UNIT 1 : MODULE 2

Project templates

In Android Studio, a project template is an Android app that has all the necessary parts, but doesn't do much. The purpose is to help you get started faster and save you some work. Some examples of templates in Android Studio are an app with a map, and an app with multiple screens.

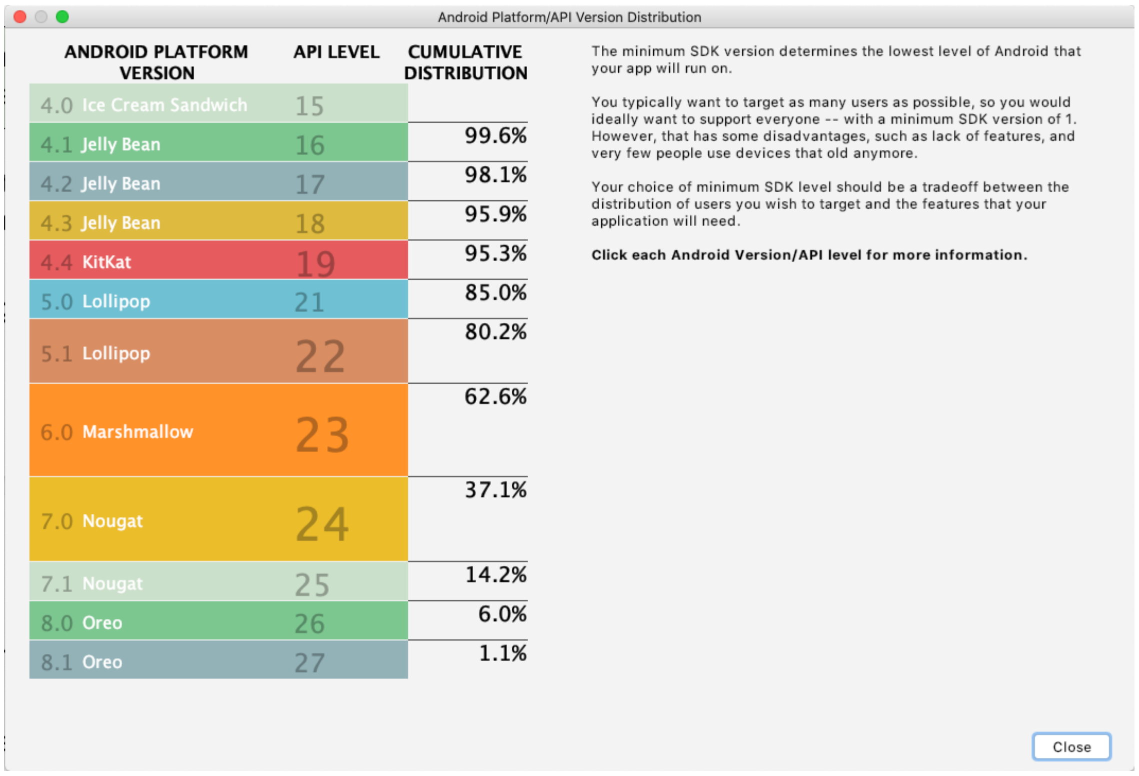
There are templates for many different types of devices (such as phones, tablets, and watches), and different types of apps (apps with scrollable screens, apps with maps, and apps with fancy navigation).

The **Empty Activity** template is the simplest template that can be used to create an app. It has a single screen and displays a simple "Hello World!" message.

Minimum SDK indicates the minimum version of Android that your app can run on. Select `API 19: Android 4.4 (KitKat)` from the dropdown list.

Note: There are many different versions of the Android operating system, each of which is given a name in alphabetical order as it is released.

Below **Minimum SDK**, notice the informational note on how many devices your app can run on with the chosen API level. If you are curious, click the **Help me choose** link to display a list of different versions of Android, as shown below. Then return to the **New Project** window.



ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.0 Ice Cream Sandwich	15	
4.1 Jelly Bean	16	99.6%
4.2 Jelly Bean	17	98.1%
4.3 Jelly Bean	18	95.9%
4.4 KitKat	19	95.3%
5.0 Lollipop	21	85.0%
5.1 Lollipop	22	80.2%
6.0 Marshmallow	23	62.6%
7.0 Nougat	24	37.1%
7.1 Nougat	25	14.2%
8.0 Oreo	26	6.0%
8.1 Oreo	27	1.1%

The minimum SDK version determines the lowest level of Android that your app will run on.

You typically want to target as many users as possible, so you would ideally want to support everyone -- with a minimum SDK version of 1. However, that has some disadvantages, such as lack of features, and very few people use devices that old anymore.

Your choice of minimum SDK level should be a tradeoff between the distribution of users you wish to target and the features that your application will need.

Click each Android Version/API level for more information.

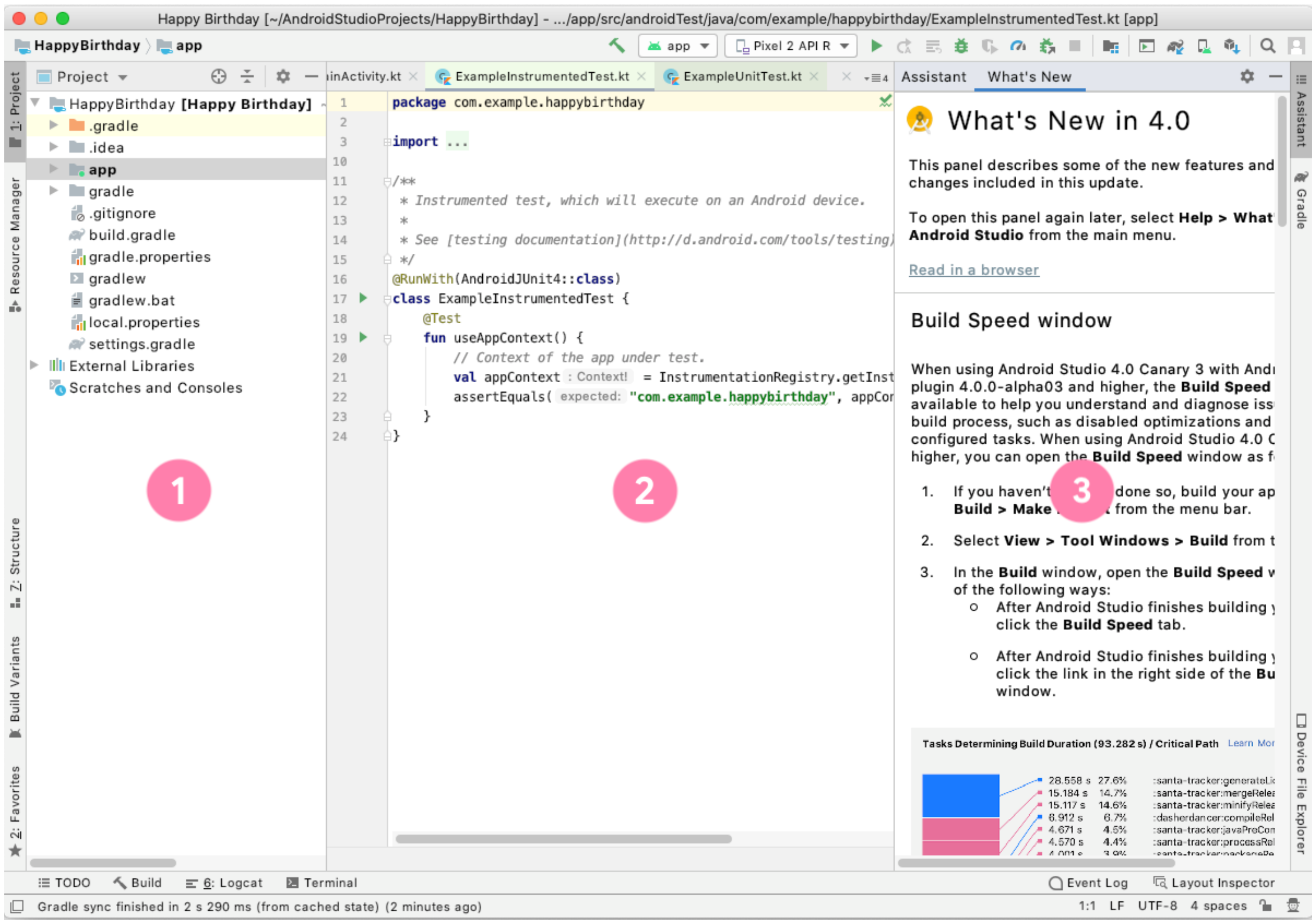
In the **Create New Project** window, make sure **Use legacy android.support libraries** is unchecked. Click the question mark if you want to know more about this.

When you first open Android Studio, you'll see three windows:

(1) The **Project** window shows the files and folders of your project.

(2) The **Editing** window is for editing code.

(3) The **What's New** window shows news and useful tips.



```
package com.example.happybirthday

import ...

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing)
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext : Context!  = InstrumentationRegistry.getInst
        assertEquals( expected: "com.example.happybirthday", appCor
    }
}
```

What's New in 4.0

This panel describes some of the new features and changes included in this update.

To open this panel again later, select **Help > What' Android Studio** from the main menu.

Read in a browser

Build Speed window

When using Android Studio 4.0 Canary 3 with Andı plugin 4.0.0-alpha03 and higher, the **Build Speed** available to help you understand and diagnose iss build process, such as disabled optimizations and configured tasks. When using Android Studio 4.0 C higher, you can open the **Build Speed** window as f

1. If you haven't done so, build your ap **Build > Make** from the menu bar.
2. Select **View > Tool Windows > Build** from t
3. In the **Build** window, open the **Build Speed** v of the following ways:
   - After Android Studio finishes building y click the **Build Speed** tab.
   - After Android Studio finishes building y click the link in the right side of the **Bu** window.

Tasks Determining Build Duration (93.282 s) / Critical Path Learn Mor

```
28.558 s  27.6%  :santa-tracker:generateLi
15.184 s  14.7%  :santa-tracker:mergeRelea
15.117 s  14.6%  :santa-tracker:minifyRelea
 8.912 s   6.7%  :dasherdancer:compileRel
 4.671 s   4.5%  :santa-tracker:javaPreCon
 4.570 s   4.4%  :santa-tracker:processRel
```

Gradle sync finished in 2 s 290 ms (from cached state) (2 minutes ago)

In the bottom right corner of Android Studio a progress bar or message indicates whether Android Studio is still working on setting up your project. For example:

2 processes running...

Wait until Android Studio has completed setting up your project. A message in the bottom left corner, like the one shown below, will let you know when the project is complete.

Gradle sync finished in 44 s 546 ms

Run your app on a virtual device (emulator)

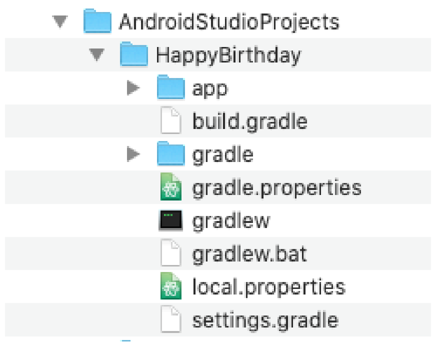
In this task, you'll use the Android Virtual Device (AVD) manager to create a software version (an emulator) of a mobile device and run it on your computer. A virtual device, or emulator, simulates the configuration for a particular type of Android device, such as a phone. This could be any phone or tablet running your choice of Android system versions. You'll then use the virtual device to run the app you created with the **Empty Activity** template.

Note: The Android Emulator is an independent application used to set up a virtual device, and it has its own system requirements. Virtual devices can use up a lot of disk space. If you run into any issues, see Run apps on the Android Emulator.

To create an Android virtual device (an emulator) to run your app, choose **Tools > AVD Manager** and then use the AVD Manager to select a hardware device and system image.

Find your project files

When you configured your project, Android Studio created a folder on your computer for all your Android projects called **AndroidStudioProjects**. Inside the **AndroidStudioProjects** folder, Android Studio also creates a folder with the same name as your app (**HappyBirthday** in this case).



```
▼ AndroidStudioProjects
  ▼ HappyBirthday
    ▶ app
      build.gradle
    ▶ gradle
      gradle.properties
      gradlew
      gradlew.bat
      local.properties
      settings.gradle
```

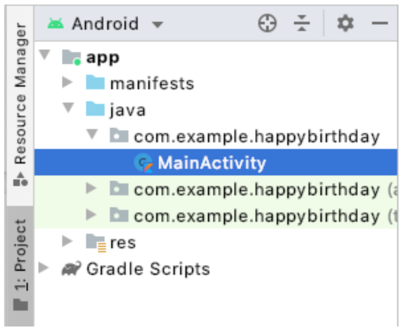
The **HappyBirthday** folder is your project folder. Android Studio saves both the files you create and the files that are created by Android Studio in your project folder.

Files in the **Project** window are organized to make navigation between the project files easier when you write code. However, if you look at the files in a file browser, such as Finder or Windows Explorer, the file hierarchy is organized very differently.

In this task you will explore these two different views of the project folder hierarchy.

2. In Android Studio, in the **Project** window, select **Android** from the drop-down menu in the top-left corner.

   You should see a file listing similar to the following:



```
Android ▾
▼ app
  ▶ manifests
  ▼ java
    ▼ com.example.happybirthday
        MainActivity
    ▶ com.example.happybirthday (a
    ▶ com.example.happybirthday (t
  ▶ res
▶ Gradle Scripts
```

This view and organization of your files is useful when working on writing code for your project.

You can also view your files as they would appear in a file browser, such as Finder (for macOS) or Explorer (for Windows).

3. In the **Project** window, select **Project Source Files** from the drop-down menu.

   Notice that the title changes to the folder where the project files are stored.

You can now browse the files in the same way as in any file explorer.

4. To switch back to the previous view, in the **Project** window, select **Android** again.

Run your app on a mobile device

To let Android Studio communicate with your Android device, you must enable USB debugging in the **Developer options** settings of the device.

To show developer options and enable USB debugging:

1. On your Android device, open **Settings**, and search for **About phone**.
2. Tap on **About phone**, and then tap **Build number** seven times. Enter your device password or pin, if prompted.
3. Return to **Settings**, and tap **System**.

   Developer options should now appear in the list. You may need to open the **Advanced** options to find it.

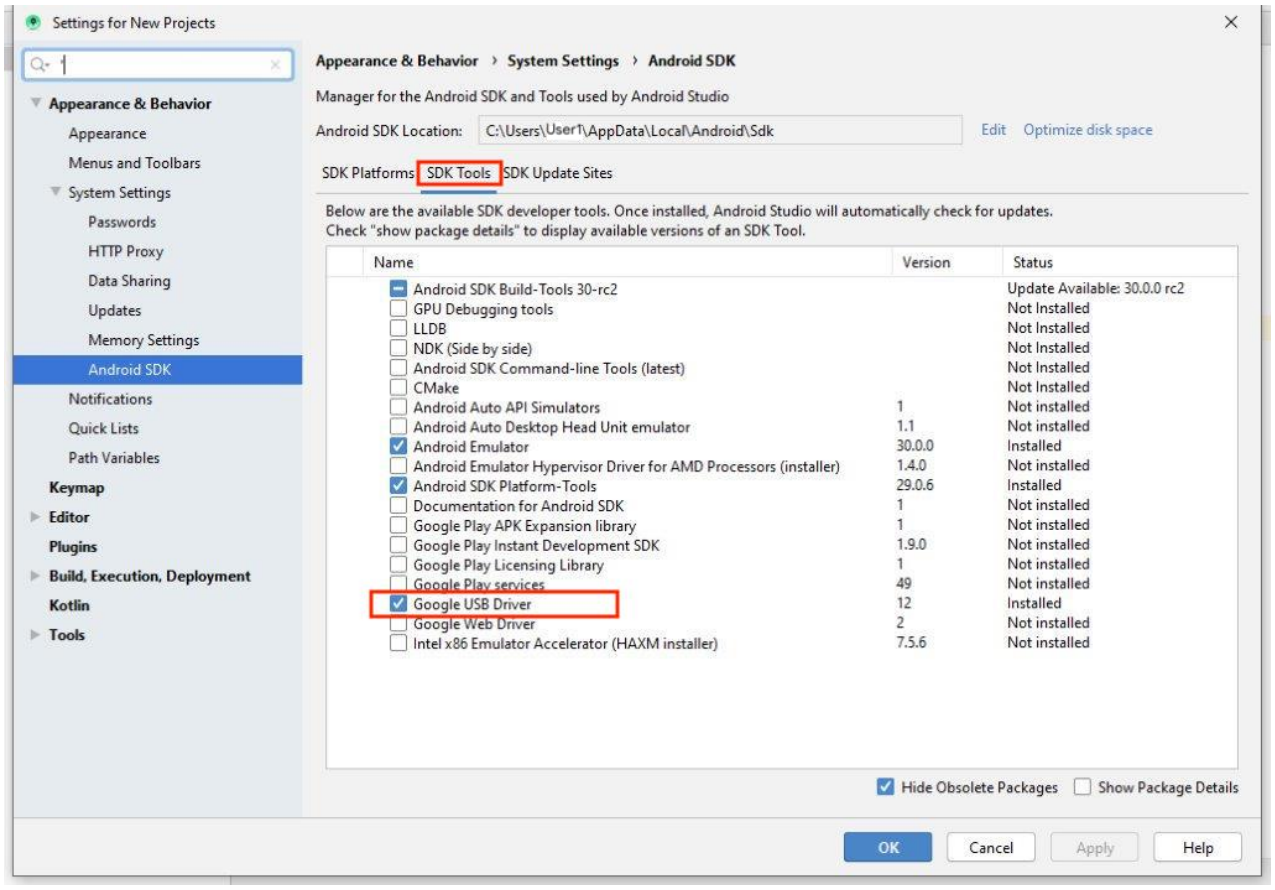
4. Tap **Developer options**, and then turn on **USB debugging**.

Install the Google USB Driver (Windows only)

If you've installed Android Studio on a Windows-based computer, you must install a USB device driver before you can run your app on a physical device.

1. In Android Studio, click **Tools > SDK Manager**.

   The **Settings for New Projects** window displays.



Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: C:\Users\User1\AppData\Local\Android\Sdk

Below are the available SDK developer tools. Once installed, Android Studio will automatically check for updates.
Check "show package details" to display available versions of an SDK Tool.

	Name	Version	Status
–	Android SDK Build-Tools 30-rc2		Update Available: 30.0.0 rc2
	GPU Debugging tools		Not Installed
	LLDB		Not Installed
	NDK (Side by side)		Not Installed
	Android SDK Command-line Tools (latest)		Not Installed
	CMake		Not Installed
	Android Auto API Simulators	1	Not installed
	Android Auto Desktop Head Unit emulator	1.1	Not installed
✓	Android Emulator	30.0.0	Installed
	Android Emulator Hypervisor Driver for AMD Processors (installer)	1.4.0	Not installed
✓	Android SDK Platform-Tools	29.0.6	Installed
	Documentation for Android SDK	1	Not installed
	Google Play APK Expansion library	1	Not installed
	Google Play Instant Development SDK	1.9.0	Not installed
	Google Play Licensing Library	1	Not installed
	Google Play services	49	Not installed
✓	Google USB Driver	12	Installed
	Google Web Driver	2	Not installed
	Intel x86 Emulator Accelerator (HAXM installer)	7.5.6	Not installed

✓ Hide Obsolete Packages ☐ Show Package Details

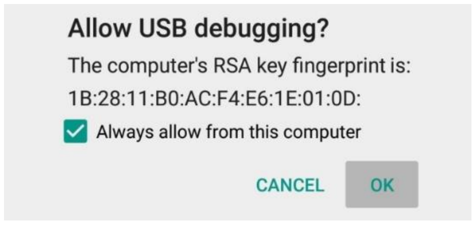
2. Click the **SDK Tools** tab.
3. Select **Google USB Driver** and click **OK**.

   When done, the driver files are downloaded into the **android_sdk\extras\google\usb_driver** directory. You should now be able to connect and run your app from Android Studio.

Run your app on the Android device (all operating systems)

Now you can connect your device and run the app from Android Studio.

1. Connect the Android device to your development machine with a USB cable. A dialog should appear on the device, asking to allow USB debugging.


Allow USB debugging?
The computer's RSA key fingerprint is:
1B:28:11:B0:AC:F4:E6:1E:01:0D:
✓ Always allow from this computer

2. Select the **Always allow** option to remember this computer. Tap **OK**. 3. On your computer, in the Android Studio toolbar, click the **Run** button ▶.

   Android Studio installs the app on your device and runs it.

   Note: If your device is running an Android platform that isn't installed in Android Studio, you might see a message asking if you want to install the needed platform. Click **Install and Continue**, then click **Finish** when the process is complete.

Troubleshooting

If Android Studio does not recognize your device, try the following:

1. Unplug the USB cable and plug it back in.
2. Restart Android Studio.

If your computer still does not find the device or declares it "unauthorized", follow these steps:

1. Disconnect the USB cable.
2. On the device, open the **Developer options** in **Settings**.
3. Tap **Revoke USB debugging authorizations**.
4. Reconnect the device to your computer.
5. When prompted, grant authorizations.

UNIT 1 : MODULE 3

Set up your Happy Birthday app

When you created this Happy Birthday app with the Empty Activity template, Android Studio set up resources for a basic Android application, including a "Hello World!" message in the middle of the screen. In this codelab, you will learn how that message gets there, how to change its text to be more of a birthday greeting, and how to add and format additional messages.

About user interfaces

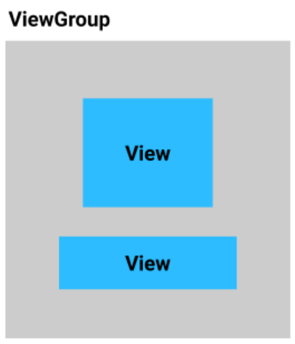
The user interface (UI) of an app is what you see on the screen: text, images, buttons, and many other types of elements. It's how the app shows things to the user, and how the user interacts with the app.

Each of these elements is what's called a `View`. Almost everything you see on the screen of your app is a `View`. `Views` can be interactive, like a clickable button or an editable input field.

In this codelab, you will work with a kind of `View` that is for displaying text and is called a `TextView`.

The `Views` in an Android app don't just float on the screen by themselves. `Views` have relationships to each other. For example, an image may be next to some text, and buttons may form a row. To organize `Views`, you put them in a container. A `ViewGroup` is a container that `View` objects can sit in, and is responsible for arranging the `Views` inside it. The arrangement, or *layout*, can change depending on the size and aspect ratio of the screen of the Android device that the app is running on, and the layout can adapt to whether the device is in portrait or landscape mode.

One kind of `ViewGroup` is a `ConstraintLayout`, which helps you arrange the `Views` inside it in a flexible way.

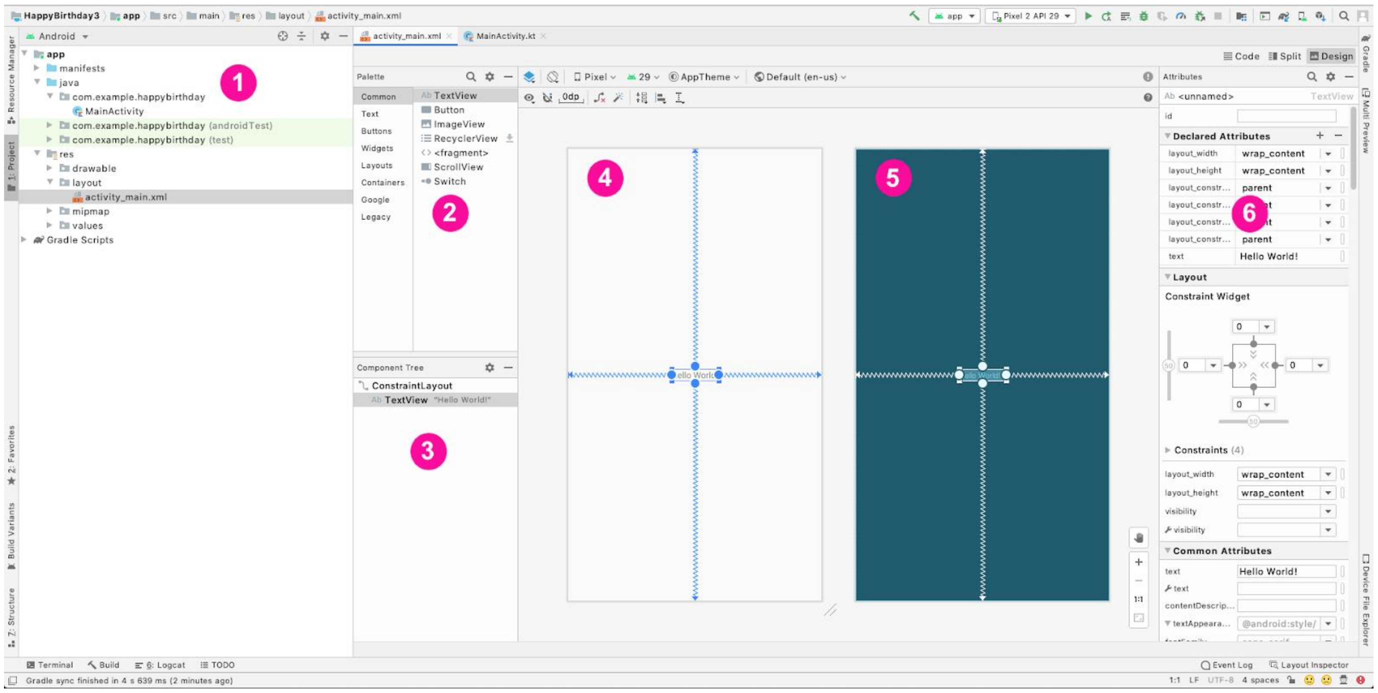


About the Layout Editor

Creating the user interface by arranging `Views` and `ViewGroups` is a big part of creating an Android App. Android Studio provides a tool that helps you do this, called the **Layout Editor**. You'll use the **Layout Editor** to change the "Hello World!" text to "Happy Birthday!", and later, to style the text.

When the **Layout Editor** opens, it has a lot of parts. You will learn to use most of them in this codelab. Use the annotated screenshot below for help recognizing the windows in the **Layout Editor**. You will learn more about each part as you make changes to your app.

Annotated screenshot of the whole **Layout Editor**:



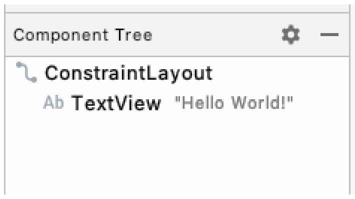
- On the left (1) is the **Project** window, which you have seen before. It lists all the files that make up your project.
- At the center you can see two drawings (4) and (5) that represent the screen layout for your app. The representation on the left (4) is a close approximation of what your screen will look like when the app runs. It's called the **Design** view.
- The representation on the right (5) is the **Blueprint** view, which can be useful for specific operations.
- The **Palette** (2) contains lists of different types of `Views` which you can add to your app.
- The **Component Tree** (3) is a different representation of the views of your screen. It lists all the views of your screen.
- On the far right (6) are **Attributes**. It shows you the attributes of a `View` and lets you change them.

  Read more about the **Layout Editor** and how to configure it in the developer reference guide.

Change the Hello World message

1. In Android Studio find the **Project** window on the left.
2. Take note of these folders and files: The **app** folder has most of the files for the app that you will change. The **res** folder is for resources, such as images or screen layouts. The **layout** folder is for screen layouts. The `activity_main.xml` file contains a description of your screen layout.
3. Expand the **app** folder, then the **res** folder, and then the **layout** folder.
4. Double-click on `activity_main.xml`. This opens `activity_main.xml` in the **Layout Editor** and shows the layout it describes in the **Design** view.

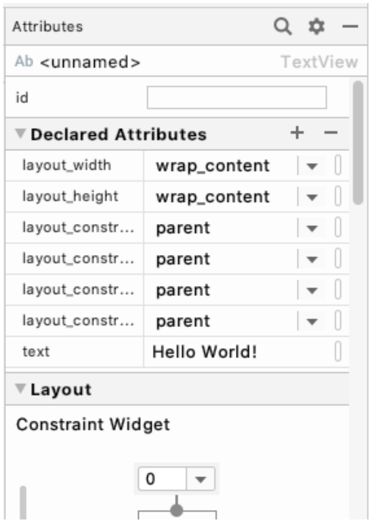
   Note: In these codelabs you will frequently be asked to open a file like in the previous steps. As a shorthand, this may be abbreviated as: Open **activity_main.xml** (**res** > **layout** > **activity_main.xml**) instead of listing each step separately.
5. Look at the list of views in the **Component Tree**. Notice that there is a `ConstraintLayout`, and a `TextView` underneath it. These represent the UI of your app. The `TextView` is indented because it is inside the `ConstraintLayout`. As you add more `Views` to the `ConstraintLayout`, they will be added to this list.
6. Notice that the `TextView` has **"Hello World!"** next to it, which is the text you see when you run the app.


Component Tree

ConstraintLayout

Ab TextView "Hello World!"

7. In the **Component Tree**, click on the `TextView`.
8. Find **Attributes** on the right.
9. Find the **Declared Attributes** section.
10. Notice that the **text** attribute in the **Declared Attributes** section contains **Hello World!**


Attributes

Ab <unnamed> TextView

id	

Declared Attributes

layout_width	wrap_content
layout_height	wrap_content
layout_constr...	parent
layout_constr...	parent
layout_constr...	parent
layout_constr...	parent
text	Hello World!

Layout

Constraint Widget

0

The **text** attribute shows the text that is printed inside a `TextView`.

11. Click on the **text** attribute where the **Hello World!** text is.
12. Change it to say **Happy Birthday!**, then press the **Enter** key. If you see a warning about a hardcoded string, don't worry about it for now. You will learn how to get rid of that warning in the next codelab.
13. Notice that the text has changed in the **Design View**.

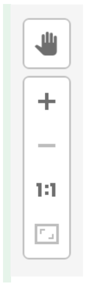
Add TextViews to the layout

The birthday card you are building looks different than what you have in your app now. Instead of the small text in the center, you need two larger messages, one in the upper left and one in the lower right corner. In this task you'll delete the existing `TextView`, and add two new `TextViews`, and learn how to position them within the `ConstraintLayout`.

Delete the current TextView

1. In the **Layout Editor**, click to select the `TextView` at the center of the layout.
2. Press the **Delete** key. Android Studio deletes the `TextView`, and your app now shows just a `ConstraintLayout` in the **Layout Editor** and the **Component Tree**.

Tip: If you want to zoom in on your layout, you can use the controls at the bottom right of the **Layout Editor** to adjust the size. Click on the bottom-most icon to return to a zoom level where the whole layout fits on your screen.

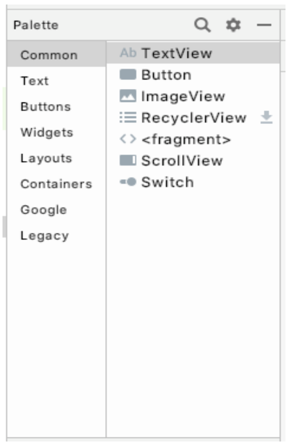


Add a TextView

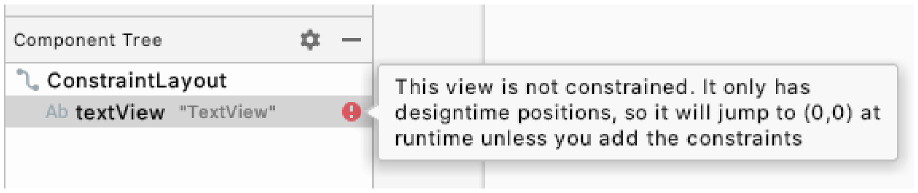
In this step, you'll add a `TextView` in the upper left of your app to hold the birthday greeting.

The **Palette** in the upper left of the **Layout Editor** contains lists of different types of `Views`, organized by category, which you can add to your app.

1. Find `TextView`. It appears both in the **Common** category and the **Text** category



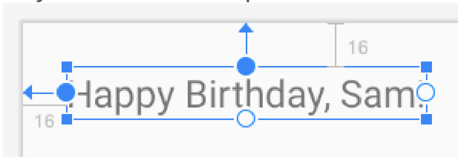
2. Drag a `TextView` from the **Palette** to the upper left of the design surface in the **Layout Editor** and drop it. You don't need to be exact, just drop it near the upper left corner.
3. Notice that there's a `TextView` added, and notice a red exclamation mark in the **Component Tree**.
4. Hover your pointer over the exclamation mark, and you'll see a warning message that the view is not constrained and will jump to a different position when you run the app. You'll fix that in the next step.



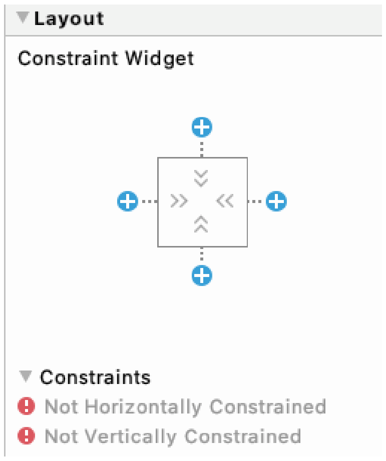
This view is not constrained. It only has designtime positions, so it will jump to (0,0) at runtime unless you add the constraints

Position the TextView

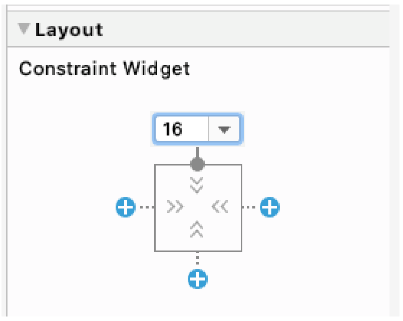
For the birthday card, the `TextView` needs to be near the upper left corner with some space around it. To fix the warning, you'll add some constraints to the `TextView`, which tell your app how to position it. Constraints are directions and limitations for where a `View` can be in a layout.

The constraints you add to the top and left will have margins. A margin specifies how far a `View` is from an edge of its container. 

1. In **Attributes** on the right, find the **Constraint Widget** in the **Layout** section. The square represents your view.

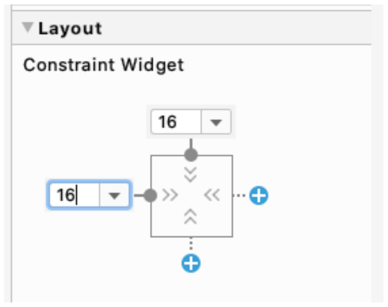


2. Click on the **+** at the top of the square. This is for the constraint between the top of your text view and the top edge of the constraint layout.
3. A field with a number appears for setting the top margin. The margin is the distance from the `TextView` to the edge of the container, the `ConstraintLayout`. The number you see will be different depending on where you dropped the `TextView`. When you set the top margin, Android Studio automatically also adds a constraint from the top of the text view to the top of the `ConstrainLayout`.

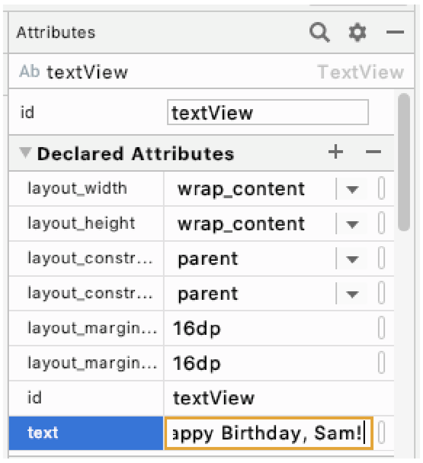


4. Change the top margin to 16.

Note: The unit for margins and other distances in the UI is *density-independent pixels* (dp). It's like centimeters or inches, but for distances on a screen. Android translates this value to the appropriate number of real pixels for each device. As a baseline, 1dp is about 1/160th of an inch, but may be bigger or smaller for some devices.

5. Do the same for the left margin. 

6. Set the **text** to wish your friend a happy birthday, for example "Happy Birthday, Sam!" and press **Enter**.



Attributes

Ab textView TextView

id	textView

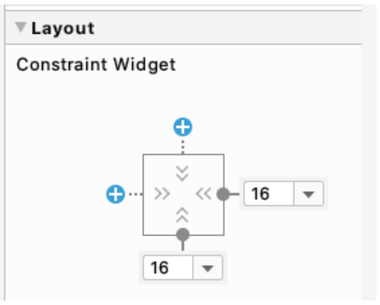
Declared Attributes

layout_width	wrap_content
layout_height	wrap_content
layout_constr...	parent
layout_constr...	parent
layout_margin...	16dp
layout_margin...	16dp
id	textView
text	appy Birthday, Sam!

7. Notice that the **Design** view updates to show what your app will look like

Add and position another TextView

Your birthday card needs a second line of text near the bottom right corner, which you'll add in this step in the same way as in the previous task. What do you think the margins of this `TextView` should be?

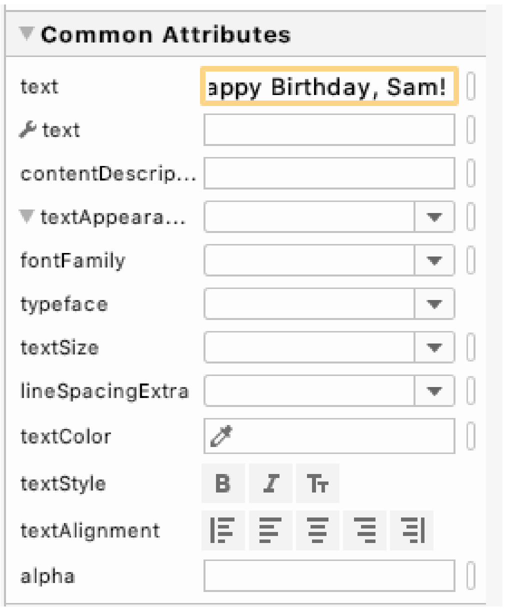
1. Drag a new `TextView` from the **Palette** and drop it near the bottom right of the app view in the Layout Editor.
2. Set a right margin of 16.
3. Set a bottom margin of 16. 

4. In **Attributes**, set the **text** attribute to sign your card, for example "From Emma."
5. Run your app. You should see your birthday wish in the upper left and your signature in the lower right.

Add style to the text

You added text to your user interface, but it doesn't look like the final app yet. In this task, you'll learn how to change the size, text color, and other attributes that affect the appearance of the `TextView`. You can also experiment with different fonts.

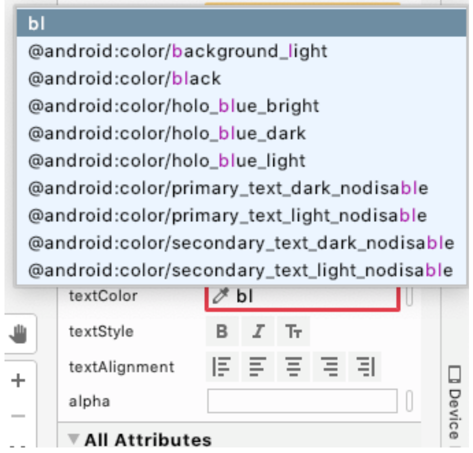
1. Click on the first `TextView` in the **Component Tree** and find the **Common Attributes** section of the **Attributes** window. You may need to scroll down to find it.



2. Notice the various attributes including **fontFamily**, **textSize**, and **textColor**.
3. Look for **textAppearance**.
4. If **textAppearance** is not expanded, click on the down triangle.
5. Set the **textSize** to **36sp**.

   Note: Just like dp is a unit of measure for distances on the screen, **sp** is a unit of measure for the font size. UI elements in Android apps use two different units of measurement, *density-independent pixels* (**dp**) which you used earlier for the layout, and *scalable pixels* (**sp**) which is used when setting the size of text. By default, sp is the same size as dp, but it resizes based on the user's preferred text size.

6. Notice the change in the **Layout Editor**.
7. Change the **fontFamily** to **casual**.
8. Try some different fonts to see what they look like. There are even more choices for fonts at the bottom of the list, under **More Fonts...**
9. When you're done trying different fonts, set the **fontFamily** to **sans-serif-light**.
10. Click on the **textColor** attribute's edit box and start typing **black**. Notice that as you type, Android Studio shows a


bl
@android:color/background_light
@android:color/black
@android:color/holo_blue_bright
@android:color/holo_blue_dark
@android:color/holo_blue_light
@android:color/primary_text_dark_nodisable
@android:color/primary_text_light_nodisable
@android:color/secondary_text_dark_nodisable
@android:color/secondary_text_light_nodisable
textColor
bl
textStyle
textAlignment
alpha
All Attributes
Device

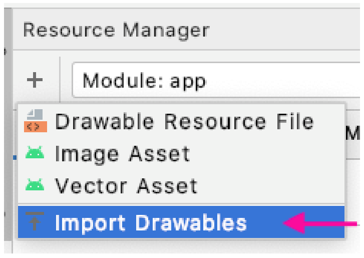
   list of colors that contain the text you've typed so far.

11. Select **@android:color/black** from the list of colors and press **Enter**.
12. In the `TextView` with your signature, change the **textSize**, **textColor** and **fontFamily** to match.
13. Run your app and look at the result.

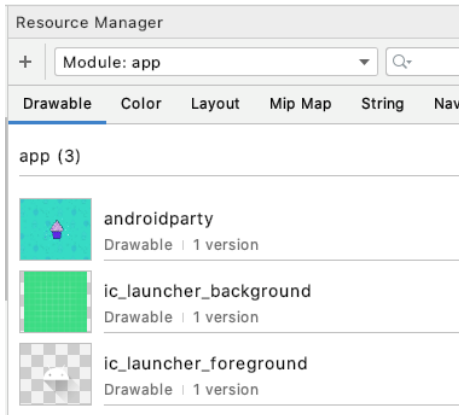
Add images to your Android app

In this task, you'll download an image from the internet and add it to your Happy Birthday app.

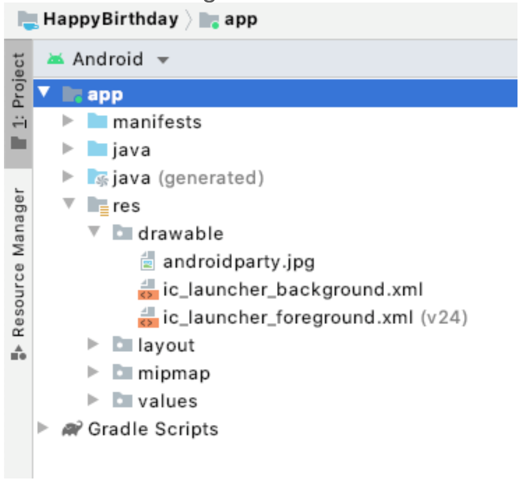
1. Click here to access the image for your birthday card on Github.
2. In Android Studio, click on **View > Tool Windows > Resource Manager** in the menus or click on the **Resource Manager** tab to the left of the **Project** window.
3. Click the **+** below **Resource Manager**, and select **Import Drawables**. This opens a file browser.



4. In the file browser find the image file you downloaded and click **Open**.
5. Click **Next**. Android Studio shows you a preview of the image.
6. Click **Import**.
7. If the image has been imported successfully, Android Studio adds the image to your **Drawable** list. This list includes all your images and icons for the app. You can now use this image in your app.



8. Switch back to the project view by clicking on **View > Tool Windows > Project** in the menus or the **Project** tab on the far left.
    1. Confirm the image is in the **drawable** folder of your app by expanding **app > res >drawable**.



Add an ImageView

In order to display an image in your app, it needs a place to be displayed. Just like you use a `TextView` to display text, you can use an `ImageView` to display images.

In this task, you'll add an `ImageView` to your app, and set its image to the cupcake image you downloaded. Then you'll position it and adjust its size so it fills the screen.

Add an ImageView and set its image

1. In the **Project** window open **activity_main.xml** (**app > res > layout > activity_main.xml**).

   Tip: If you don't see the **Layout Editor**, click on the **Design** mode button in the upper right.

2. In the **Layout Editor,** go to the **Palette** and drag an `ImageView` to your app. Drop it near the center and not overlapping any of the text

   The **Pick a Resource** dialog opens. This dialog lists all the image resources available to your app. Notice the birthday image listed under the **Drawable** tab. A *drawable resource* is a general concept for a graphic that can be drawn to the screen. It includes images, bitmaps, and icons and many other types of drawn resources.

3. In the **Pick a Resource** dialog find the cake image in the **Drawable** list.
4. Click on the image and then click **OK**.

   This adds the image to your app, but it's probably not in the right place and it doesn't fill the screen. You'll fix that in the next step.

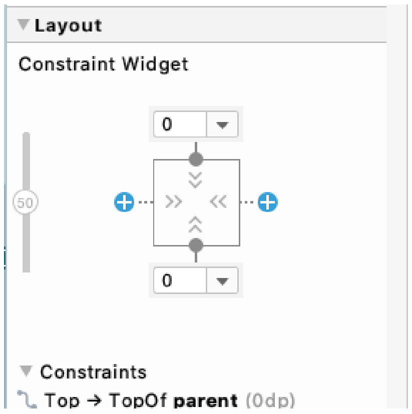
Position and size the ImageView

1. Click and drag the `ImageView` around in the **Layout Editor**, and note that as you drag, a pink rectangle appears around the app screen in the **Design** view. The pink rectangle indicates the boundaries of the screen for positioning your `ImageView`.
2. Drop the `ImageView` so that the left and right edges align with the pink rectangle. Android Studio will "snap" the image to the edges when you're close. (You'll take care of the top and bottom in a moment.)

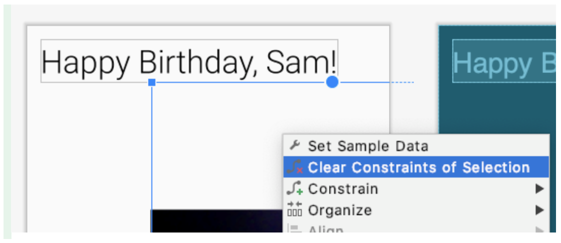
   `Views` in a `ConstraintLayout` need to have horizontal and vertical constraints to tell the `ConstraintLayout` how to position them. You'll add those next.

   Note: A constraint in the context of the **Layout Editor** represents a connection or alignment of a view to another view, the parent layout, or an invisible guideline. For example, a view can be constrained to be a certain distance from the edge of its container, or always be to the right of another view, or always the top view inside a container.

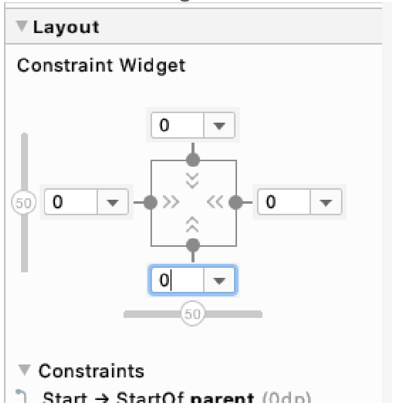
1. Hold the pointer over the circle at the top of the outline of the `ImageView`, and it highlights with another circle.
2. Drag the circle towards the top of the app screen, and an arrow connects the circle to your pointer as you are dragging. Drag until it snaps to the top of the screen. You've added a constraint from the top of the `ImageView` to the top of the `ConstraintLayout`.
3. Add a constraint from the bottom of the `ImageView` to the bottom of the `ConstraintLayout`. It may be too close to the edge to drag as you did for the top. In that case, you can click on the bottom **+** in the **Constraint Widget** in the **Attributes** window to add a constraint. Make sure the margin is 0.



Tip: Adding constraints can be more difficult as you add more `Views` on the screen. If you add a constraint to the wrong `View`, you can undo that step with `Control+Z` (`Command+Z` on a Mac.) Or right-click on the `View` which will pop up a menu, and choose **Clear Constraints of Selection** which will remove all the constraints on that `View`.

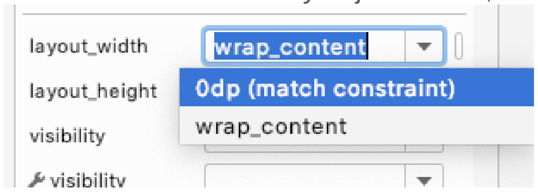


4. In the **Attributes** pane, use the **Constraint Widget** to add a margin of 0 to the left and to the right sides. This automatically creates a constraint in that direction.

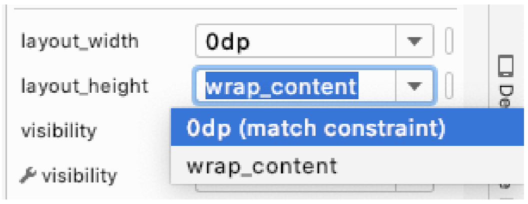


The image is now centered, but it isn't taking up the whole screen yet. You'll fix that in the next steps.

1. Below the **Constraint Widget** in the **Constraints** section, set the **layout_width** to **0dp (match constraint)**. 0dp is a shorthand to tell Android Studio to use *match constraint* for the width of the `ImageView`. "match constraints means.... Because of constraints you just added, this makes it as wide as the `ConstraintLayout`, minus any margins.

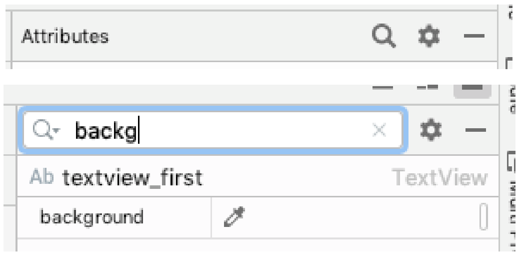


2. Set the **layout_height** to **0dp (match constraint)**. Because of the constraints you added, this makes the `ImageView` as tall as the `ConstraintLayout`, minus any margins.

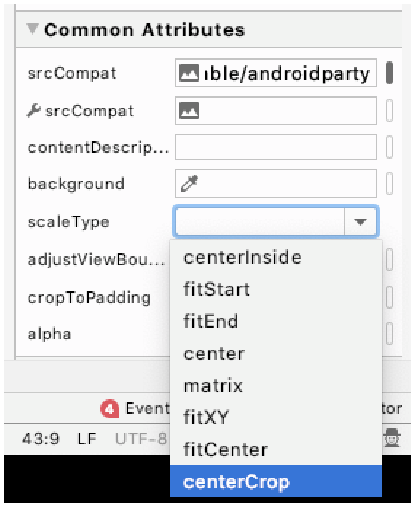


3. The `ImageView` is as wide and tall as the app screen, but the image is centered inside the `ImageView` and there is a bunch of whitespace above and below the image. Since that doesn't look very attractive, you will adjust the **scaleType** of the `ImageView`, which says how to size and align the image. Read more about `ScaleType` in the developer reference guide. Your app should now look as shown below.
4. Find the **scaleType** attribute. You may need to scroll down or search for this attribute. Try setting different values for the **scaleType** to see what they do.

   Tip: To find a property in the list of all the properties, click on the magnifying glass icon to the right of Attributes, and begin typing the name of the property. Android Studio will show just the properties that contain that string.



5. When you're done, set the **scaleType** to **centerCrop** because that makes the image fill the screen without distorting it.



The cake image should now fill the entire screen, as shown below.

But now you can't see your birthday greeting and signature. You'll fix that in the next step.

Move the ImageView behind the text

After making the `ImageView` fill the screen, you can't see the text anymore. This is because the image now covers up the text. It turns out that the order of your UI elements matters. You added the `TextViews` first, then you added the `ImageView`, which means it went on top. When you add views to a layout, they are added at the end of a list of views, and they get drawn in the order they are in the list. The `ImageView` was added to the end of the list of `Views` in the `ConstraintLayout`, which means it's drawn last, and draws over the `TextViews`. To fix that, you'll change the order of the views and move the `ImageView` to the beginning of the list so it's drawn first.

In the **Component Tree**, click on the `ImageView`, and drag it above the `TextViews` to just below the `ConstraintLayout`. A blue line with a triangle appears showing where the `ImageView` will go. Drop the `ImageView` just below the `ConstraintLayout.`

The `ImageView` should now be first in the list below the `ConstraintLayout`, with the `TextViews` after it. Both the "Happy Birthday, Sam!" and the "From Emma." text should now be visible. (Ignore the warning for the missing content description for now.)

Congratulations! You've added an image to your Android app!

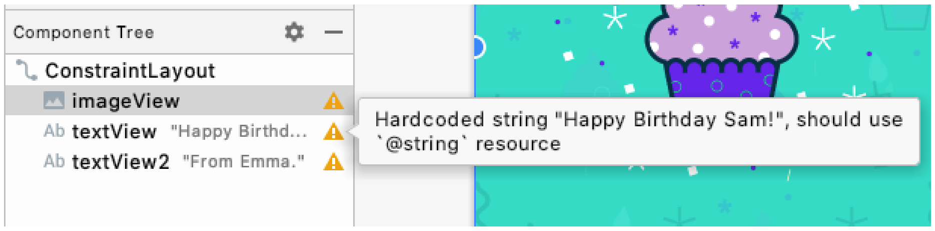
Adopt good coding practices

When you added the `TextViews` in the previous codelab, Android Studio flagged them with warning triangles. In this step, you'll fix those warnings and also fix the warning on the `ImageView`.

Translating

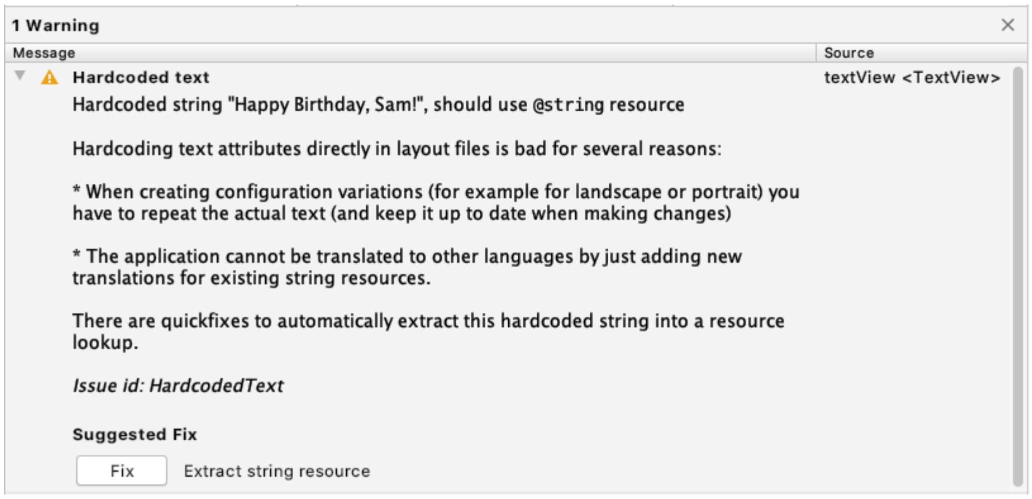
When you write apps, it's important to remember that they may be translated into another language at some point. As you learned in an earlier codelab, a string is a sequence of characters like "Happy Birthday, Sam!".

A *hardcoded* string is one that is written directly in the code of your app. Hardcoded strings make it more difficult to translate your app into other languages, and harder to reuse a string in different places in your app. You can deal with those issues by "extracting strings into a resource file". That means instead of hardcoding the string in your code, you put the string into a file, give it a name, and then use the name whenever you want to use this string. The name will stay the same, even if you change the string or translate it to a different language.



Hardcoded string "Happy Birthday Sam!", should use `@string` resource

1. Click on the orange triangle next to the first `TextView` with "Happy Birthday, Sam!" Android Studio opens a window with more information and a suggested fix. You may need to scroll down to see the **Suggested Fix**.



Hardcoded text textView <TextView>

Hardcoded string "Happy Birthday, Sam!", should use @string resource

Hardcoding text attributes directly in layout files is bad for several reasons:

* When creating configuration variations (for example for landscape or portrait) you have to repeat the actual text (and keep it up to date when making changes)

* The application cannot be translated to other languages by just adding new translations for existing string resources.

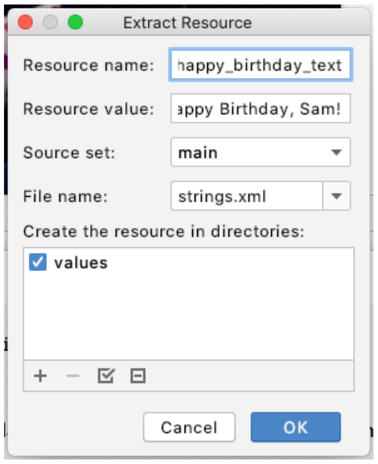
There are quickfixes to automatically extract this hardcoded string into a resource lookup.

Issue id: HardcodedText

Suggested Fix

Extract string resource

2. Click the **Fix** button. Android Studio opens the **Extract Resource** dialog. In this dialog, you can customize what your string resource is called, and some details on how to store it. The **Resource name** is what the string is going to be called. The **Resource value** will be the actual string itself.
3. In the **Extract Resource** dialog, change the **Resource name** to **happy_birthday_text**. String resources should have lowercase names, and multiple words should be separate with an underscore. Leave the other settings with the defaults.


Extract Resource

Resource name: happy_birthday_text

Resource value: appy Birthday, Sam!

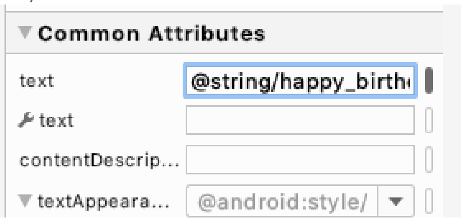
Source set: main

File name: strings.xml

Create the resource in directories:

☑ values

4. Click the **OK** button.
5. In the **Attributes** pane, find the **text** attribute, and notice that Android Studio has automatically set it to **@string/happy_birthday_text** for you.


Common Attributes

text: @string/happy_birth

text:

contentDescrip...:

textAppeara...: @android:style/

6. Open **strings.xml** (**app > res > values > strings.xml**) and notice that Android Studio has created a string resource called **happy_birthday_text**.

```
<resources>
    <string name="app_name">Happy Birthday</string>
    <string name="happy_birthday_text">Happy Birthday, Sam!</string>
</resources>
```

The `strings.xml` file has a list of strings that the user will see in your app. Note that the name of your app is also a string resource. By putting the strings all in one place, you can more easily translate all the text in your app, and more easily reuse a string in different parts of your app.

7. Follow the same steps to extract the text for the signature `TextView`, and name the string resource **signature_text**.

   Your finished file should look like this.

```
<resources>
    <string name="app_name">Happy Birthday</string>
    <string name="happy_birthday_text">Happy Birthday, Sam!</string>
    <string name="signature_text">From Emma.</string>
</resources>
```

Check your app for accessibility

Following good coding practices for accessibility allows all of your users, including users with disabilities, to navigate and interact with your app more easily.

Note: Android provides many tools for users. For example, TalkBack is the Google screen reader included on Android devices. TalkBack gives users spoken feedback so that users can use their device without looking at the screen. Read more about accessibility in the developer reference guide.

Android Studio provides hints and warnings to help you make your app more accessible.

1. In the **Component Tree**, notice the orange triangle next to the `ImageView` that you added earlier. If you hover the pointer over it, you'll see a tooltip with a warning about a missing 'contentDescription' attribute on the image. A content description can help make your app more usable with TalkBack by defining the purpose of the UI element.

   However, the image in this app is only included for decorative purposes. Instead of setting the content description that is announced to the user, you can just tell TalkBack to skip the `ImageView` by setting its **importantForAccessibility** attribute to **no**.

2. In the **Component Tree**, select the `ImageView`.
3. In the **Attributes** window, in the **All Attributes** section, find **importantForAccessibility** and set it to **no**.

   The orange triangle next to the `ImageView` disappears.

4. Run your app again to make sure it still works.

   Congratulations! You've added an image to your Happy Birthday app, followed the accessibility guidelines, and made it.

Summary

- The **Resource Manager** in Android Studio helps you add and organize your images and other resources.
- An `ImageView` is a UI element for displaying images in your app.
- `ImageViews` should have a content description to help make your app more accessible.
- Text that is shown to the user like the birthday greeting should be extracted into a string resource to make it easier to translate your app into other languages.

UNIT 1 : MODULE 4

2. Roll random numbers

Games often have a random element to them. You could earn a random prize or advance a random number of steps on the game board. In your everyday life, you can use random numbers and letters to generate safer passwords!

Instead of rolling actual dice, you can write a program that simulates rolling dice for you. Each time you roll the dice, the outcome can be any number within the range of possible values. Fortunately, you don't have to build your own random-number generator for such a program. Most programming languages, including Kotlin, have a built-in way for you to generate random numbers. In this task, you will use the Kotlin code to generate a random number.

Set up your starter code

1. In your browser, open the website https://developer.android.com/training/kotlinplayground.
2. Delete all the existing code in the code editor and replace it with the code below. This is the `main()` function you worked with in earlier codelabs.

```
fun main() {

}
```

Use the random function

To roll a dice, you need a way to represent all the valid dice roll values. For a regular 6-sided dice, the acceptable dice rolls are: 1, 2, 3, 4, 5, and 6.

Previously, you learned that there are types of data like `Int` for integer numbers and `String` for text. `IntRange` is another data type, and it represents a range of integer numbers from a starting point to an endpoint. `IntRange` is a suitable data type for representing the possible values a dice roll can produce.

1. Inside your `main()` function, define a variable as a `val` called `diceRange`. Assign it to an `IntRange` from 1 to 6, representing the range of integer numbers that a 6-sided dice can roll.

```
val diceRange = 1..6
```

You can tell that `1..6` is a Kotlin range because it has a start number, two dots, followed by an ending number (no spaces in between). Other examples of integer ranges are `2..5` for the numbers 2 through 5, and `100..200` for the numbers 100 through 200.

Tip: Notice that it does not say `IntRange` in this definition, in the same way you did not have to specify `Int` or `String` when creating a variable for an integer or a string. Most of the time, the system can figure out what data type you intend.

For example, the system interprets this:

```
val diceRange = 1..6
```

as this:

```
val diceRange: IntRange = 1..6
```

Similar to how calling `println()` tells the system to print the given text, you can use a function called `random()` to generate and return a random number for you for a given range. As before, you can store the result in a variable.

2. Inside `main()`, define a variable as a `val` called `randomNumber`.
3. Make `randomNumber` have the value of the result of calling `random()` on the `diceRange` range, as shown below.

```
 val randomNumber = diceRange.random()
```

Notice that you are calling `random()` on `diceRange` using a period, or dot, between the variable and the function call. You can read this as "generating a random number from `diceRange`". The result is then stored in the `randomNumber` variable.

4. To see your randomly generated number, use the string formatting notation (also called a "string template") `${randomNumber}` to print it, as shown below.

```
println("Random number: ${randomNumber}")
```

Your finished code should look like this.

```
fun main() {
    val diceRange = 1..6
    val randomNumber = diceRange.random()
    println("Random number: ${randomNumber}")
}
```

5. Run your code several times. Each time, you should see output as below, with different random numbers.

```
Random number: 4
```

Tips about ranges:

- Ranges can be between any integers. The following are valid ranges: `3..46`, `0..270`, `-6..+6`, `-10..-4`.
- You can call functions directly on a range, for example: `(1..6).random()`.

3. Create a Dice class

When you roll dice, they are real objects in your hands. While the code you just wrote works perfectly fine, it's hard to imagine that it's about actual dice. Organizing a program to be more like the things it represents makes it easier to understand. So, it would be cool to have programmatic dice that you can roll!

All dice work essentially the same. They have the same properties, such as sides, and they have the same behavior, such as that they can be rolled. In Kotlin, you can create a programmatic blueprint of a dice that says that dice have sides and can roll a random number. This blueprint is called a *class*.

From that class, you can then create actual dice objects, called *object instances*. For example, you can create a 12-sided dice, or a 4-sided dice.

Tip: A *class* is similar to how an architect's blueprint plans are not the house; they are the instructions of how to build the house. The house is the actual thing or *object instance* created according to the blueprint.

Note: Organizing everything about dice into a class is called encapsulation. *Encapsulation* is a big fancy word, but all it means is that you can enclose functionality that is logically related into a single place.

Define a Dice class

In the following steps, you will define a new class called `Dice` to represent a rollable dice.

1. To start afresh, clear out the code in the `main()` function so that you end up with the code as shown below.

```
fun main() {

}
```

2. Below this `main()` function, add a blank line, and then add code to create the `Dice` class. As shown below, start with the keyword `class`, followed by the name of the class, followed by an opening and closing curly brace. Leave space in between the curly braces to put your code for the class.

```
class Dice {

}
```

Note:

- Similar to using the `fun` keyword in Kotlin to create a new function, use the `class` keyword to create a new class.
- You can choose any name for a `class`, but it is helpful if the name indicates what the class represents. By convention, the class name is in camel case. For example: `Car`, `ParkingMeter`, and `CustomerRecord` are all valid class names, and you can guess at what they represent.

Inside a class definition, you can specify one or more properties for the class using variables. Real dice can have a number of sides, a color, or a weight. In this task, you'll focus on the property of number of sides of the dice.

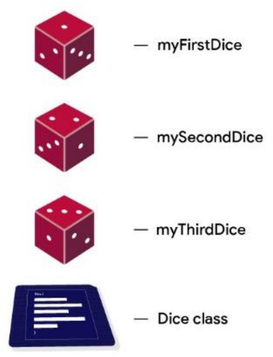
3. Inside the `Dice` class, add a `var` called `sides` for the number of sides your dice will have. Set `sides` to 6.

```
class Dice {
    var sides = 6
}
```

That's it. You now have a very simple class representing dice.

Create an instance of the Dice class

With this `Dice` class, you have a blueprint of what a dice is. To have an actual dice in your program, you need to create a `Dice` object instance. (And if you needed to have three dice, you would create three object instances.)



1. To create an object instance of `Dice`, in the `main()` function, create a `val` called `myFirstDice` and initialize it as an instance of the `Dice` class. Notice the parentheses after the class name, which denote that you are creating a new object instance from the class.

```
fun main() {
    val myFirstDice = Dice()
}
```

Now that you have a `myFirstDice` object, a thing made from the blueprint, you can access its properties. The only property of `Dice` is its `sides`. You access a property using the "dot notation". So, to access the `sides` property of `myFirstDice`, you call `myFirstDice.sides` which is pronounced "`myFirstDice` dot `sides`".

2. Below the declaration of `myFirstDice`, add a `println()` statement to output the number of `sides` of `myFirstDice`.

```
println(myFirstDice.sides)
```

Note: You used the dot notation earlier when calling `diceRange.random()`. Generalized, you can think of the dot notation as saying, "on something-dot-perform some action". Such as here, with `myFirstDice.sides`, "on this instance, get the `sides` property".

Your code should look like this.

```
fun main() {
    val myFirstDice = Dice()
    println(myFirstDice.sides)
}

class Dice {
    var sides = 6
}
```

3. Run your program and it should output the number of `sides` defined in the `Dice` class.

```
6
```

You now have a `Dice` class and an actual dice `myFirstDice` with 6 `sides`.

Let's make the dice roll!

Make the Dice Roll

You previously used a function to perform the action of printing cake layers. Rolling dice is also an action that can be implemented as a function. And since all dice can be rolled, you can add a function for it inside the `Dice` class. A function that is defined inside a class is also called a *method*.

1. In the `Dice` class, below the `sides` variable, insert a blank line and then create a new function for rolling the dice. Start with the Kotlin keyword `fun`, followed by the name of the method, followed by parentheses `()`, followed by opening and closing curly braces `{}`. You can leave a blank line in between the curly braces to make room for more code, as shown below. Your class should look like this.

```
class Dice {
    var sides = 6

    fun roll() {

    }
}
```

Note: You can name this method anything you want, but it is helpful to give it a name that indicates what action it performs. The naming convention for functions and methods is to start with a lowercase letter, use camel case, and start with an action verb, if possible.

When you roll a six-sided dice, it produces a random number between 1 and 6.

2. Inside the `roll()` method, create a `val randomNumber`. Assign it a random number in the `1..6` range. Use the dot notation to call `random()` on the range.

```
val randomNumber = (1..6).random()
```

3. After generating the random number, print it to the console. Your finished `roll()` method should look like the code below.

   ```
   fun roll() {
       val randomNumber = (1..6).random()
       println(randomNumber)
   }
   ```

4. To actually roll `myFirstDice`, in `main()`, call the `roll()` method on `myFirstDice`. You call a method using the "dot notation". So, to call the `roll()` method of `myFirstDice`, you type `myFirstDice.roll()` which is pronounced "`myFirstDice` dot `roll()`".

   ```
   myFirstDice.roll()
   ```

   Your completed code should look like this.

   ```
   fun main() {
      val myFirstDice = Dice()
      println(myFirstDice.sides)
      myFirstDice.roll()
   }

   class Dice {
      var sides = 6

      fun roll() {
         val randomNumber = (1..6).random()
         println(randomNumber)
      }
   }
   ```

5. Run your code! You should see the result of a random dice roll below the number of sides. Run your code several times, and notice that the number of sides stays the same, and the dice roll value changes.

   ```
   6

   4
   ```

   Congratulations! You have defined a `Dice` class with a `sides` variable and a `roll()` function. In the `main()` function, you created a new `Dice` object instance and then you called the `roll()` method on it to produce a random number.

4. Return your dice roll's value

Currently you are printing out the value of the `randomNumber` in your `roll()` function and that works great! But sometimes it's more useful to return the result of a function to whatever called the function. For example, you could assign the result of the `roll()` method to a variable, and then move a player by that amount! Let's see how that's done.

1. In main() modify the line that says myFirstDice.roll(). Create a val called diceRoll. Set it equal to the value returned by the roll() method.

   ```
   val diceRoll = myFirstDice.roll()
   ```

   This doesn't do anything yet, because roll() doesn't return anything yet. In order for this code to work as intended, roll() has to return something.

   In previous codelabs you learned that you need to specify a data type for input arguments to functions. In the same way, you have to specify a data type for data that a function returns.

2. Change the roll() function to specify what type of data will be returned. In this case, the random number is an Int, so the return type is Int. The syntax for specifying the return type is: After the name of the function, after the parentheses, add a colon, space, and then the Int keyword for the return type of the function. The function definition should look like the code below.

   ```
   fun roll(): Int {
   ```

3. Run this code. You will see an error in the **Problems View**. It says:

   ```
   A ‘return'  expression is required in a function with a block body.
   ```

   You changed the function definition to return an Int, but the system is complaining that your

   code doesn't actually return an Int. "Block body" or "function body" refers to the code between the curly braces of a function. You can fix this error by returning a value from a function using a return statement at the end of the function body.

4. In roll(), remove the println() statement and replace it with a return statement for randomNumber. Your roll() function should look like the code below.

   ```
   fun roll(): Int {
       val randomNumber = (1..6).random()
       return randomNumber
   }
   ```

5. In main() remove the print statement for the sides of the dice.
6. Add a statement to print out the value of sides and diceRoll in an informative sentence. Your finished main() function should look similar to the code below.

   ```
   fun main() {
      val myFirstDice = Dice()
      val diceRoll = myFirstDice.roll()
      println("Your ${myFirstDice.sides} sided dice rolled ${diceRoll}!")
   }
   ```

7. Run your code and your output should be like this.

   ```
   Your 6 sided dice rolled 4!
   ```

   Here is all your code so far.

```
fun main() {
    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.sides} sided dice rolled ${diceRoll}!")
}


class Dice {
    var sides = 6

    fun roll(): Int {
        val randomNumber = (1..6).random()
        return randomNumber
    }
}
```

5. Change the number of sides on your Dice

Not all dice have 6 sides! Dice come in all shapes and sizes: 4 sides, 8 sides, up to 120 sides!

1. In your `Dice` class, in your `roll()` method, change the hard-coded `1..6` to use `sides` instead, so that the range, and thus the random number rolled, will always be right for the number of sides.

```
val randomNumber = (1..sides).random()
```

2. In the `main()` function, below and after printing the dice roll, change `sides` of my`FirstDice` to be set to 20.

```
myFirstDice.sides = 20
```

3. Copy and paste the existing print statement below after where you changed the number of sides.
4. Replace the printing of `diceRoll` with printing the result of calling the `roll()` method on `myFirstDice`.

```
println("Your ${myFirstDice.sides} sided dice has rolled a ${myFirstDice.roll()}!")
```

Your program should look like this.

```
fun main() {

    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.sides} sided dice rolled ${diceRoll}!")

    myFirstDice.sides = 20
    println("Your ${myFirstDice.sides} sided dice rolled ${myFirstDice.roll()}!")
}

class Dice {
    var sides = 6

    fun roll(): Int {
        val randomNumber = (1..sides).random()
        return randomNumber
    }
}
```

5. Run your program and you should see a message for the 6-sided dice, and a second message for the 20-sided dice.

```
Your 6 sided dice rolled 3!

Your 20 sided dice rolled 15!
```

6. Customize your dice

The idea of a class is to represent a thing, often something physical in the real world. In this case, a `Dice` class does represent a physical dice. In the real world, dice cannot change their number of sides. If you want a different number of sides, you need to get a different dice. Programmatically, this means that instead of changing the sides property of an existing `Dice` object instance, you should create a new dice object instance with the number of sides you need.

In this task, you are going to modify the `Dice` class so that you can specify the number of sides when you create a new instance. Change the `Dice` class definition so you can supply the number of sides. This is similar to how a function can accept arguments for input.

1. Modify the `Dice` class definition to accept an integer called `numSides`. The code inside your class does not change.

```
class Dice(val numSides: Int) {
  // Code inside does not change.
}
```

2. Inside the `Dice` class, delete the `sides` variable, as you can now use `numSides`.
3. Also, fix the range to use `numSides`.

Your `Dice` class should look like this.

```
class Dice (val numSides: Int) {

    fun roll(): Int {
        val randomNumber = (1..numSides).random()
        return randomNumber
    }
}
```

If you run this code, you will see a lot of errors, because you need to update `main()` to work with the changes to the `Dice` class.

4. In `main()`, to create `myFirstDice` with 6 sides, you must now supply in the number of sides as an argument to the `Dice` class, as shown below.

   ```
   val myFirstDice = Dice(6)
   ```

5. In the print statement, change `sides` to `numSides`.
6. Below that, delete the code that changes `sides` to 20, because that variable does not exist anymore.
7. Delete the `println` statement underneath it as well.

   Your `main()` function should look like the code below, and if you run it, there should be no errors.

   ```
   fun main() {
       val myFirstDice = Dice(6)
       val diceRoll = myFirstDice.roll()
       println("Your ${myFirstDice.numSides} sided dice rolled ${diceRoll}!")
   }
   ```

8. After printing the first dice roll, add code to create and print a second `Dice` object called `mySecondDice` with 20 sides.

   ```
   val mySecondDice = Dice(20)
   ```

9. Add a print statement that rolls and prints the returned value.

   ```
   println("Your ${mySecondDice.numSides} sided dice rolled  ${mySecondDice.roll()}!")
   ```

10. Your finished `main()` function should look like this.

    ```
    fun main() {
        val myFirstDice = Dice(6)
        val diceRoll = myFirstDice.roll()
        println("Your ${myFirstDice.numSides} sided dice rolled ${diceRoll}!")

        val mySecondDice = Dice(20)
        println("Your ${mySecondDice.numSides} sided dice rolled ${mySecondDice.roll()}!")
    }

    class Dice (val numSides: Int) {

        fun roll(): Int {
            val randomNumber = (1..numSides).random()
            return randomNumber
        }
    }
    ```

11. Run your finished program, and your output should look like this.

    ```
    Your 6 sided dice rolled 5!

    Your 20 sided dice rolled 7!
    ```

7. Adopt Good Coding Practices

When writing code, concise is better. You can get rid of the `randomNumber` variable and return the random number directly.

1. Change the `return` statement to return the random number directly.

```
fun roll(): Int {
   return (1..numSides).random()
}
```

In the second print statement, you put the call to get the random number into the string template. You can get rid of the `diceRoll` variable by doing the same thing in the first print statement.

2. Call `myFirstDice.roll()` in the string template and delete the `diceRoll` variable. The first two lines of your `main()` code now look like this.

```
val myFirstDice = Dice(6)
println("Your ${myFirstDice.numSides} sided dice rolled ${myFirstDice.roll()}!")
```

3. Run your code and there should be no difference in the output.

Note: Changing code to make it shorter, more efficient, or easier to read and understand is called *refactoring*. It's like writing a document, where you write a first draft that has all the information, and then edit and refine your words.

This is your final code after *refactoring* it .

```
fun main() {
   val myFirstDice = Dice(6)
   println("Your ${myFirstDice.numSides} sided dice rolled ${myFirstDice.roll()}!")

   val mySecondDice = Dice(20)
   println("Your ${mySecondDice.numSides} sided dice rolled ${mySecondDice.roll()}!")
}

class Dice (val numSides: Int) {

   fun roll(): Int {
      return (1..numSides).random()
   }
}
```

9. Summary

- Call the `random()` function on an `IntRange` to generate a random number: `(1..6).random()`
- Classes are like a blueprint of an object. They can have properties and behaviors, implemented as variables and functions.

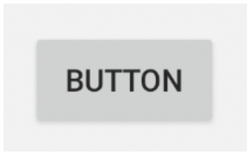
- An instance of a class represents an object, often a physical object, such as a dice. You can call the actions on the object and change its attributes.
- You can supply values to a class when you create an instance. For example: `class Dice(val numSides: Int)` and then create an instance with `Dice(6)`.
- Functions can return something. Specify the data type to be returned in the function definition, and use a `return` statement in the function body to return something. For example: `fun example(): Int { return 5 }`

3. Create the layout for the app

Open the Layout Editor

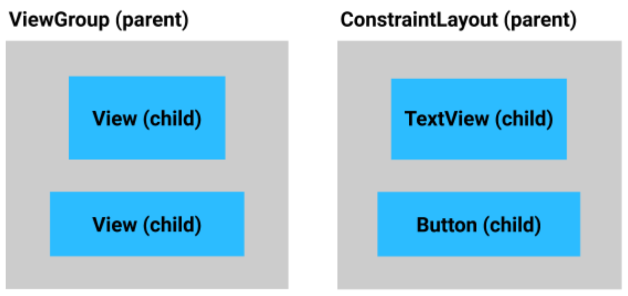
1. In the **Project** window, double-click `activity_main.xml` (**app > res** > **layout** > **activity_main.xml**) to open it. You should see the **Layout Editor**, with only the "Hello World" `TextView` in the center of the app.

   Next you will add a `Button` to your app. A `Button` is a user interface (UI) element in Android that the user can tap to perform an action.



In this task, you add a `Button` below the "Hello World" `TextView`. The `TextView` and the `Button` will be located within a `ConstraintLayout`, which is a type of `ViewGroup`.

When there are `Views` within a `ViewGroup`, the `Views` are considered *children* of the *parent* `ViewGroup`. In the case of your app, the `TextView` and `Button` would be considered children of the parent `ConstraintLayout`.



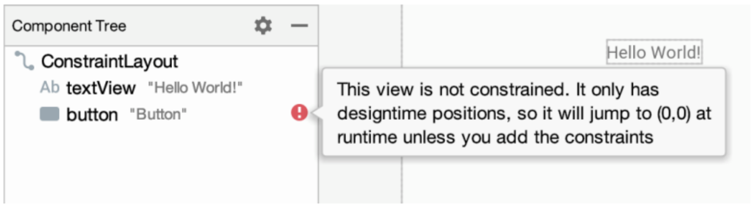
Add a `Button` as a child of the existing `ConstraintLayout` in your app.

Note: Like in a family tree, in a *view hierarchy*, parent views can themselves be child views, and child views can be parents to other children.

Add a Button to the layout

1. Drag a `Button` from the **Palette** onto the **Design** view, positioning it below the "Hello World" `TextView`.
2. Below the **Palette** in the **Component Tree**, verify that the `Button` and `TextView` are listed under the `ConstraintLayout` (as children of the `ConstraintLayout`).

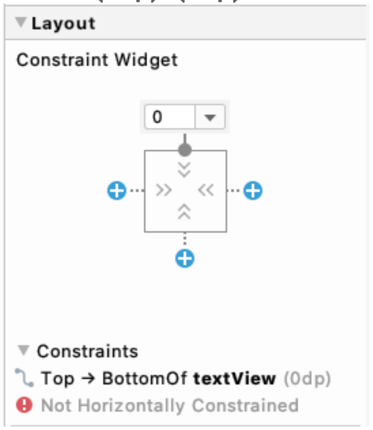
1. Notice an error that the Button is not constrained. Since the Button is sitting within a ConstraintLayout, you must set vertical and horizontal constraints to position it.



This view is not constrained. It only has designtime positions, so it will jump to (0,0) at runtime unless you add the constraints

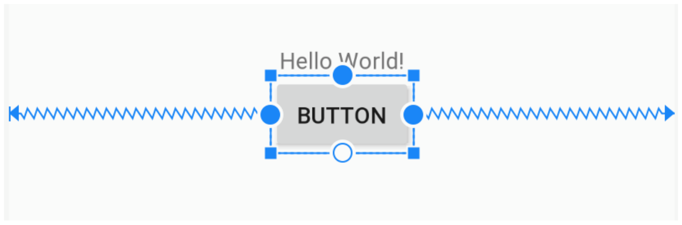
Position the Button

In this step you'll add a vertical constraint from the top of the Button to the bottom of the TextView. This will position the Button below the TextView.

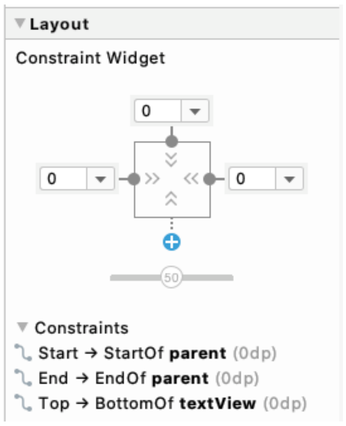
1. In the **Design** view, at the top edge of the Button, press and hold the white circle with a blue border. Drag the pointer, and an arrow will follow the pointer. Release when you reach the bottom edge of the "Hello World" TextView. This establishes a layout constraint, and the Button slides up to just beneath the TextView.
    2. Look at the **Attributes** on the right hand side of the **Layout Editor**.
    3. In the **Constraint Widget,** notice a new layout constraint that is set to the bottom of the TextView, **Top** → **BottomOf textView (0dp)**. **(0dp)** means there is a margin of 0. You also have an error for missing horizontal  constraints.

Top → BottomOf **textView** (0dp)
Not Horizontally Constrained

    4. Add a horizontal constraint from the left side of the Button to the left side of the parent ConstraintLayout.
    5. Repeat on the right side, connecting the right edge of the Button to the right edge of the ConstraintLayout. The result should look like this:



    6. With the Button still selected, the **Constraint Widget** should look like this. Notice two additional constraints that have been added: **Start** → **StartOf parent (0dp)** and **End** → **EndOf parent (0dp)**. This means the Button is horizontally centered in its parent, the ConstraintLayout.



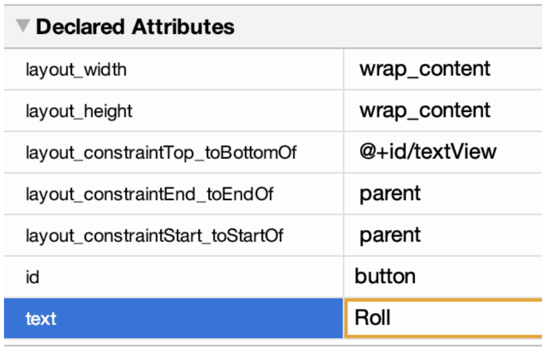
Constraints
Start → StartOf **parent** (0dp)
End → EndOf **parent** (0dp)
Top → BottomOf **textView** (0dp)

Change the Button text

You're going to make a couple more UI changes in the **Layout Editor**.

Instead of having the Button label show "Button", change it to something that indicates what the button is going to do: "Roll".

1. In the **Layout Editor**, with the Button selected, go to **Attributes**, change the **text** to **Roll**, and press the Enter (Return on the Mac) key.

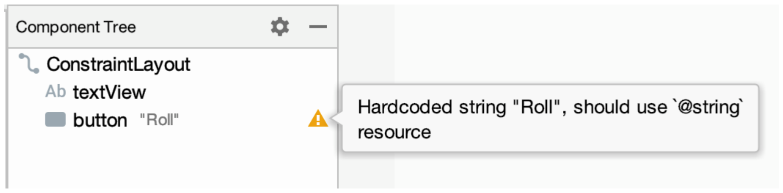


Declared Attributes

layout_width	wrap_content
layout_height	wrap_content
layout_constraintTop_toBottomOf	@+id/textView
layout_constraintEnd_toEndOf	parent
layout_constraintStart_toStartOf	parent
id	button
text	Roll

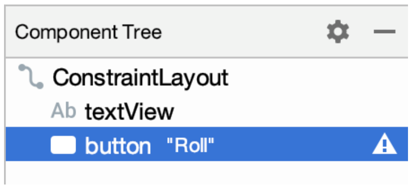
2. In the **Component Tree**, an orange warning triangle appears next to the Button. If you hover the pointer over the triangle, a message appears. Android Studio has detected a *hardcoded string* ("Roll") in your app code and suggests using a *string resource* instead.

   Having a hardcoded string means the app will be harder to translate into other languages and it is harder to reuse strings in different parts of your app. Fortunately, Android Studio has an automatic fix for you.

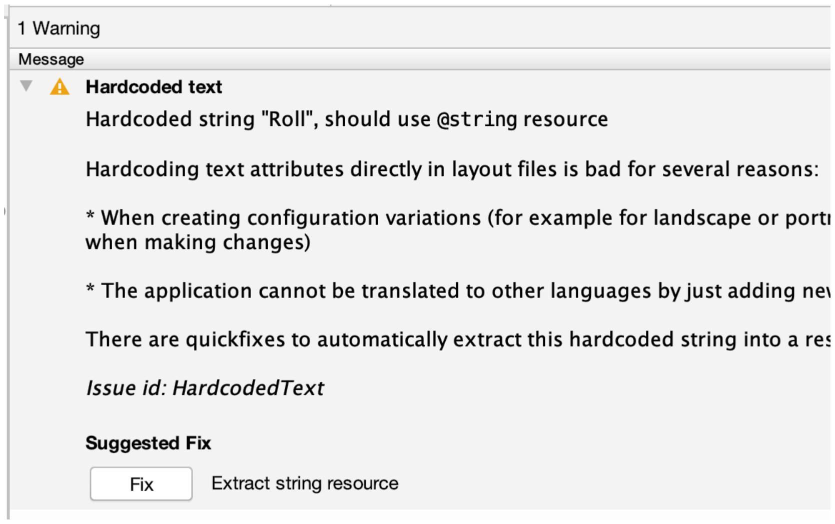


Hardcoded string "Roll", should use `@string` resource

3. In the **Component Tree,** click on the orange triangle.



The full warning message opens.



1 Warning

Message

Hardcoded text

Hardcoded string "Roll", should use @string resource

Hardcoding text attributes directly in layout files is bad for several reasons:

* When creating configuration variations (for example for landscape or portr
when making changes)

* The application cannot be translated to other languages by just adding nev

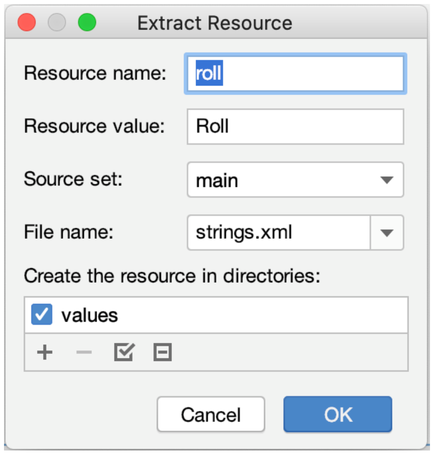
There are quickfixes to automatically extract this hardcoded string into a res

Issue id: HardcodedText

Suggested Fix

Extract string resource

4. At the bottom of the message, under **Suggested Fix**, click the **Fix** button. (You may need to scroll down.)
5. The **Extract Resource** dialog opens. To extract a string means to take the "Roll" text and create a string resource called `roll` in `strings.xml` (**app > res > values > strings.xml**). The default values are correct, so click **OK**.



Extract Resource

Resource name: roll

Resource value: Roll

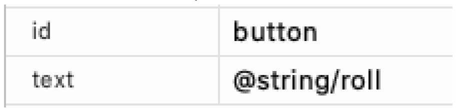
Source set: main

File name: strings.xml

Create the resource in directories:

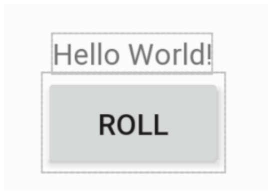
☑ values

6. Notice that in **Attributes**, the **text** attribute for the `Button` now says `@string/roll`, referring to the resource you just created.



id	button
text	@string/roll

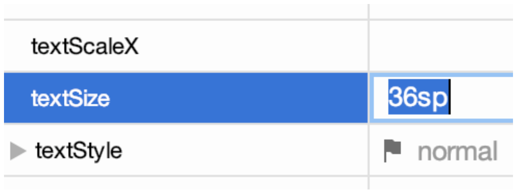
In the **Design** view, the `Button` should still say **Roll** on it.



Style the TextView

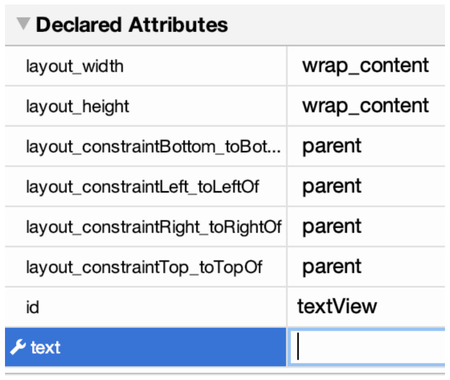
The "Hello World!" text is quite small, and the message isn't relevant for your app. In this step you'll replace the small "Hello, World!" message with a number to show the rolled value, and make the font bigger, so it is easier to see.

1. In the **Design Editor**, select the `TextView` so that its attributes appear in the **Attributes** window.
2. Change the **textSize** of the `TextView` to **36sp**, so that it's large and easy to read. You may need to scroll to find **textSize**.



textScaleX	
textSize	36sp
▶ textStyle	normal

3. Clear the **text** attribute of the `TextView`. You don't need to display anything in the `TextView` until the user rolls the dice.

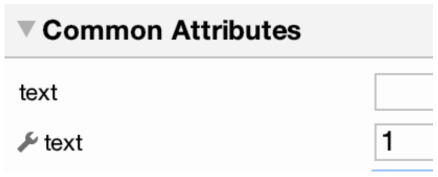


Declared Attributes	
layout_width	wrap_content
layout_height	wrap_content
layout_constraintBottom_toBot...	parent
layout_constraintLeft_toLeftOf	parent
layout_constraintRight_toRightOf	parent
layout_constraintTop_toTopOf	parent
id	textView
text	

However, it's very helpful to see some text in the `TextView` when you're editing the layout and code for your app. For this purpose, you can add text to the `TextView` that is only visible for the layout preview, but not when the app is running

.

4. Select the `TextView` in the **Component Tree**.
5. Under **Common Attributes**, find the **text** attribute, and below it, another **text** attribute with a tool icon. The **text** attribute is what will be displayed to the user when the app is running. The **text** attribute with a tool icon is the "tools text" attribute that is just for you as a developer.
6. Set the tools text to be "1" in the `TextView` (to pretend you have a dice roll of 1). The "1" will only appear in the **Design Editor** within Android Studio, but it will not appear when you run the app on an actual device or emulator.



Common Attributes	
text	
text	1

Note that because this text is only viewed by app developers, you don't need to make a string resource for it.

7. Look at your app in the preview. The "1" is showing.
8. Run your app. This is what the app looks like when it's run on an emulator. The "1" is not showing. This is the correct behavior.

4. Introduction to Activities

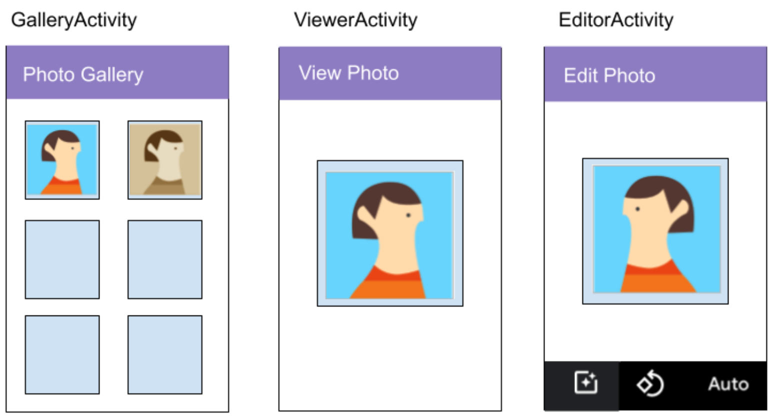
An `Activity` provides the window in which your app draws its UI. Typically, an `Activity` takes up the whole screen of your running app. Every app has one or more activities. The top-level or first activity is often called the `MainActivity` and is provided by the project template. For example, when the user scrolls through the list of apps on their device and taps on the "Dice Roller" app icon, the Android System will start up the `MainActivity` of the app.

In your `MainActivity` code, you need to provide details on the `Activity`'s layout and how the user should interact with it.

- In the Birthday Card app, there is one `Activity` that displays the birthday message and image.
- In the Dice Roller app, there is one `Activity` that displays the `TextView` and `Button` layout you just built.

For more complicated apps, there may be multiple screens and more than one `Activity`. Each `Activity` has a specific purpose.

For example, in a photo gallery app, you could have an `Activity` for displaying a grid of photos, a second `Activity` for viewing an individual photo, and a third `Activity` for editing an individual photo.



Open the MainActivity.kt file

You will add code to respond to a button tap in the `MainActivity`. In order to do this correctly, you need to understand more about the `MainActivity` code that's already in your app.

1. Navigate to and open the `MainActivity.kt` file (**app > java > com.example.diceroller > MainActivity.kt**). Below is what you should see. If you see `import...`, click on the `...` to expand the imports.

```
package com.example.diceroller

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

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

You don't need to understand every single word of the above code, but you do need to have a general idea of what it does. The more you work with Android code, the more familiar it'll become, and the more you'll understand it.

2. Look at the Kotlin code for the `MainActivity` class, identified by the keyword `class` and then the name.

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

3. Notice that there is no `main()` function in your `MainActivity`.

   Earlier, you learned that every Kotlin program must have a `main()` function. Android apps operate differently. Instead of calling a `main()` function, the Android system calls the `onCreate()` method of your `MainActivity` when your app is opened for the first time.

4. Find the `onCreate()` method, which looks like the code below.

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

   You'll learn about `override` in a later codelab (so don't worry about it for now). The rest of the `onCreate()` method sets up the `MainActivity` by using code from the imports and by setting the starting layout with `setContentView()`.

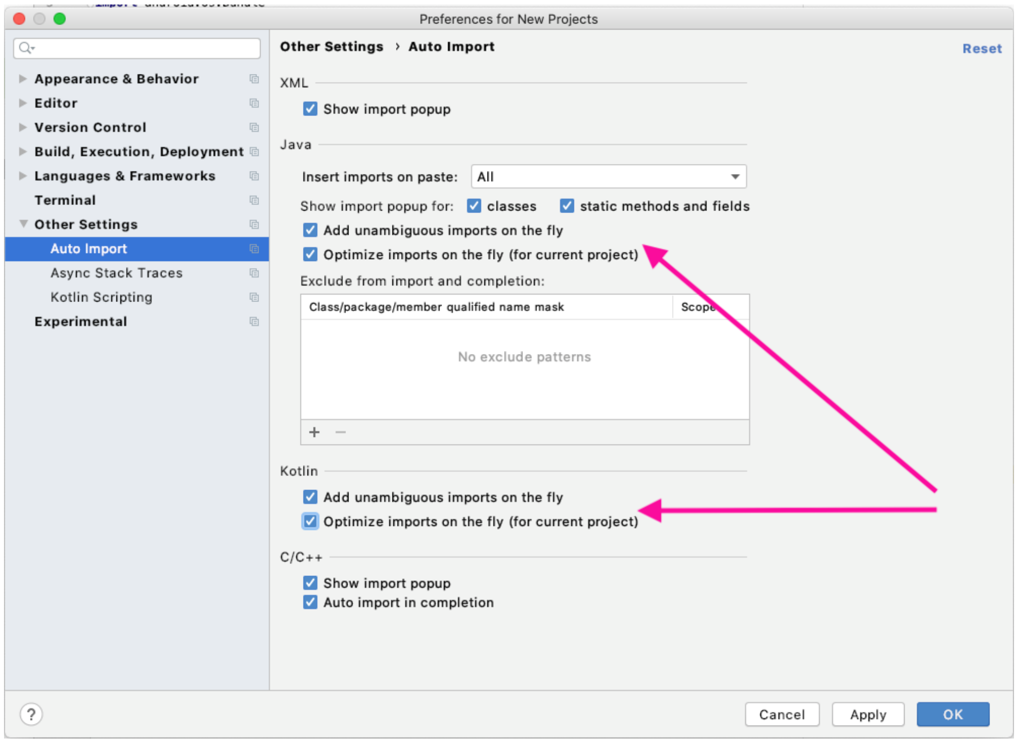
5. Notice the lines beginning with `import`.

   Android provides a *framework* of numerous classes to make writing Android apps easier, but it needs to know exactly which class you mean. You can specify which class in the framework to use in your code by using an `import` statement. For example, the `Button` class is defined in `android.widget.Button`.

Enable auto imports

It can become a lot of work to remember to add `import` statements when you use more classes. Fortunately, Android Studio helps you choose the correct imports when you are using classes provided by others. In this step you'll configure Android Studio to automatically add imports when it can, and automatically remove unused imports from your code.

1. In Android Studio, open the settings by going to **File > New Project Settings > Preferences for New Projects** on macOS. In Windows go to **File > Other Setting > Setting for New Projects...** .
2. Expand **Other Settings > Auto Import**. In the **Java** and **Kotlin** sections, make sure **Add unambiguous imports on the fly** and **Optimize imports on the fly (for current project)** are checked. Note that there are two checkboxes in each


Preferences for New Projects

Other Settings › Auto Import

XML
- ☑ Show import popup

Java
- Insert imports on paste: All
- Show import popup for: ☑ classes ☑ static methods and fields
- ☑ Add unambiguous imports on the fly
- ☑ Optimize imports on the fly (for current project)
- Exclude from import and completion: No exclude patterns

Kotlin
- ☑ Add unambiguous imports on the fly
- ☑ Optimize imports on the fly (for current project)

C/C++
- ☑ Show import popup
- ☑ Auto import in completion

section. The **unambiguous imports** settings tell Android Studio to automatically add an import statement, as long as it can determine which one to use. The **optimize imports** settings tell Android Studio to remove any imports that aren't being used by your code.

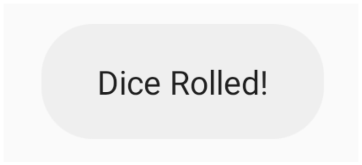
3. Save the changes and close settings by pressing **OK**.

5. Make the Button interactive

Now that you know a little more about the MainActivity, you'll modify the app so that clicking the Button does something on the screen.

Display a message when the Button is clicked

In this step, you will specify that when the button is clicked, a brief message shows at the bottom of the screen.



1. Add the following code to the onCreate() method after the setContentView() call. The findViewById() method finds the Button in the layout. R.id.button is the resource ID for the Button, which is a unique identifier for it. The code saves a *reference* to the Button object in a variable called rollButton, not the Button object itself.

```
val rollButton: Button = findViewById(R.id.button)
```

Note: Android automatically assigns ID numbers to the resources in your app. For example, the **Roll** button has a resource ID, and the string for the button text also has a resource ID. Resource IDs are of the form `R.<type>.<name>`; for example, `R.string.roll`. For `View` IDs, the `<type>` is `id`, for example, `R.id.button`.

The code saves the reference to the `Button` object in a variable called `rollButton`, not the `Button` object itself.

Important: When it assigns an object to a variable, Kotlin doesn't copy the entire object each time, it saves a *reference* to the object. You can think of a reference similar to a national ID number; the number refers to a person, but it isn't the person itself. When you copy the number, you don't copy the person.

The `onCreate()` method should now look like this.

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

   val rollButton: Button = findViewById(R.id.button)
}
```

2. Verify that Android Studio automatically added an `import` statement for the `Button`. Notice there are 3 import statements now—the third one was automatically added.

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
```

Note: If enabling auto imports didn't work, `Button` will be highlighted in red. You can manually add the correct import by putting the text cursor within the word `Button`, then pressing `Alt+Enter` (`Option+Enter` on a Mac).

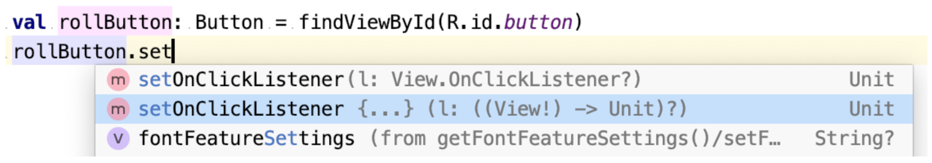
Next you need to associate code with the `Button`, so that the code can be executed when the `Button` is tapped. A *click listener* is some code for what to do when a tap or click happens. You can think of it as code that is just sitting, "listening" for the user to click, in this case, on the `Button`.

3. Use the `rollButton` object and set a click listener on it by calling the `setOnClickListener()` method. Instead of the parentheses following the method name, you will actually be using curly braces following the method name. This is a special syntax for declaring a Lambda, which you'll learn more about in a future codelab.

What you need to know for now is that within the curly braces, you put instructions for what should happen when the button is tapped. You'll have your app display a Toast, which is a brief message in the next step.

```
rollButton.setOnClickListener {
}
```

As you type, Android Studio may show multiple suggestions. For this case, choose the **setOnClickListener {...}** option.



```
val rollButton: Button = findViewById(R.id.button)
rollButton.set
```

m setOnClickListener(l: View.OnClickListener?) Unit
m setOnClickListener {...} (l: ((View!) -> Unit)?) Unit
v fontFeatureSettings (from getFontFeatureSettings()/setF… String?

Within the curly braces, you put instructions for what should happen when the button is tapped. For now, you'll have your app display a Toast, which is a brief message that appears to the user.

4. Create a `Toast` with the text `"Dice Rolled!"` by calling `Toast.makeText()`.

```
val toast = Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT)
```

5. Then tell the `Toast` to display itself by calling the `show()` method.

```
toast.show()
```

This is what your updated the `MainActivity` class looks like; the `package` and `import` statements are still at the top of the file:

```
class MainActivity : AppCompatActivity() {

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

     val rollButton: Button = findViewById(R.id.button)
     rollButton.setOnClickListener {
        val toast = Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT)
        toast.show()
     }
  }
}
```

You could combine the two lines in the click listener into a single line without a variable. This is a common pattern you might find in other code.

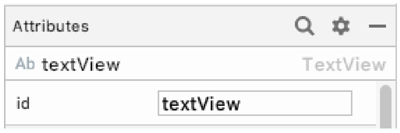
```
Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT).show()
```

6. Run the app and click the **Roll** button. A toast message should pop up at the bottom of the screen and disappear after a short time.

Update the TextView when the Button is clicked

Instead of showing a temporary `Toast` message, you'll write code to update the `TextView` on screen when the **Roll** button is clicked.

1. Go back to `activity_main.xml` (**app > res > layout >activity_main.xml**)
2. Click on the `TextView`.
3. Note that the **id** is **textView**.



4. Open `MainActivity.kt` (**app > java > com.example.diceroller > MainActivity.kt**)
5. Delete the lines of code that create and show the `Toast`.

```
rollButton.setOnClickListener {

}
```

6. In their place, create a new variable called `resultTextView` to store the `TextView`.
7. Use `findViewById()` to find `textView` in the layout using its ID, and store a reference to it.

```
val resultTextView: TextView = findViewById(R.id.textView)
```

8. Set the text on `resultTextView` to be "6" in quotations.

```
resultTextView.text = "6"
```

This is similar to what you did by setting the **text** in **Attributes**, but now it's in your code, so the text needs to be inside double quotation marks. Setting this explicitly means that for now, the `TextView` always displays 6. You'll add the code to roll the dice and show different values in the next task.

This is what the `MainActivity` class should look like:

```
class MainActivity : AppCompatActivity() {

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

     val rollButton: Button = findViewById(R.id.button)
     rollButton.setOnClickListener {
        val resultTextView: TextView = findViewById(R.id.textView)
        resultTextView.text = "6"
     }
  }
}
```

9. Run the app. Click the button. It should update the `TextView` to "6".

6. Add the dice roll logic

The only thing that's missing is actually rolling the dice. You can reuse the `Dice` class from the previous codelab, which handles the logic for rolling a dice.

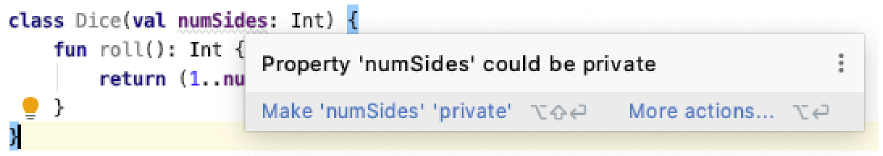
Add the Dice class

1. After the last curly brace in the MainActivity class, create the Dice class with a roll() method.

```
class Dice(val numSides: Int) {

  fun roll(): Int {
     return (1..numSides).random()
  }
}
```

2. Notice that Android Studio may underline numSides with a wavy gray line. (This may take a moment to appear.)
3. Hover your pointer over numSides, and a popup appears saying **Property 'numSides' could be private**.



```
class Dice(val numSides: Int) {
    fun roll(): Int {
        return (1..nu
    }
}
```

Property 'numSides' could be private

Marking numSides as private will make it only accessible within the Dice class. Since the only code that will be using numSides is inside the Dice class, it's okay to make this argument private for the Dice class. You'll learn more about private versus public variables in the next unit.

4. Go ahead and make the suggested fix from Android Studio by clicking **Make 'numSides' 'private'**.

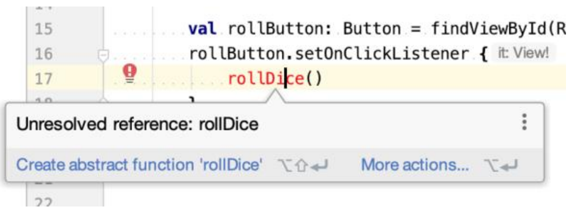
Create a rollDice() method

Now that you've added a Dice class to your app, you'll update MainActivity to use it. To organize your code better, put all the logic about rolling a dice into one function.

1. Replace the code in the click listener that sets the text to "6" with a call to rollDice().

```
rollButton.setOnClickListener {
  rollDice()
}
```

2. Because rollDice() isn't defined yet, Android Studio flags an error and shows rollDice() in red.
3. If you hover your pointer over rollDice(), Android Studio displays the problem and some possible solutions.

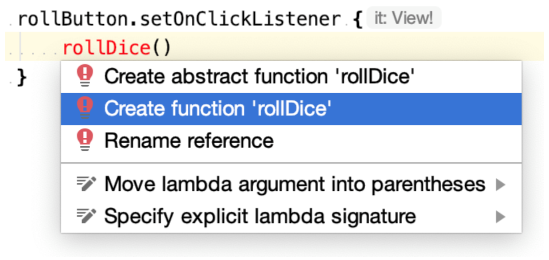


```
val rollButton: Button = findViewById(R
rollButton.setOnClickListener { it: View!
    rollDice()
```

Unresolved reference: rollDice

4. Click on **More actions...** which brings up a menu. Android Studio offers to do more work for you!

Tip: If you find it difficult to hover the pointer and then click on **More actions...** you can click on `rollDice()` and press `Alt+Enter` (`Option+Enter` on a Mac) to bring up the menu.



```
rollButton.setOnClickListener { it: View!
    rollDice()
}
```

Create abstract function 'rollDice'
Create function 'rollDice'
Rename reference
Move lambda argument into parentheses
Specify explicit lambda signature

5. Select **Create function 'rollDice'**. Android Studio creates an empty definition for the function inside `MainActivity`.

```
private fun rollDice() {
    TODO("Not yet implemented")
}
```

Create a new Dice object instance

In this step you'll make the `rollDice()` method create and roll a dice, and then display the result in the `TextView`.

1. Inside `rollDice()`, delete the `TODO()` call.
2. Add code to create a dice with 6 sides.

```
val dice = Dice(6)
```

3. Roll the dice by calling the `roll()` method, and save the result in a variable called `diceRoll`.

```
val diceRoll = dice.roll()
```

4. Find the `TextView` by calling `findViewById()`.

```
val resultTextView: TextView = findViewById(R.id.textView)
```

The variable `diceRoll` is a number, but the `TextView` uses text. You can use the `toString()` method on `diceRoll` to convert it into a string.

5. Convert `diceRoll` to a string and use that to update the text of the `resultTextView`.

```
resultTextView.text = diceRoll.toString()
```

This is what the `rollDice()` method looks like:

```
private fun rollDice() {
    val dice = Dice(6)
    val diceRoll = dice.roll()
    val resultTextView: TextView = findViewById(R.id.textView)
    resultTextView.text = diceRoll.toString()
}
```

6. Run your app. The dice result should change to other values besides 6! Since it is a random number from 1 to 6, the value 6 might appear sometimes, too.

7. Adopt good coding practices

It's normal for your code to look a little messy after you tweak parts here and there to get your app to work. But before you walk away from your code, you should do some easy cleanup tasks. Then the app will be in good shape and easier to maintain going forward.

These habits are what professional Android developers practice when they write their code.

Android Style Guide

As you work on teams, it's ideal for team members to write code in a similar way, so there's some consistency across the code. That is why Android has a Style Guide for how to write Android code—naming conventions, formatting, and other good practices to follow. Conform to these guidelines when you write Android code: Kotlin Style Guide for Android Developers.

Below are a couple of ways you can adhere to the style guide.

Clean up your code

Condense your code

You can make your code more concise by condensing code into a shorter number of lines. For example, here is the code that sets the click listener on the `Button`.

```
rollButton.setOnClickListener {
    rollDice()
}
```

Since the instructions for the click listener are only 1 line long, you can condense the `rollDice()` method call and the curly braces all onto one line. This is what it looks like. One line instead of three lines!

```
rollButton.setOnClickListener { rollDice() }
```

Reformat your code

Now you'll reformat your code to make sure it follows recommended code formatting conventions for Android.

1. In the `MainActivity.kt` class, select all the text in the file with the keyboard shortcut `Control+A` on Windows (or `Command+A` on Mac). Or you could go to the menu in Android Studio **Edit > Select All**.
2. With all the text selected in the file, go to the Android Studio menu **Code > Reformat Code** or use the keyboard shortcut `Ctrl+Alt+L` (or `Command+Option+L` on Mac).

That updates the formatting of your code, which includes whitespace, indentation, and more. You may not see any change, and that's good. Your code was already formatted correctly then!

Comment your code

Add some comments to your code to describe what is happening in the code you wrote. As code gets more complicated, it's also important to note *why* you wrote the code to work the way you did. If you come back to the code later to make changes, *what* the code does may still be clear, but you may not remember why you wrote it the way you did.

It is common to add a comment for each class (`MainActivity` and `Dice` are the only classes you have in your app) and each method you write. Use the `/**` and `**/` symbols at the beginning and end of your comment to tell the system that this is not code. These lines will be ignored when the system executes your code.

Example of a comment on a class:

```
/**
* This activity allows the user to roll a dice and view the result
* on the screen.
*/
class MainActivity : AppCompatActivity() {
```

Example of a comment on a method:

```
/**
* Roll the dice and update the screen with the result.
*/
private fun rollDice() {
```

Within a method, you're free to add comments if that would help the reader of your code. Recall that you can use the `//` symbol at the start of your comment. Everything after the `//` symbol on a line is considered a comment.

Example of 2 comments inside a method:

```
private fun rollDice() {
  // Create new Dice object with 6 sides and roll it
  val dice = Dice(6)
  val diceRoll = dice.roll()

  // Update the screen with the dice roll
```

```
    val resultTextView: TextView = findViewById(R.id.textView)
    resultTextView.text = diceRoll.toString()
}
```

1. Go ahead and take some time to add comments to your code.
2. With all these commenting and formatting changes, it's good practice to run your app again to make sure it still works as expected.

   See the solution code for one way that you could have commented your code.

9. Summary

- Add a `Button` in an Android app using the **Layout Editor**.
- Modify the `MainActivity.kt` class to add interactive behavior to the app.
- Pop up a `Toast` message as a temporary solution to verify you're on the right track.
- Set an on-click listener for a `Button` using `setOnClickListener()` to add behavior for when a `Button` is clicked.
- When the app is running, you can update the screen by calling methods on the `TextView`, `Button`, or other UI elements in the layout.
- Comment your code to help other people who are reading your code understand what your approach was.
- Reformat your code and clean up your code.

3. Create the Lucky Dice Roll game

In this section, using what you learned in the previous task, you will update the Dice Roller program to check whether you have rolled a preset lucky number. If you have, you win!

Set up your starter code

You are starting the Lucky Dice Roller with code that is similar to the solution code of the previous Kotlin Dice Roller program. You can edit the `main()` function in your previous code to match, or you can copy and paste the code below to get started.

```
fun main() {
   val myFirstDice = Dice(6)
   val diceRoll = myFirstDice.roll()
   println("Your ${myFirstDice.numSides} sided dice rolled ${diceRoll}!")
}

class Dice (val numSides: Int) {

   fun roll(): Int {
       return (1..numSides).random()
   }
}
```

Check if the lucky number has been rolled

Create a lucky number first, and then compare the dice roll with that number.

1. In `main()`, delete the `println()` statement.
2. In `main()`, add a `val` called `luckyNumber` and set it to 4. Your code should look like this.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4
}
```

3. Below, add an `if` statement with a condition inside the parentheses `()` that checks if `rollResult` is equal (`==`) to `luckyNumber`. Leave some room between the curly braces `{}` so you can add more code.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4
    if (rollResult == luckyNumber) {

    }
}
```

4. Inside the curly braces `{}`, add a `println` statement to print `"You win!"`

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4

    if (rollResult == luckyNumber) {
        println("You win!")
    }
}
```

5. Run your program. You may need to run it several times before you get lucky and see the winning message in the output!

```
You win!
```

Respond when the lucky number has not been rolled

Getting no feedback from the program if the user didn't win can leave them wondering if the program is broken. It is a good practice to always provide a response when the user does something. For the Lucky Dice Roller program, you can let them know they didn't win by using an `else` statement.

1. Add an `else` statement to print `"You didn't win, try again!"`.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4

    if (rollResult == luckyNumber) {
        println("You win!")
    } else {
        println("You didn't win, try again!")
    }
}
```

2. Run the program, and no matter the outcome, your users are always notified.

   At this point, users know if they won or not, but not why. Always give users information so they understand the result of their actions! Imagine your program was a loan application. "You didn't get approved because your credit rating is poor," is a lot more informative than, "Sorry, no loan for you, try again!" For Lucky Dice Roller, you can give users a different informative message for each roll if they lost. Use multiple `else if` statements to accomplish this.

3. Add `else if` statements to print a different message for each roll. Refer to the format you learned in the previous task, if necessary.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4

    if (rollResult == luckyNumber) {
        println("You win!")
    } else if (rollResult == 1) {
        println("So sorry! You rolled a 1. Try again!")
    } else if (rollResult == 2) {
        println("Sadly, you rolled a 2. Try again!")
    } else if (rollResult == 3) {
        println("Unfortunately, you rolled a 3. Try again!")
    } else if (rollResult == 4) {
        println("No luck! You rolled a 4. Try again!")
    } else if (rollResult == 5) {
        println("Don't cry! You rolled a 5. Try again!")
    } else {
        println("Apologies! you rolled a 6. Try again!")
    }
}
```

In the code above, you

- Check whether the `rollResult` is the `luckyNumber`.
- If the `rollResult` is the `luckyNumber`, print the winning message.
- Otherwise, check whether the `rollResult` is 1, and if so, print a try again message.
- Otherwise, check whether the `rollResult` is 2, and if so, print a different try again message.
- Otherwise, keep checking through the number 5.
- If the number hasn't been any of 1 - 5, the only option left is 6, so there is no need for another test with `else if`, and you can just catch that last option with the final `else` statement.

Tip: You can have only one **`if`** statement with one **`else`** statement in an if-else code block, but in between, you can have as many **`else if`** statements as you need.

Because having multiple `else if` cases is very common, Kotlin has a simpler way of writing them.

4. Use a when statement

Testing for many different outcomes, or cases, is very common in programming. Sometimes, the list of possible outcomes can be very long. For example, if you were rolling a 12-sided dice, you'd have 11 `else if` statements between the success and the final `else`. To make these kinds of statements easier to write and read, which helps avoid errors, Kotlin makes available a `when` statement.

You are going to change your program to use a `when` statement. A `when` statements starts with the keyword `when`, followed by parentheses `()`. Inside the parentheses goes the value to test. This is followed by curly braces `{}` for the code to execute for different conditions.

1. In your program, in `main()`, select the code from the first `if` statement to the curly brace `}` that closes the last `else` statement and delete it.

```
fun main() {
   val myFirstDice = Dice(6)
   val rollResult = myFirstDice.roll()
   val luckyNumber = 4
}
```

2. In `main()`, below the declaration of `luckyNumber`, create a `when` statement. Because your `when` needs to test against the rolled result, put `rollResult` in between the parentheses `()`. Add curly braces `{}` with some extra spacing, as shown below.

```
fun main() {
   val myFirstDice = Dice(6)
   val rollResult = myFirstDice.roll()
   val luckyNumber = 4

   when (rollResult) {

   }
}
```

As before, first test whether `rollResult` is the same as the `luckyNumber`.

1. Inside the curly braces `{}` of the `when` statement, add a statement that tests `rollResult` against `luckyNumber`, and if they are the same, print the winning message. The statement looks like this:

```
luckyNumber -> println("You win!")
```

This means:

- You first put the value you are comparing to `rollResult`. That's `luckyNumber`.
- Follow that with an arrow (`->`).
- Then add the action to perform if there is a match.

Read this as, "If `rollResult` is `luckyNumber`, then print the `"You win!"` message."

And your `main()` code looks like this.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4

    when (rollResult) {
        luckyNumber -> println("You win!")
    }
}
```

2. Use the same pattern to add lines and messages for the possible rolls 1 - 6, as shown below. Your finished `main()` function should look like this.

```
fun main() {
    val myFirstDice = Dice(6)
    val rollResult = myFirstDice.roll()
    val luckyNumber = 4
    when (rollResult) {
        luckyNumber -> println("You won!")
        1 -> println("So sorry! You rolled a 1. Try again!")
        2 -> println("Sadly, you rolled a 2. Try again!")
        3 -> println("Unfortunately, you rolled a 3. Try again!")
        4 -> println("No luck! You rolled a 4. Try again!")
        5 -> println("Don't cry! You rolled a 5. Try again!")
        6 -> println("Apologies! you rolled a 6. Try again!")
    }
}
```

3. Run your program. There is no difference in the output, but your code is much more compact and easier to read.

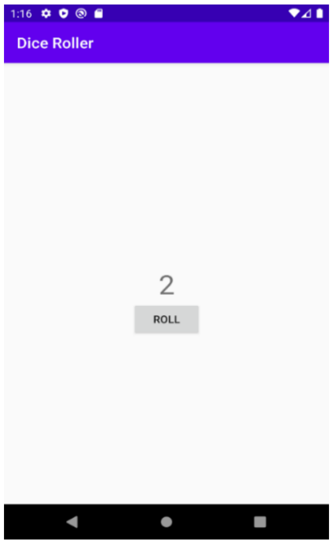
2. Update the layout for the app

In this task, you'll replace the `TextView` in your layout with an `ImageView` that displays an image of the dice roll result.

Open Dice Roller app

1. Open and run the Dice Roller app from the previous codelab in Android Studio. You can use the solution code or the code you created.

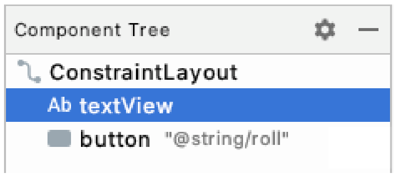
   The app should look like this.



2. Open `activity_main.xml` (**app > res > layout > activity_main.xml**). This opens the **Layout Editor**.

Delete the TextView

1. In the **Layout Editor**, select the `TextView` in the **Component Tree**.

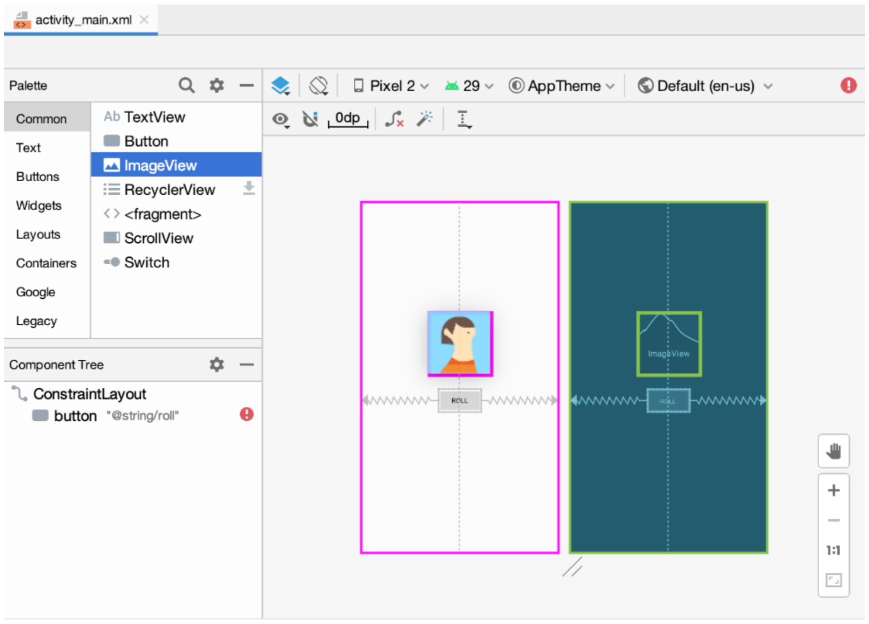


Tip: As you add more UI components and are adding and removing constraints, you may temporarily find one **`View`** overlapping another, making it hard to select the one in back. In that case, you can select a **`View`** by selecting it in the **Component Tree** instead.

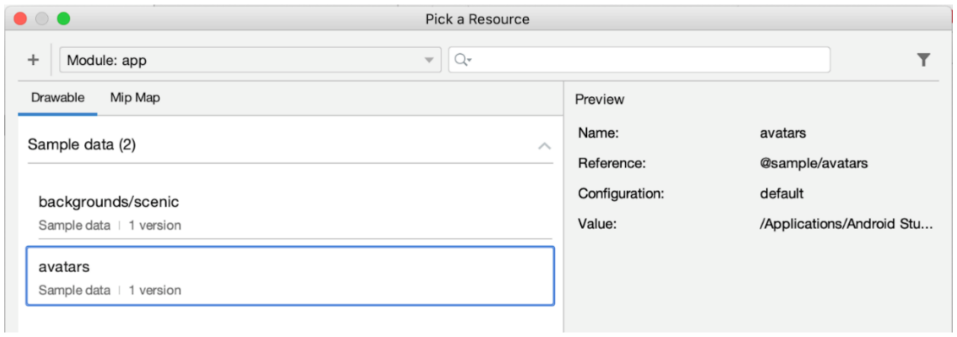
2. Right-click and choose **Delete** or press the `Delete` key.
3. Ignore the warning on the `Button` for now. You'll fix that in the next step.

Add an ImageView to the layout

1. Drag an `ImageView` from the **Palette** onto the **Design** view, positioning it above the `Button`.

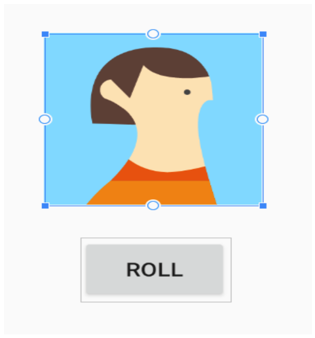


2. In the **Pick a Resource** dialog, select **avatars** under **Sample data**. This is the temporary image you will use until you add the dice images in the next task.

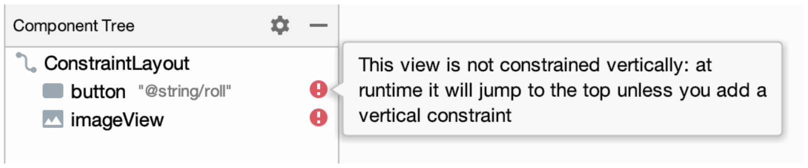


Name:	avatars
Reference:	@sample/avatars
Configuration:	default
Value:	/Applications/Android Stu...

3. Press **OK**. The **Design** view of your app should look like this.



4. In the **Component Tree**, you will notice two errors. The `Button` is not vertically constrained, and the `ImageView` is neither vertically nor horizontally constrained.



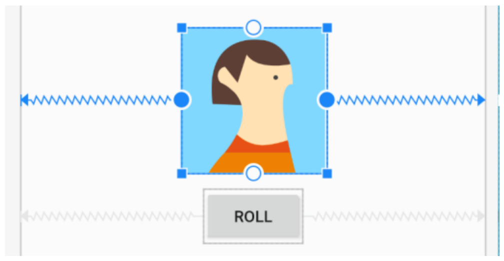
This view is not constrained vertically: at runtime it will jump to the top unless you add a vertical constraint

The `Button` is not vertically constrained because you removed the `TextView` below which it was originally positioned. Now you need to position the `ImageView` and the `Button` below it.

Position the ImageView and Button

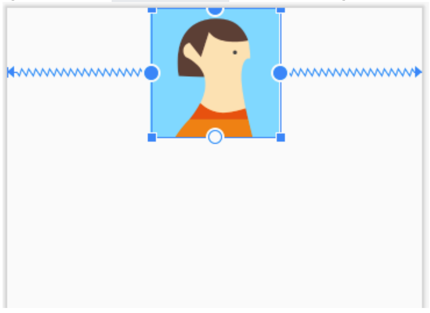
You need to vertically center the ImageView in the screen, regardless of where the Button is located.

1. Add horizontal constraints to the ImageView. Connect the left side of the ImageView to the left edge of the parent ConstraintLayout.
2. Connect the right side of the ImageView to the right edge of the parent. This will horizontally center the ImageView within the parent.



3. Add a vertical constraint to the ImageView, connecting the top of the ImageView to the top of the parent.

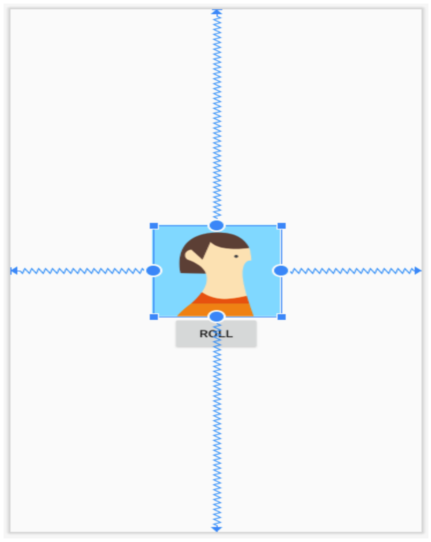
   The ImageView will slide up to the top of the ConstraintLayout.



4. Add a vertical constraint to the Button, connecting the top of the Button to the bottom of the ImageView. The Button will slide up beneath the ImageView.
5. Now select the ImageView again and add a vertical constraint connecting the bottom of the ImageView to the bottom of the parent. This centers the ImageView vertically in the ConstraintLayout.

   All the warnings about constraints should now be gone.

   After all that, the **Design** view should look like this, with the ImageView in the center and the Button just below it.



You may notice a warning on the `ImageView` in the **Component Tree** that says to add a content description to your `ImageView`. Don't worry about this warning for now because later in the codelab, you will be setting the content description of the `ImageView` based on what dice image you're displaying. This change will be made in the Kotlin code.

3. Add the dice images

In this task, you'll download some dice images and add them to your app.

Important! - You will be able to refer to these images in your Kotlin code with their resource IDs:

- `R.drawable.dice_1`
- `R.drawable.dice_2`
- `R.drawable.dice_3`
- `R.drawable.dice_4`
- `R.drawable.dice_5`
- `R.drawable.dice_6`

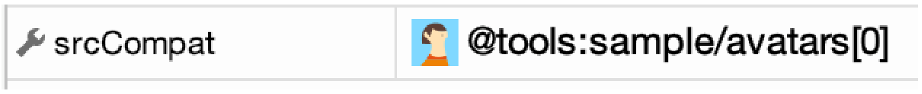
4. Use the dice images

Replace the sample avatar image

1. In the **Design Editor**, select the `ImageView`.
2. In **Attributes** in the **Declared Attributes** section, find the tool **srcCompat** attribute, which is set to the avatar image.

   Remember that the tools **srcCompat** attribute uses the provided image only inside the **Design** view of Android Studio. The image is only displayed to developers as you build the app, but will not be seen when you actually run the app on the emulator or on a device.

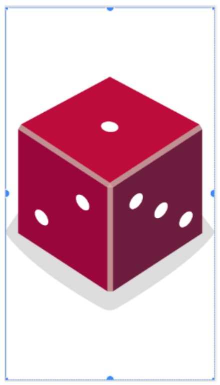
3. Click the tiny preview of the avatar. This opens a dialog to pick a new resource to use for this `ImageView`.



srcCompat @tools:sample/avatars[0]

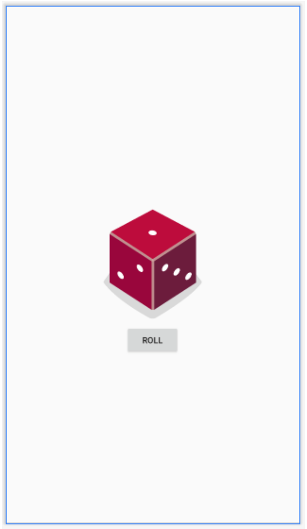
4. Select the `dice_1` drawable and click **OK**.

   Whoa! The `ImageView` takes up the whole screen.



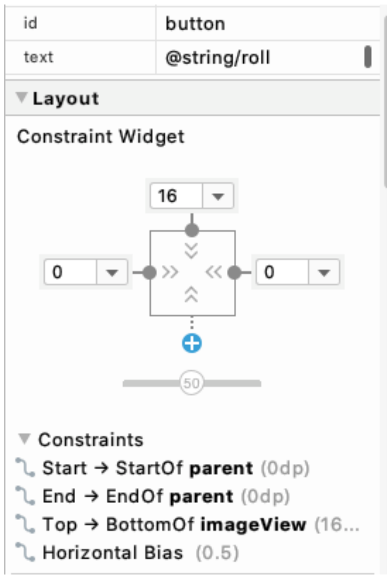
Next, you'll adjust the width and height of the ImageView, so it doesn't hide the Button.

5. In the **Attributes** window under the **Constraints Widget**, locate the **layout_width** and **layout_height** attributes. They are currently set to **wrap_content**, meaning that the ImageView will be as tall and as wide as the content (the source image) inside it.
6. Instead, set a fixed width of 160dp and fixed height of 200dp on the ImageView. Press **Enter**.



The ImageView is much smaller now.

You might find the Button is a little too close to the image.



id	button
text	@string/roll

Layout

Constraint Widget

16

0 0

50

Constraints

Start → StartOf **parent** (0dp)

End → EndOf **parent** (0dp)

Top → BottomOf **imageView** (16...

Horizontal Bias (0.5)

7. Add a top margin to the button of 16dp by setting it in the **Constraint Widget**.

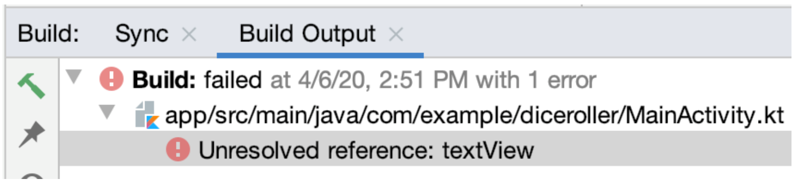
Once the **Design** view updates, the app looks much better!

Note: Use density independent pixels (dp) as the unit to define these dimensions so that the image size scales appropriately on devices with different pixel resolution.

Change the dice image when the button is clicked

The layout has been fixed, but the MainActivity class needs to be updated to use the dice images.

There is currently an error in the app in the MainActivity.kt file. If you try to run the app, you'll see this build error:

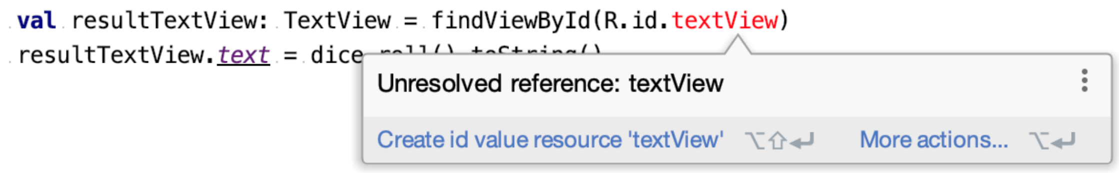


```
Build: failed at 4/6/20, 2:51 PM with 1 error
app/src/main/java/com/example/diceroller/MainActivity.kt
Unresolved reference: textView
```

This is because your code is still referencing the TextView that you deleted from the layout.

1. Open MainActivity.kt (**app > java > com.example.diceroller > MainActivity.kt**)

   The code refers to R.id.textView, but Android Studio doesn't recognize it.



```
val resultTextView: TextView = findViewById(R.id.textView)
resultTextView.text = dice.roll().toString()
```

```
Unresolved reference: textView
```

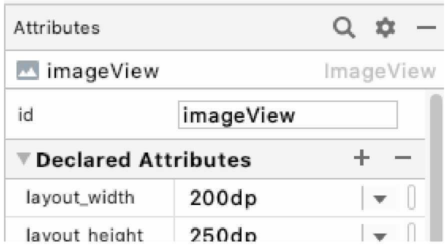
2. Within the rollDice() method, select any code that refers to TextView and delete it.

```
// Update the TextView with the dice roll
val resultTextView: TextView = findViewByID(R.id.textView)
resultTextView.text = dice.roll().toString()
```

3. Still within rollDice(), create a new variable called diceImage of type ImageView. Set it equal to the ImageView from the layout. Use the findViewById() method and pass in the resource ID for the ImageView, R.id.imageView, as the input argument.

```
val diceImage: ImageView = findViewById(R.id.imageView)
```

   If you're wondering how to figure out the precise resource ID of the ImageView, check the **id** at the top of



imageView	ImageView
id	imageView
layout_width	200dp
layout_height	250dp

   the **Attributes** window.

   When you refer to this resource ID in Kotlin code, make sure you type it exactly the same (lowercase i, capital V, no spaces). Otherwise Android Studio will show an error.

4. Add this line of code to test that you can correctly update the ImageView when the button is clicked. The dice roll will not always be "2" but just use the dice_2 image for testing purposes.

```
diceImage.setImageResource(R.drawable.dice_2)
```

This code calls the `setImageResource()` method on the `ImageView`, passing the resource ID for the `dice_2` image. This will update the `ImageView` on screen to display the `dice_2` image.

The rollDice() method should look like this now:

```
private fun rollDice() {
    val dice = Dice(6)
    val diceRoll = dice.roll()
    val diceImage: ImageView = findViewById(R.id.imageView)
    diceImage.setImageResource(R.drawable.dice_2)
}
```

5. Run your app to verify that it runs without errors.

 The app should start off with a blank screen except for the **Roll** button.

 Once you tap the button, a dice image displaying the value 2 will appear.

5. Display the correct dice image based on the dice roll

Clearly the dice result won't always be a 2. Use the control flow logic that you learned in the Add Conditional Behavior for Different Dice Rolls codelab so that the appropriate dice image will be displayed on screen depending on the random dice roll.

Before you start to type code, think conceptually about how the app should behave by writing some *pseudocode* that describes what should happen. For example:

If the user rolls a 1, then display the dice_1 image.

If the user rolls a 2, then display the dice_2 image.

etc...

Note: *Pseudocode* is an informal description of how some code might work. It uses some elements of computer language like **`if / else`**, but describes things in a human understandable way. It can be useful for planning the correct approach to take before all the details have been decided.

The above pseudocode can be written with if / else statements in Kotlin based on the value of the dice roll.

```
if (diceRoll == 1) {
   diceImage.setImageResource(R.drawable.dice_1)
} else if (diceRoll == 2) {
   diceImage.setImageResource(R.drawable.dice_2)
}
 ...
```

Writing if / else for each case gets pretty repetitive, though. The same logic can be expressed more simply with a when statement. This is more concise (less code)! Use this approach in your app.

```
when (diceRoll) {
   1 -> diceImage.setImageResource(R.drawable.dice_1)
   2 -> diceImage.setImageResource(R.drawable.dice_2)
```

```
...
```

Update the rollDice() method

1. In the `rollDice()` method, delete the line of code that sets the image resource ID to `dice_2` image every time.

```
diceImage.setImageResource(R.drawable.dice_2)
```

2. Replace it with a `when` statement that updates the `ImageView` based on the `diceRoll` value.

```
  when (diceRoll) {
     1 -> diceImage.setImageResource(R.drawable.dice_1)
     2 -> diceImage.setImageResource(R.drawable.dice_2)
     3 -> diceImage.setImageResource(R.drawable.dice_3)
     4 -> diceImage.setImageResource(R.drawable.dice_4)
     5 -> diceImage.setImageResource(R.drawable.dice_5)
     6 -> diceImage.setImageResource(R.drawable.dice_6)
  }
```

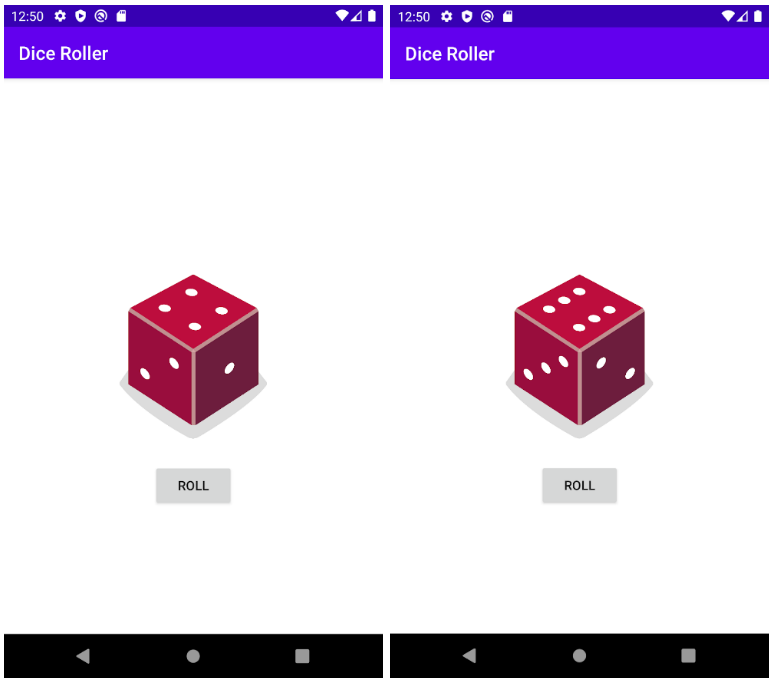
The `rollDice()` method should look like this when you're done with the changes.

```
private fun rollDice() {
  val dice = Dice(6)
  val diceRoll = dice.roll()

  val diceImage: ImageView = findViewById(R.id.imageView)

  when (diceRoll) {
     1 -> diceImage.setImageResource(R.drawable.dice_1)
     2 -> diceImage.setImageResource(R.drawable.dice_2)
     3 -> diceImage.setImageResource(R.drawable.dice_3)
     4 -> diceImage.setImageResource(R.drawable.dice_4)
     5 -> diceImage.setImageResource(R.drawable.dice_5)
     6 -> diceImage.setImageResource(R.drawable.dice_6)
  }
}
```

3. Run the app. Clicking the **Roll** button changes the dice image to other values aside from 2. It works!



Optimize your code

If you want to write even more concise code, you could make the following code change. It doesn't have any visible impact to the user of your app, but it will make your code shorter and less repetitive.

You may have noticed that the call to `diceImage.setImageResource()`appears 6 times in your when statement.

```
when (diceRoll) {
    1 -> diceImage.setImageResource(R.drawable.dice_1)
    2 -> diceImage.setImageResource(R.drawable.dice_2)
    3 -> diceImage.setImageResource(R.drawable.dice_3)
    4 -> diceImage.setImageResource(R.drawable.dice_4)
    5 -> diceImage.setImageResource(R.drawable.dice_5)
    6 -> diceImage.setImageResource(R.drawable.dice_6)
}
```

The only thing that changes between each case is the resource ID that's being used. That means you can create a variable to store the resource ID to use. Then you can call `diceImage.setImageResource()`only once in your code and pass in the correct resource ID.

1. Replace the code above with the following.

```
val drawableResource = when (diceRoll) {
   1 -> R.drawable.dice_1
   2 -> R.drawable.dice_2
   3 -> R.drawable.dice_3
   4 -> R.drawable.dice_4
   5 -> R.drawable.dice_5
   6 -> R.drawable.dice_6
}
```

```
diceImage.setImageResource(drawableResource)
```

A new concept here is that a `when` expression can actually return a value. With this new code snippet, the `when` expression returns the correct resource ID, which will be stored in the `drawableResource` variable. Then you can use that variable to update the image resource displayed.

2. Notice that `when` is now underlined in red. If you hover your pointer over it, you'll see an error message: **'when' expression must be exhaustive, add necessary 'else' branch**.

   The error is because the value of the `when` expression is assigned to `drawableResource`, so the `when` must be exhaustive—it must handle all the cases possible so that a value is always returned, even if you change to a 12-sided dice. Android Studio suggests adding an `else` branch. You can fix this by changing the case for `6` to `else`. The cases for `1` through `5` are the same, but all others including `6` are handled by the `else`.

```
val drawableResource = when (diceRoll) {
   1 -> R.drawable.dice_1
   2 -> R.drawable.dice_2
   3 -> R.drawable.dice_3
   4 -> R.drawable.dice_4
   5 -> R.drawable.dice_5
   else -> R.drawable.dice_6
}

diceImage.setImageResource(drawableResource)
```

3. Run the app to make sure it still works correctly. Be sure to test it enough to make sure that you see all the numbers appear with the dice images 1 through 6.

Set an appropriate content description on the ImageView

Now that you've replaced the rolled number with an image, screen readers cannot tell anymore what number was rolled. To fix this, after you've updated the image resource, update the content description of the `ImageView`. The content description should be a text description of what is shown in the `ImageView` so that screen readers can describe it.

```
diceImage.contentDescription = diceRoll.toString()
```

Screen readers can read aloud this content description, so if the dice roll of "6" image is displayed on the screen, the content description would be read out loud as "6".

Note: Normally, a content description should use string resources that can be translated into other languages, but we'll get into this in a future lesson.

6. Adopt good coding practices

Create a more useful launch experience

When the user opens the app for the first time, the app is blank (except the **Roll** button), which looks odd. Users may not know what to expect, so change the UI to display a random dice roll when you first start the app and create the `Activity`. Then users are more likely to understand that tapping the **Roll** button will produce a dice roll.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val rollButton: Button = findViewById(R.id.button)
    rollButton.setOnClickListener { rollDice() }
    // Do a dice roll when the app starts
    rollDice()
}
```

Comment Your Code

Add some comments to your code to describe what is happening in the code you wrote.

After you've made all these changes, this is what your `rollDice()` method might look like.

```
/**
* Roll the dice and update the screen with the result.
*/
private fun rollDice() {
    // Create new Dice object with 6 sides and roll the dice
    val dice = Dice(6)
    val diceRoll = dice.roll()

    // Find the ImageView in the layout
    val diceImage: ImageView = findViewById(R.id.imageView)

    // Determine which drawable resource ID to use based on the dice roll
    val drawableResource = when (diceRoll) {
        1 -> R.drawable.dice_1
        2 -> R.drawable.dice_2
        3 -> R.drawable.dice_3
        4 -> R.drawable.dice_4
        5 -> R.drawable.dice_5
        else -> R.drawable.dice_6
    }
    // Update the ImageView with the correct drawable resource ID
    diceImage.setImageResource(drawableResource)
    // Update the content description
    diceImage.contentDescription = diceRoll.toString()
}
```