of the view's current position in the layout, to keep the views in place during design. You can safely ignore these `tools` attributes, because Android Studio removes them after you create the constraints.

Using design-time attributes, you can also add sample preview data to a text view or image view from within the Layout Editor.

Try experimenting with sample data:

1. Add a new text view to your layout.
2. In the design editor, hold the pointer over the new view. The **constraint_layout** icon

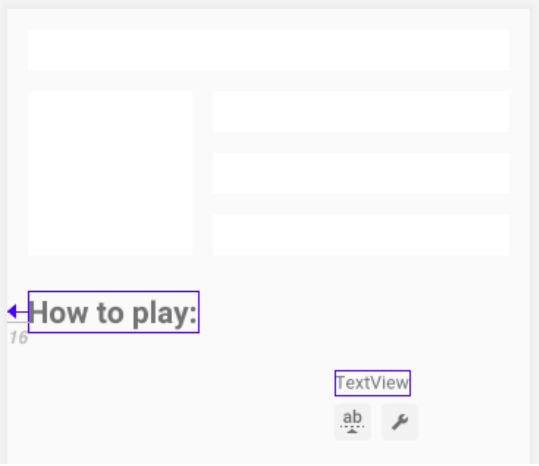
   appears below the view. Click the icon to display the **Design-time View Attributes** drop-down menu, as shown below:



3. Select a type of the sample data from the drop-down list. For example, if you set text sample data to **date/mmddyy,** today's date is displayed in the design.
4. Delete the text view that you just created.

Step 2: Add a text view for the info text

1. Drag another text view from the **Palette** pane into the design editor. Place the view next to and below the `label_text` text view, as shown below. This new text view is for the help info that the user will see. Make sure that the new view is vertically offset from the `label_text` view, so that you can see what happens when you create the baseline constraint.



2. In `strings.xml`, create a string resource for the new text view.

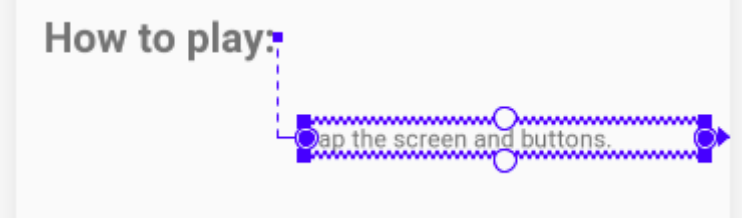
```
<string name="tap_the_boxes_and_buttons">Tap the screen and buttons.</string>
```

3. Use the **Attributes** pane to set the following attributes to the new text view:

Attribute	Value
`ID`	`info_text`
`layout_width`	`0dp` (which is equivalent to `match_constraint`)
`fontFamily`	`roboto`

`text`	`@string/tap_the_boxes_and_buttons`

4. Constrain the right side of the `info_text` to the right edge of the parent element. Constrain the left side of the `info_text` to the right (end) of the `label_text`.

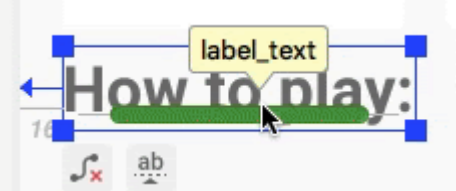


Step 3: Align the baselines of the two text views

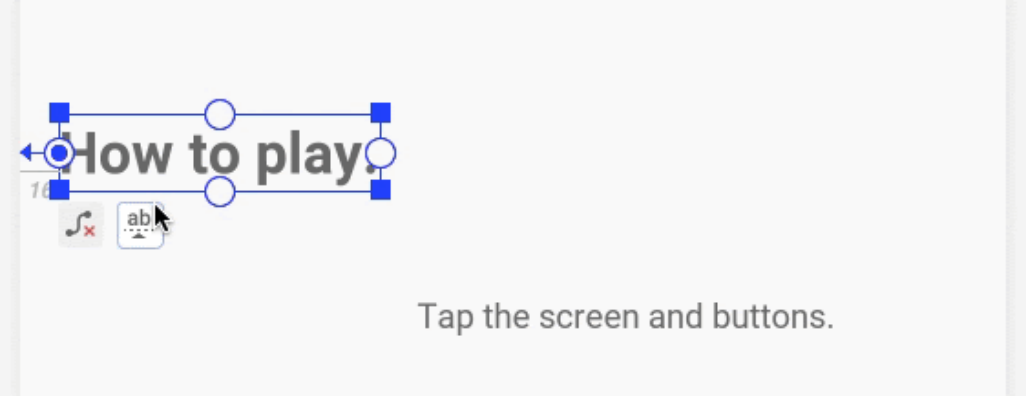
1. Click the `label_text`. The **Edit Baseline** icon

appears below the view. Click the

icon. (The view might jump to a new location in your layout.)
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 info text view.



Tip

To delete all the constraints that are set on a view:

1. In the **Design** tab, click on the view.
2. Click the **Delete Constraints** icon

that appears below the view.

To delete a particular constraint:

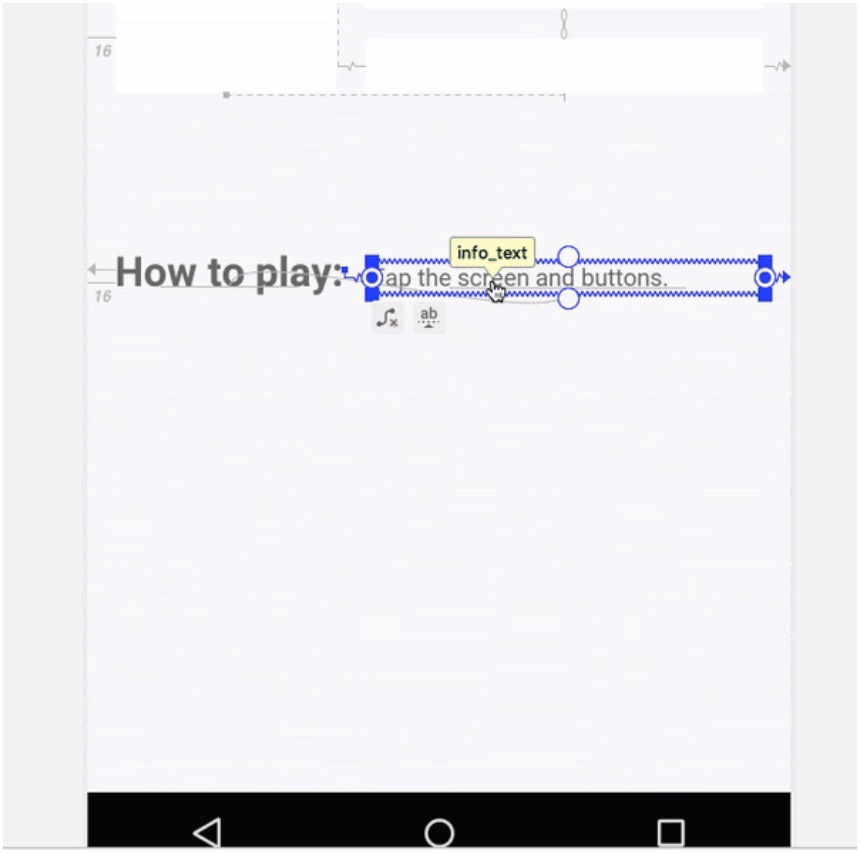
1. Hold the pointer over the constraint handle for the constraint you want to delete. The constraint handle (the dot) turns red.

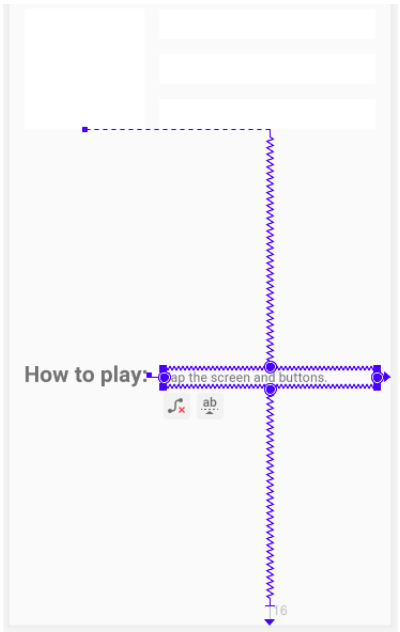
2. Click the red constraint handle.

Step 4: Add vertical constraints to the two text views

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 bottom of the `info_text` to the bottom of the layout.
2. Attach the top of the `info_text` to the bottom of the `box_two_text`.

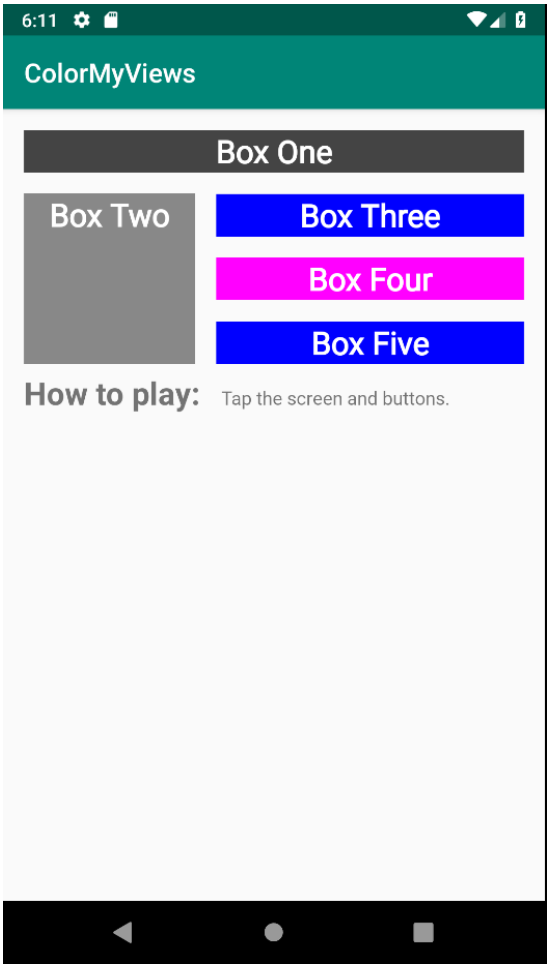




Try moving the `info_text` view up or down. Notice that the `label_text` view follows, and stays aligned at the baseline. 3. In the view inspector, change the vertical bias of the `info_text` view to `0`. This keeps the text views closer to the top constrained view, **Box Two**. (If the view inspector isn't visible in the **Attributes** pane when you click on the view in the design editor, close and reopen Android Studio.) 4. The generated XML code should look similar to this:

```
<TextView
    android:id="@+id/label_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_wide"
    android:text="@string/how_to_play"
    android:textSize="24sp"
    android:textStyle="bold"
    app:layout_constraintBaseline_toBaselineOf="@+id/info_text"
    app:layout_constraintStart_toStartOf="parent" />

<TextView
    android:id="@+id/info_text"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_wide"
    android:layout_marginTop="@dimen/margin_wide"
    android:layout_marginEnd="@dimen/margin_wide"
    android:layout_marginBottom="@dimen/margin_wide"
    android:text="@string/tap_the_boxes_and_buttons"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toEndOf="@+id/label_text"
    app:layout_constraintTop_toBottomOf="@+id/box_two_text"
    app:layout_constraintVertical_bias="0.0" />
```

5. Run your app. Your screen should look similar to the screenshot below.

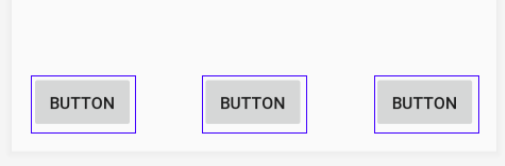

6:11
ColorMyViews
Box One
Box Two
Box Three
Box Four
Box Five
How to play:
Tap the screen and buttons.

11. Task: Add a chain of buttons

In this task, you add three `Button` views and chain them together.

Step 1: Add three buttons

1. Open the `activity_main.xml` file in the **Design** tab. Drag three buttons from the **Palette** pane onto the bottom of the layout.



2. In the `strings.xml` file, add the following string resources for the `Button` views:

```
<string name="button_red">RED</string>
<string name="button_yellow">YELLOW</string>
<string name="button_green">GREEN</string>
```

3. Set the following attributes to the button views:

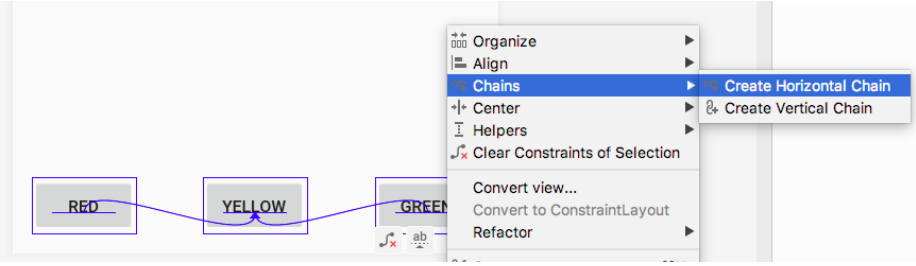
Attribute	Left button	Middle button	Right button
`ID`	`red_button`	`yellow_button`	`green_button`
`text`	`@string/button_red`	`@string/button_yellow`	`@string/button_green`

4. Align the button labels vertically with each other. To do this, constrain the baselines of `red_button` and `green_button` to the baseline of the `yellow_button`. (To add a baseline constraint to a view, click on the view and use the **Edit Baseline** icon  that appears below the view.)

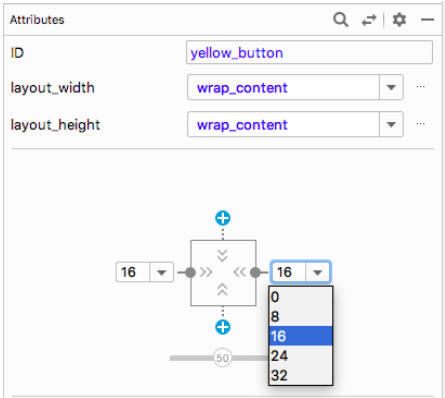
> **Tip:** Baseline constraint and the bottom constraint are mutually exclusive, so you can't create both for the same view. If you add a bottom constraint and then a baseline constraint, the Layout Editor removes the bottom constraint.

Step 2: Create a horizontal chain and constrain it

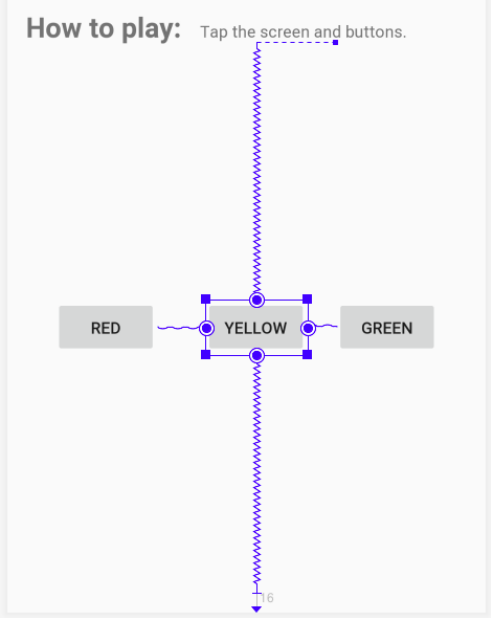
1. In the design editor or in the **Component Tree** pane select all three button views and right-click. Select **Chains > Create Horizontal chain**.



2. Use the view inspector to set right and left margins of `16dp` for the `yellow_button`, if these margins aren't already set.



3. Using the view inspector, set the left margin of the `red_button` to `16dp`. Set the right margin of the `green_button` to `16dp`.
4. Constrain the top of the `yellow_button` to the bottom of the `info_text`.
5. Constrain the bottom of the `yellow_button` to the bottom of the layout.



6. Change the vertical bias of the `yellow_button` to `100` (`1.0` in the XML), to drop down the buttons to the bottom of the layout.
7. Test your layout for different devices and orientations. The layout may not work for all devices and orientations, but should work for most of them.

The generated XML code for the `Button` views will be similar to the following:

```
<Button
    android:id="@+id/red_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:layout_marginStart="@dimen/margin_wide"
    android:text="@string/button_red"
    android:visibility="visible"
    app:layout_constraintBaseline_toBaselineOf="@+id/yellow_button"
    app:layout_constraintEnd_toStartOf="@+id/yellow_button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent" />

<Button
    android:id="@+id/yellow_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_wide"
    android:layout_marginTop="@dimen/margin_wide"
    android:layout_marginBottom="@dimen/margin_wide"
    android:text="@string/button_yellow"
    android:visibility="visible"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/green_button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/red_button"
    app:layout_constraintTop_toBottomOf="@+id/info_text"
    app:layout_constraintVertical_bias="1.0" />

<Button
    android:id="@+id/green_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="16dp"
    android:text="@string/button_green"
    app:layout_constraintBaseline_toBaselineOf="@+id/yellow_button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/yellow_button" />
```

12. Task: Add click handlers to the buttons

In this task, you add a click handler to each `Button` view. The click handler changes the color of the `TextView` views.

1. Add the following colors to the `res/values/colors.xml` file:

```
<color name="my_green">#12C700</color>
<color name="my_red">#E54304</color>
<color name="my_yellow">#FFC107</color>
```

2. In `MainActivity.kt`, use `findViewById` to get references for the button views. To do this, put the following code inside the `setListeners()` click-handler function, above the `clickableViews` declaration:

```
val redButton = findViewById<Button>(R.id.red_button)
val greenButton = findViewById<Button>(R.id.green_button)
val yellowButton = findViewById<Button>(R.id.yellow_button)
```

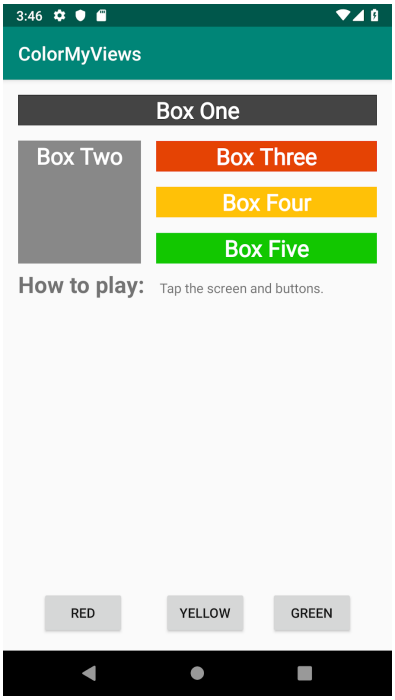
3. Inside `setListeners()`, add the references of the `Button` views to the list of clickable views.

```
private fun setListeners() {
    ...
    val clickableViews: List<View> =
        listOf(boxOneText, boxTwoText, boxThreeText,
boxFourText, boxFiveText, rootConstraintLayout,
redButton, greenButton, yellowButton
)
    ...
}
```

4. Inside the `makeColored()` function, add code to change the text views' colors when the user taps the buttons. Use the `setBackgroundResource()` function to set a custom color, defined in the resources as a view's background. Use the `setBackgroundColor()` function to set predefined colors as view's background Add the new code above the `else` statement, as shown:

```
private fun makeColored(view: View) {
    when (view.id) {

        ...

        // Boxes using custom colors for background
        R.id.red_button -> box_three_text.setBackgroundResource(R.color.my_red)
        R.id.yellow_button -> box_four_text.setBackgroundResource(R.color.my_yellow)
        R.id.green_button -> box_five_text.setBackgroundResource(R.color.my_green)

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

5. Run your final app. Click on the text views and the buttons. Your screen will look something like this.


3:46
ColorMyViews
Box One
Box Two
Box Three
Box Four
Box Five
How to play:
Tap the screen and buttons.
RED
YELLOW
GREEN

Android Kotlin Fundamentals 02.4: Data binding basics

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

Introduction

In previous codelabs in this course, you used the `findViewById()` function to get references to views. When your app has complex view hierarchies, `findViewById()` is expensive and slows down the app, because Android traverses the view hierarchy, starting at the root, until it finds the desired view. Fortunately, there's a better way.

To set data in views, you've used string resources and set the data from the activity. It would be more efficient if the view knew about the data. And fortunately again, this is possible.

In this codelab, you learn how to use data binding to eliminate the need for `findViewById()`. You also learn how to use data binding to access data directly from a view.

What you should already know

You should be familiar with:

- What an activity is, and how to set up an activity with a layout in `onCreate()`.
- Creating a text view and setting the text that the text view displays.
- Using `findViewById()` to get a reference to a view.
- Creating and editing a basic XML layout for a view.

What you'll learn

How to use the Data Binding Library to eliminate inefficient calls to `findViewById()`.

How to access app data directly from XML.

What you'll do

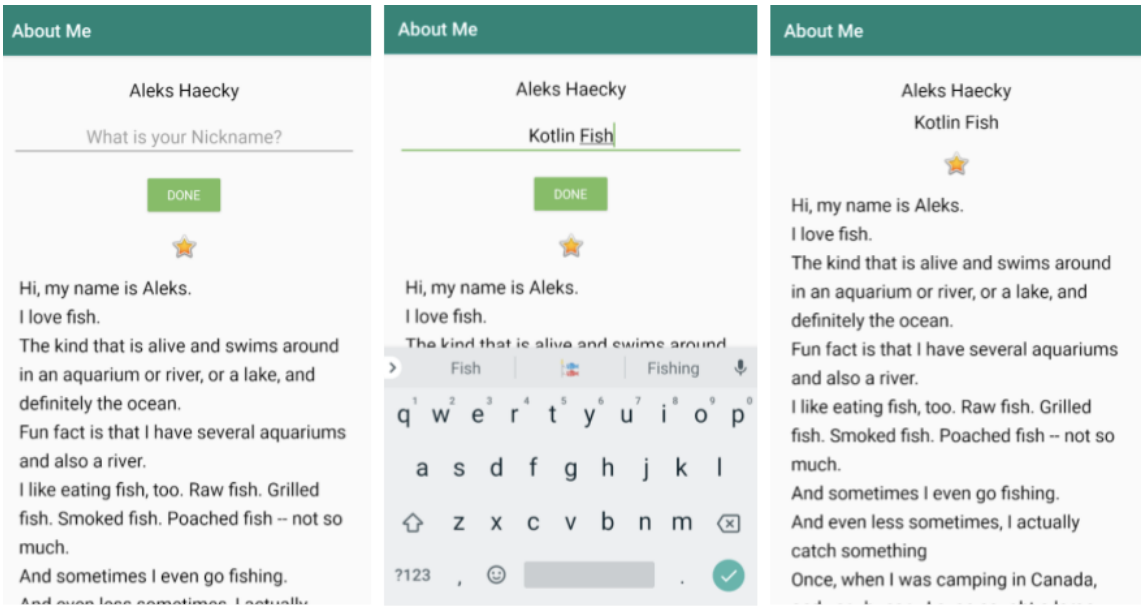
- Modify an app to use data binding instead of `findViewById()`, and to access data directly from the layout XML files.

2. App overview

In this codelab, you start with the AboutMe app and change the app to use data binding. The app will look exactly the same when you are done!

Here's what the AboutMe app does:

- When the user opens the app, the app shows a name, a field to enter a nickname, a **Done** button, a star image, and scrollable text.
- The user can enter a nickname and tap the **Done** button. The editable field and button are replaced by a text view that shows the entered nickname.



You can use the code you built in the previous codelab, or you can download the AboutMeDataBinding-Startercode from GitHub.

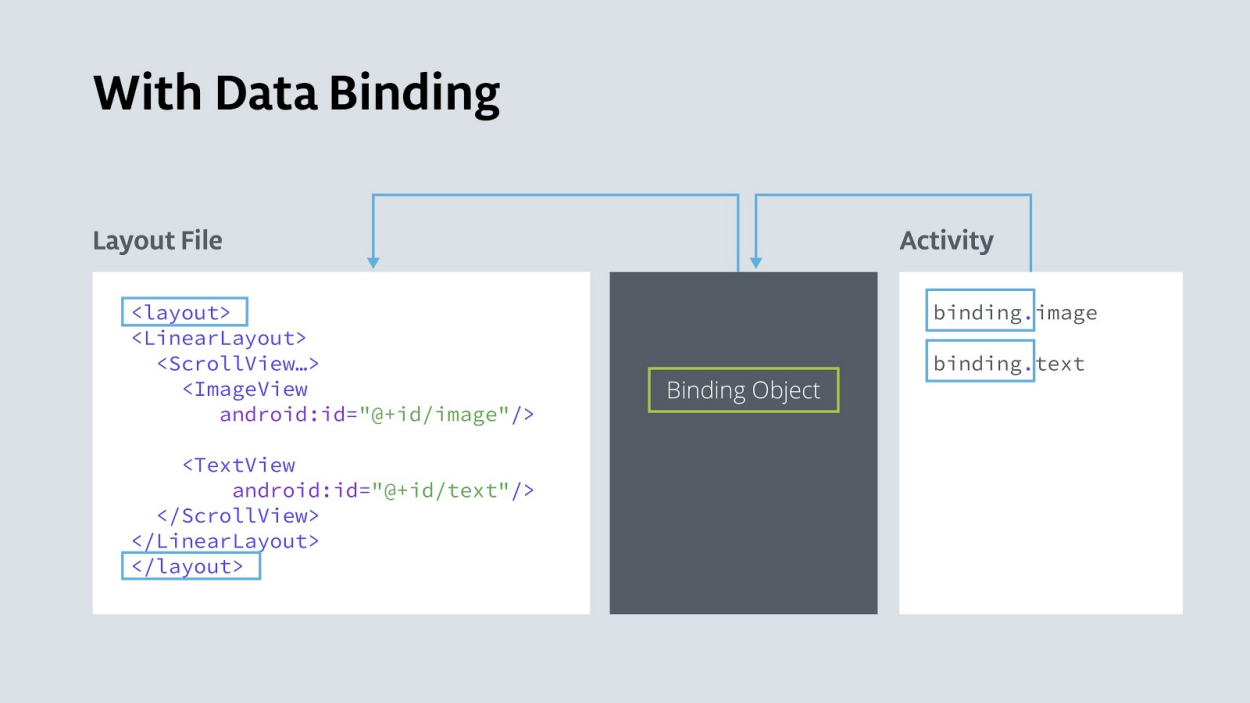
3. Task: Use data binding to eliminate findViewById()

The code you wrote in previous codelabs uses the `findViewById()` function to obtain references to views.

Every time you use `findViewById()` to search for a view after the view is created or recreated, the Android system traverses the view hierarchy at runtime to find it. When your app has only a handful of views, this is not a problem. However, production apps may have dozens of views in a layout, and even with the best design, there will be nested views.

Think of a linear layout that contains a scroll view that contains a text view. For a large or deep view hierarchy, finding a view can take enough time that it can noticeably slow down the app for the user. Caching views in variables can help, but you still have to initialize a variable for each view, in each namespace. With lots of views and multiple activities, this adds up, too.

One solution is to create an object that contains a reference to each view. This object, called a `Binding` object, can be used by your whole app. This technique is called *data binding*. Once a binding object has been created for your app, you can access the views, and other data, through the binding object, without having to traverse the view hierarchy or search for the data.



Data binding has the following benefits:

- Code is shorter, easier to read, and easier to maintain than code that uses `findViewById()`.
- Data and views are clearly separated. This benefit of data binding becomes increasingly important later in this course.
- The Android system only traverses the view hierarchy once to get each view, and it happens during app startup, not at runtime when the user is interacting with the app.
- You get type safety for accessing views. (*Type safety* means that the compiler validates types while compiling, and it throws an error if you try to assign the wrong type to a variable.)

In this task, you set up data binding, and you use data binding to replace calls to `findViewById()` with calls to the binding object.

Step 1: Enable data binding

To use data binding, you need to enable data binding in your Gradle file, as it's not enabled by default. This is because data binding increases compile time and may affect app startup time.

1. If you do not have the AboutMe app from a previous codelab, get the AboutMeDataBinding-Starter code from GitHub. Open it in Android Studio.

2. Open the `build.gradle (Module: app)` file.
3. Inside the `android` section, before the closing brace, add a `buildFeatures` section and set `dataBinding` to `true`.

```
buildFeatures {
    dataBinding true
}
```

4. When prompted, **Sync** the project. If you're not prompted, select **File > Sync Project with Gradle Files**.
5. You can run the app, but you won't see any changes.

Step 2: Change layout file to be usable with data binding

To work with data binding, you need to wrap your XML layout with a `<layout>` tag. This is so that the root class is no longer a view group, but is instead a layout that contains view groups and views. The binding object can then know about the layout and the views in it.

1. Open the `activity_main.xml` file.
2. Switch to the **Text** tab.
3. Add `<layout></layout>` as the outermost tag around the `<LinearLayout>`.

```
<layout>
    <LinearLayout ... >
    ...
    </LinearLayout>
</layout>
```

4. Choose **Code > Reformat code** to fix the code indentation.

The namespace declarations for a layout must be in the outermost tag.

5. Cut the namespace declarations from the `<LinearLayout>` and paste them into the `<layout>` tag. Your opening `<layout>` tag should look as shown below, and the `<LinearLayout>` tag should only contain view properties.

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
```

6. Build and run your app to verify that you did this correctly.

Step 3: Create a binding object in the main activity

Add a reference to the binding object to the main activity, so that you can use it to access views:

1. Open the `MainActivity.kt` file.
2. Before `onCreate()`, at the top level, create a variable for the binding object. This variable is customarily called `binding`.

The type of `binding`, the `ActivityMainBinding` class, is created by the compiler specifically for this main activity. The name is derived from the name of the layout file, that is, `activity_main + Binding`.

```
private lateinit var binding: ActivityMainBinding
```

3. If prompted by Android Studio, import `ActivityMainBinding`. If you aren't prompted, click on `ActivityMainBinding` and press `Alt+Enter` (`Option+Enter` on a Mac) to import this missing class.

(For more keyboard shortcuts, see Keyboard shortcuts.)

The `import` statement should look similar to the one shown below.

```
import com.example.android.aboutme.databinding.ActivityMainBinding
```

Next, you replace the current `setContentView()` function with an instruction that does the following:

- Creates the binding object.
- Uses the `setContentView()` function from the `DataBindingUtil` class to associate the `activity_main` layout with the `MainActivity`. This `setContentView()` function also takes care of some data binding setup for the views.

4. In `onCreate()`, replace the `setContentView()` call with the following line of code.

```
binding = DataBindingUtil.setContentView(this, R.layout.activity_main)
```

5. Import `DataBindingUtil`.

```
import androidx.databinding.DataBindingUtil
```

Step 4: Use the binding object to replace all calls to findViewById()

You can now replace all calls to `findViewById()` with references to the views that are in the binding object. When the binding object is created, the compiler generates the names of the views in the binding object from the IDs of the views in the layout, converting them to camel case. So, for example, `done_button` is `doneButton` in the binding object, `nickname_edit` becomes becomes `nicknameEdit`, and `nickname_text` becomes `nicknameText`.

1. In `onCreate()`, replace the code that uses `findViewById()` to find the `done_button` with code that references the button in the binding object.

Replace this code: `findViewById<Button>(R.id. done_button )` with: `binding.doneButton`

Your finished code to set the click listener in `onCreate()` should look like this.

```
binding.doneButton.setOnClickListener {
    addNickname(it)
}
```

2. Do the same for all calls to `findViewById()` in the `addNickname()` function. Replace all occurrences of `findViewById< View >(R.id. id_view )` with `binding. idView`. Do this in the following way:

- Delete the definitions for the `editText` and `nicknameTextView` variables along with their calls to `findViewById()`. This will give you errors.
- Fix the errors by getting the `nicknameText`, `nicknameEdit`, and `doneButton` views from the `binding` object instead of the (deleted) variables.
- Replace `view.visibility` with `binding.doneButton.visibility`. Using `binding.doneButton` instead of the passed-in `view` makes the code more consistent.

The result is the following code:

```
binding.nicknameText.text = binding.nicknameEdit.text
binding.nicknameEdit.visibility = View.GONE
binding.doneButton.visibility = View.GONE
binding.nicknameText.visibility = View.VISIBLE
```

- There is no change in functionality. Optionally, you can now eliminate the `view` parameter and update all uses of `view` to use `binding.doneButton` inside this function.

3. The `nicknameText` requires a `String`, and `nicknameEdit.text` is an `Editable`. When using data binding, it is necessary to explicitly convert the `Editable` to a `String`.

```
binding.nicknameText.text = binding.nicknameEdit.text.toString()
```

4. You can delete the grayed out imports.

Kotlinize the function by using `apply{}`.

```
binding.apply {
    nicknameText.text = nicknameEdit.text.toString()
    nicknameEdit.visibility = View.GONE
    doneButton.visibility = View.GONE
    nicknameText.visibility = View.VISIBLE
}
```

5. Build and run your app...and it should look and work exactly the same as before.

Tip: If you see compiler errors after you make changes, select **Build > Clean Project** followed by **Build > Rebuild Project**. Doing this usually updates the generated files. Otherwise, select **File > Invalidate Caches/Restart** to do a more thorough cleanup.

Tip: You previously learned about the `Resources` object that holds references to all resources in the app. You can think of the `Binding` object in a similar fashion when referencing views; however, the `Binding` object is a lot more sophisticated.

4. Task: Use data binding to display data

You can take advantage of data binding to make a data class directly available to a view. This technique simplifies the code, and is extremely valuable for handling more complex cases.

For this example, instead of setting the name and nickname using string resources, you create a data class for the name and nickname. You make the data class available to the view using data binding.

Step 1: Create the MyName data class

1. In Android Studio in the `java` directory, open the `MyName.kt` file. If you don't have this file, create a new Kotlin file and call it `MyName.kt`.
2. Define a data class for the name and nickname. Use empty strings as the default values.

```
data class MyName(var name: String = "", var nickname: String = "")
```

Step 2: Add data to the layout

In the `activity_main.xml` file, the name is currently set in a `TextView` from a string resource. You need to replace the reference to the name with a reference to data in the data class.

1. Open `activity_main.xml` in the **Text** tab.
2. At the top of the layout, between the `<layout>` and `<LinearLayout>` tags, insert a `<data></data>` tag. This is where you will connect the view with the data.

```
<data>

</data>
```

Inside the data tags, you can declare named variables that hold a reference to a class.

3. Inside the `<data>` tag, add a `<variable>` tag.
4. Add a `name` parameter to give the variable a name of `"myName"`. Add a `type` parameter and set the type to a fully qualified name of the `MyName` data class (package name + variable name).

```
<variable
        name="myName"
        type="com.example.android.aboutme.MyName" />
```

Now, instead of using the string resource for the name, you can reference the `myName` variable.

> **Note**: Depending on how you set up the project in the project wizard, your package name may differ. Make sure the package name of your app matches with the package name for the `type`variable.

5. Replace `android:text="@string/name"` with the code below.

`@={}` is a directive to get the data that is referenced inside the curly braces.

`myName` references the `myName` variable that you previously defined, which points to the `myName` data class and fetches the `name` property from the class.

```
android:text="@={myName.name}"
```

Step 3: Create the data

You now have a reference to the data in your layout file. Next, you create the actual data.

1. Open the `MainActivity.kt` file.
2. Above `onCreate()`, create a private variable, also called `myName` by convention. Assign the variable an instance of the `MyName` data class, passing in the name.

```
private val myName: MyName = MyName("Aleks Haecky")
```

3. In `onCreate()`, set the value of the `myName` variable in the layout file to the value of the `myName` variable that you just declared. You can't access the variable in the XML directly. You need to access it through the binding object.

```
binding.myName = myName
```

4. This may show an error, because you need to refresh the binding object after making changes. Build your app, and the error should go away.

Step 4: Use the data class for the nickname in the TextView

The final step is to also use the data class for the nickname in the `TextView`.

1. Open `activity_main.xml`.
2. In the `nickname_text` text view, add a `text` property. Reference the `nickname` in the data class, as shown below.

```
android:text="@={myName.nickname}"
```

3. In `MainActivity`, replace `nicknameText.text = nicknameEdit.text.toString()` with code to set the nickname in the `myName` variable.

```
myName?.nickname = nicknameEdit.text.toString()
```

After the nickname is set, you want your code to refresh the UI with the new data. To do this, you must invalidate all binding expressions so that they are recreated with the correct data.

4. Add `invalidateAll()` after setting the nickname so that the UI is refreshed with the value in the updated binding object.

```
binding.apply {
    myName?.nickname = nicknameEdit.text.toString()
    invalidateAll()
    ...
}
```

5. Build and run your app, and it should work exactly the same as before.