About this codelab

Last updated Jun 18, 2020

Written by Google Developers Training team

1. Welcome

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page (https://codelabs.developers.google.com/android-kotlin-fundamentals/).

Introduction

In this codelab, you learn how to install Android Studio, Google's Android development environment.

What you'll need

- A computer running Windows or Linux, or a Mac running macOS. Make sure that your system meets the latest system requirements (https://developer.android.com/studio/index.html#Requirements).
- Several gigabytes of free storage disk space in which to install and run Android Studio and related SDK and emulator images.
- Internet access, or an alternative way to load the latest Android Studio installation files onto your computer.

What you'll learn

How to install and start the Android Studio integrated development environment (IDE).

Next (#)

Android Studio provides a complete IDE, including an advanced code editor and app templates. It also contains tools for development, debugging, testing, and performance that make it faster and easier to develop apps. You can use Android Studio to test your apps with a large range of preconfigured emulators, or on your own mobile device. You can also build production apps and publish apps on the Google Play store.

> **Note:** Android Studio is continually being improved. For the latest information on system requirements and installation instructions, see the Android Studio download page (https://developer.android.com/studio/).

Android Studio is available for computers running Windows or Linux, and for Macs running macOS. The newest OpenJDK (Java Development Kit) is bundled with Android Studio.

The installation is similar for all platforms. Any differences are noted below.

1. Navigate to the Android Studio download page (https://developer.android.com/studio/) and follow the instructions to download and install Android Studio (https://developer.android.com/studio/install.html).
2. Accept the default configurations for all steps, and ensure that all components are selected for installation.
3. After the install is complete, the setup wizard downloads and installs additional components, including the Android SDK. Be patient, because this process might take some time, depending on your internet speed.
4. When the installation completes, Android Studio starts, and you are ready to create your first project.

> **Troubleshooting:** If you run into problems with your installation, see the Android Studio release notes (https://developer.android.com/studio/releases/index.html) or Troubleshoot Android Studio (https://developer.android.com/studio/troubleshoot).

Back (#) Next (#)

Android Kotlin Fundamentals: Get started

About this codelab

subject Last updated Jun 18, 2020

account_circle Written by Google Developers Training team

1. Welcome

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

Introduction

In this codelab you create and run your first Android app, HelloWorld, on an emulator and on a physical device. You also explore what an Android project looks like.

What you should already know

- You should understand the general software development process for object-oriented apps using an IDE (integrated development environment) such as Android Studio.
- You should have at least one year of experience in object-oriented programming, with at least an understanding of Java and Kotlin.

What you'll learn

How to build a basic Android app in Android Studio.

How to create an Android project from a template.

How to find the major components of an Android project.

How to run an Android app on an emulator or physical device.

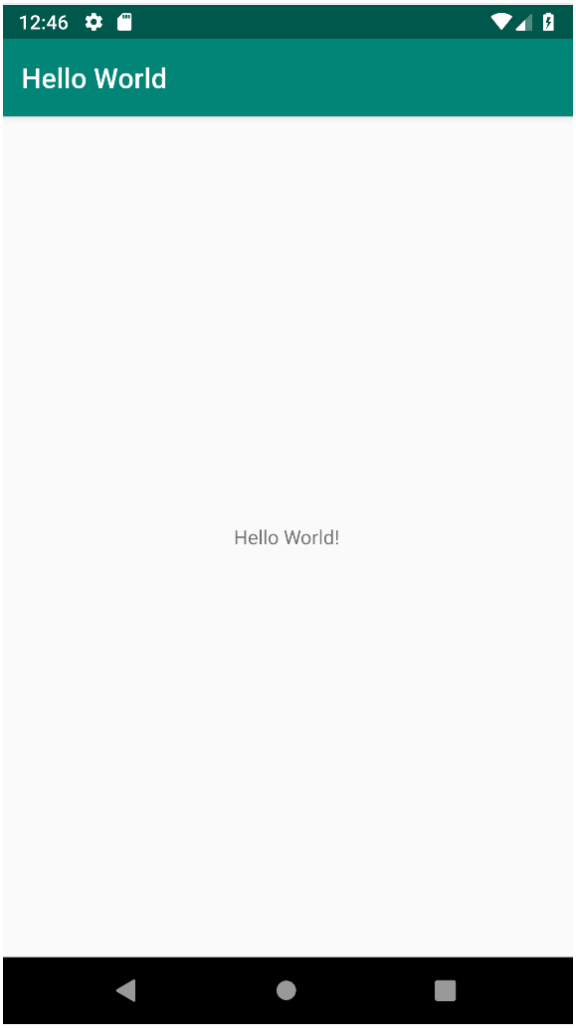
What you'll **do**

- Create a new Android project and a default app called HelloWorld.
- Create an emulator (a virtual device) so you can run your app on your computer.
- Run the HelloWorld app on virtual and physical devices.
- Explore the project layout.

- Explore the `AndroidManifest.xml` file.

2. App overview

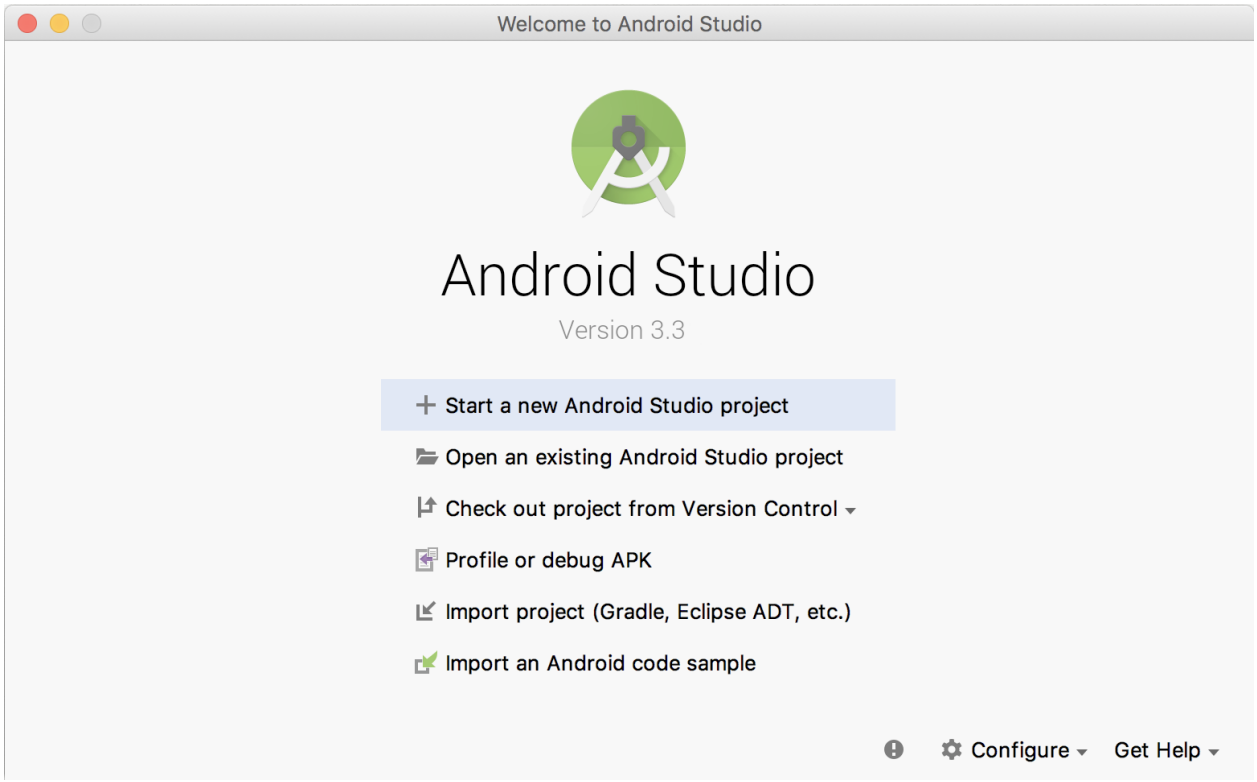
The HelloWorld app displays the string "Hello World" on the screen of an Android virtual device or physical device. Here's what the app looks like:



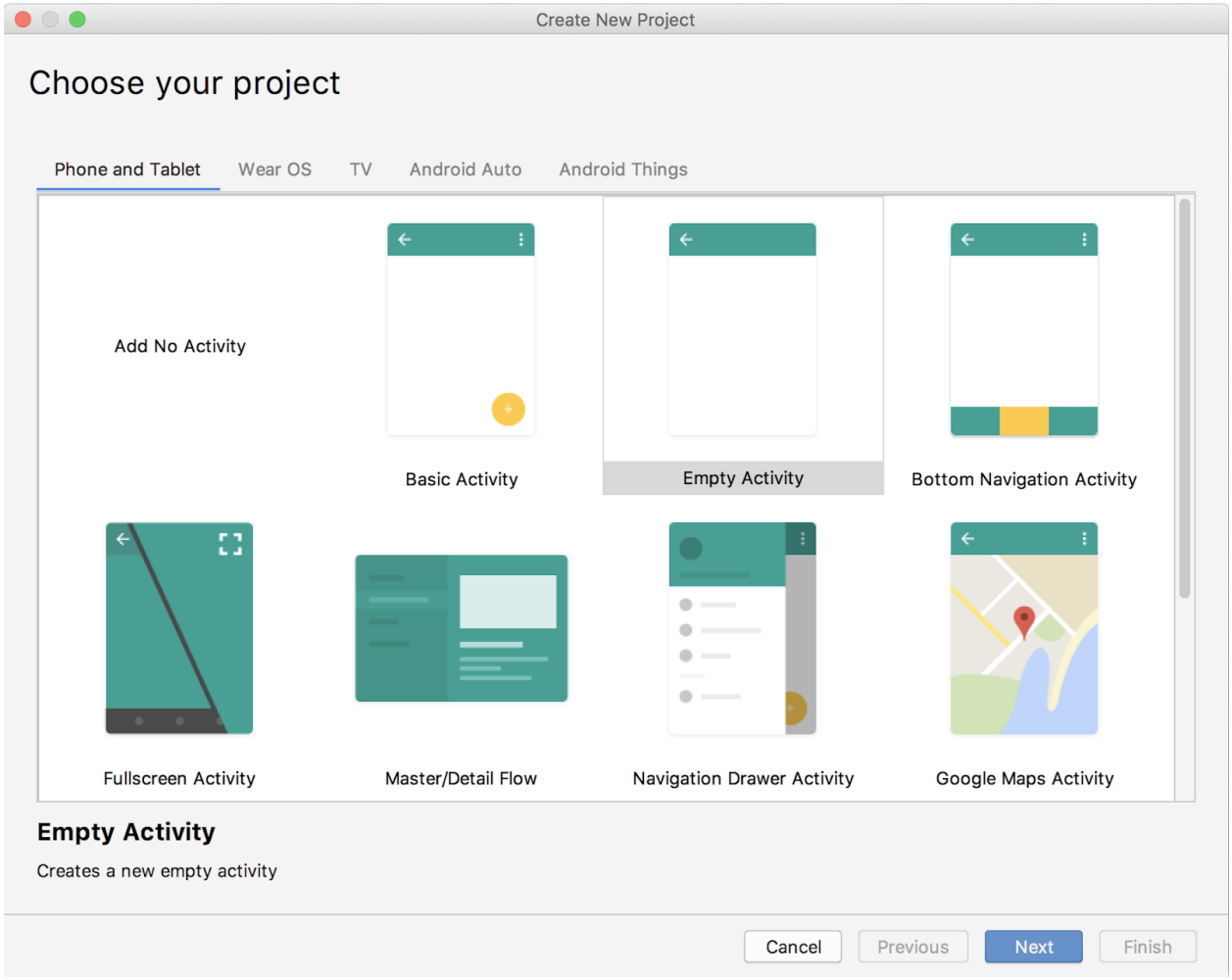
3. Task: Create the HelloWorld project

In this task, you create a new app project to verify that Android Studio is correctly installed.

1. Open Android Studio if it is not already opened.
2. In the main **Welcome to Android Studio** dialog, click **Start a new Android Studio project**.

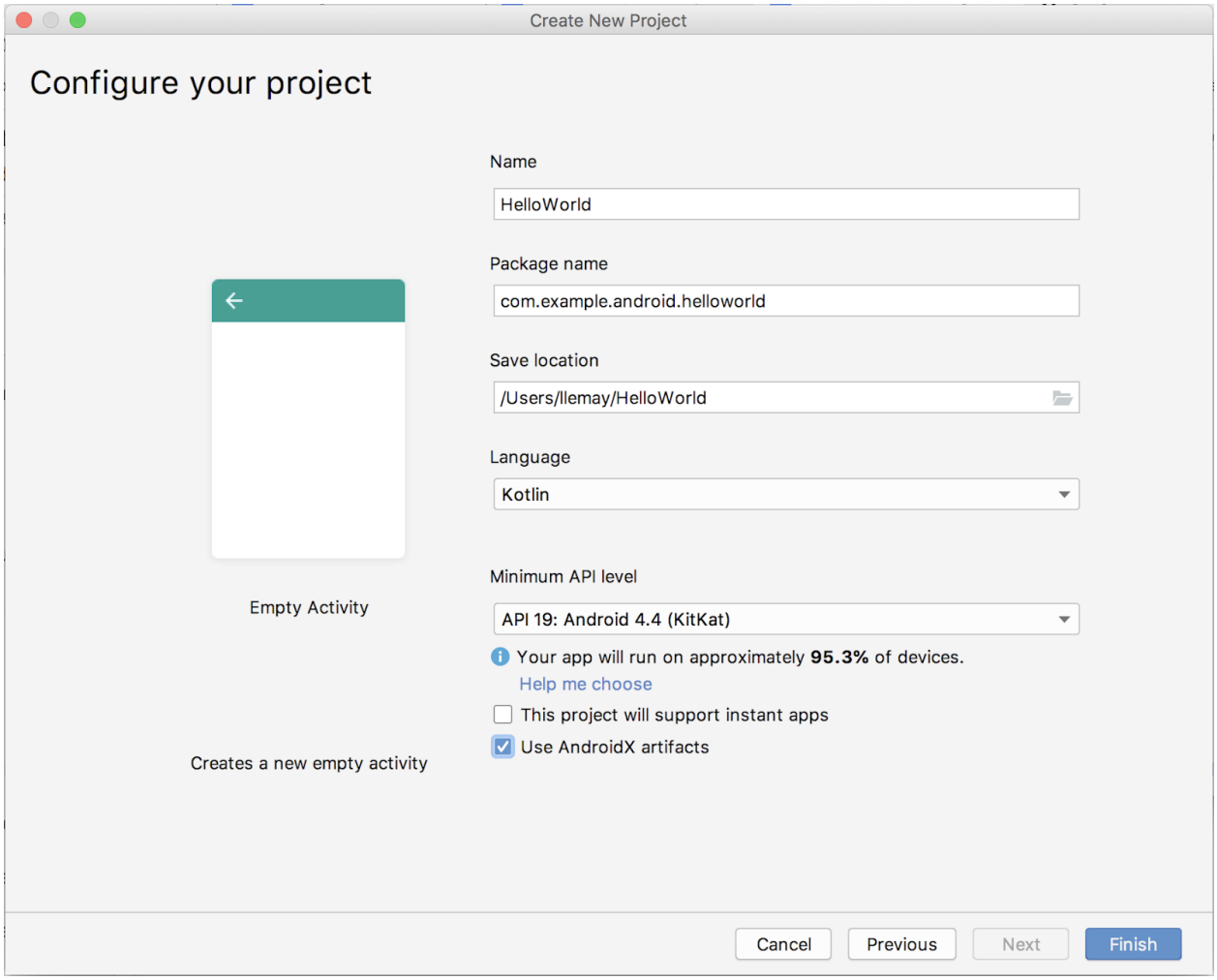


3. The **Choose your project** dialog appears. Select **Empty Activity** as shown below, and click **Next**.



An `Activity` is a single, focused thing that the user can do. Every app must have at least one activity as its entry point. Think of this entry-point activity as the `main()` function in other programs. An activity typically has a layout associated with it that defines how user interface (UI) elements appear on a screen. Android Studio provides several `Activity` templates to help you get started.

4. In the **Configure your project** dialog, enter "HelloWorld" for the **Name**.



5. Accept the default **android.example.com** for **Company domain**, or create a unique company domain. This value plus the name of the app is the package name for your app. If you are not planning to publish your app, accept the default. You can change the package name of your app later, but it is extra work.
6. Verify that the default **Save location** is where you want to store your app. If not, change the location to your preferred directory.
7. Make sure the **Language** is Kotlin.
8. Make sure the Minimum API level is **API 19: Android 4.4 (KitKat)**. At the time this codelab was written, Android Studio indicated that with this API level, the app would run on approximately 95.3% of devices. (You learn more about minimum API levels in a later codelab. To learn more right now, click **Help me choose**, which opens a window with information about the API levels.)
9. Select the **Use AndroidX artifacts** checkbox.
10. Leave all the other checkboxes cleared, and click **Finish**. If your project requires more components for your chosen target SDK, Android Studio installs them automatically, which might take a while. Follow the prompts and accept the default options.

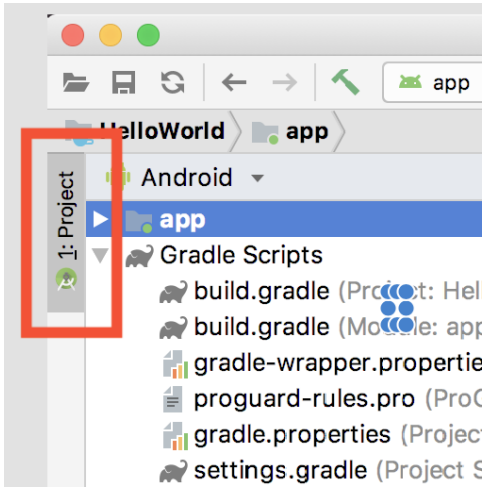
Android Studio now creates your project, which can take some time. You should not get any errors. If you get any warnings, ignore them.

4. Task: Explore Android Studio

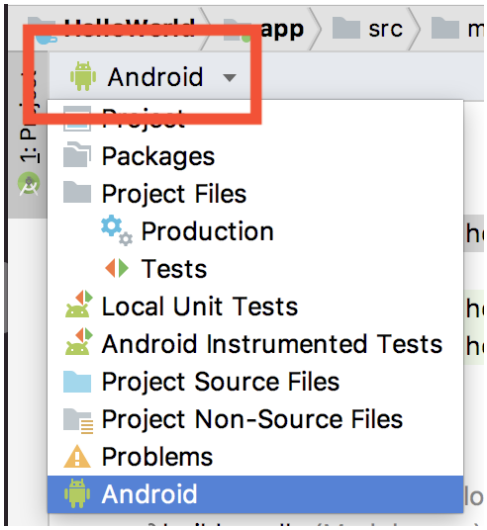
In this task, you explore the HelloWorld project in Android Studio and learn the basics of developing with Android Studio.

Step 1: Explore the Project pane

1. If the **Project** tab is not already selected, select it. The **Project** tab is in the vertical tab column on the left side of the Android Studio window. The Project pane opens.



2. To view the project as a standard Android project hierarchy, select **Android** from the drop-down menu at the top of the Project pane. (**Android** is the default.) You can view the project files in many different ways, including viewing the files how they appear in the filesystem hierarchy. However, the project is easier to work with using the Android view.

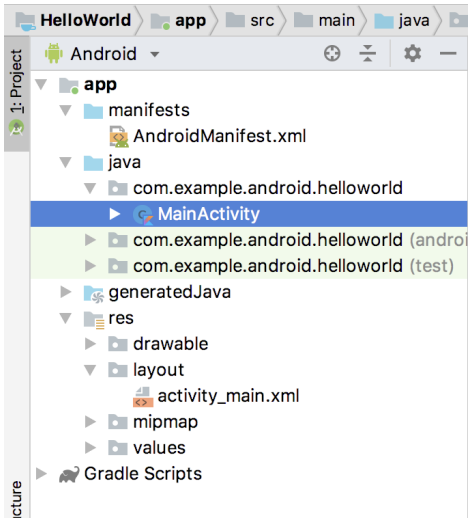


Note: This codelab and others refer to the Project pane, when set to **Android**, as the **Project > Android** pane.

Step 2: Explore the app folder

All code and resources for your app are located within the `app` folder.

1. In the **Project > Android** pane, expand the **app** folder. Inside the `app` folder are four subfolders: `manifests`, `java`, `generatedJava`, and `res`.
2. Expand the **java** folder, and then expand the **com.example.android.HelloWorld** folder to see the **MainActivity** Kotlin file.



The **java** folder contains all the main Kotlin code for an Android app. There are historical reasons why your Kotlin code appears in the java folder. That convention allows Kotlin to interoperate seamlessly with code written in the Java programming language, even in the same project and app.

Your app's class files are contained in three subfolders, as shown in the figure above.
The **com.example.hello.helloworld** (or the domain name you have specified) folder contains all the files for an app package. In particular, the `MainActivity` class is the main entry point for your app. You learn more about `MainActivity` in the next codelab. The other two folders in the `java` folder are used for code related to testing, such as unit tests.

Note: In the file system, your Kotlin files have a `.kt` extension and a **K** icon. In the Project view, Android Studio shows you the class name (`MainActivity`) without the extension.

3. Note the **generatedJava** folder. This folder contains files that Android Studio generates when it builds the app. Don't edit anything in this folder, because your changes might be overridden when you rebuild the app. But it's useful to know about this folder when you need to look at these files during debugging.

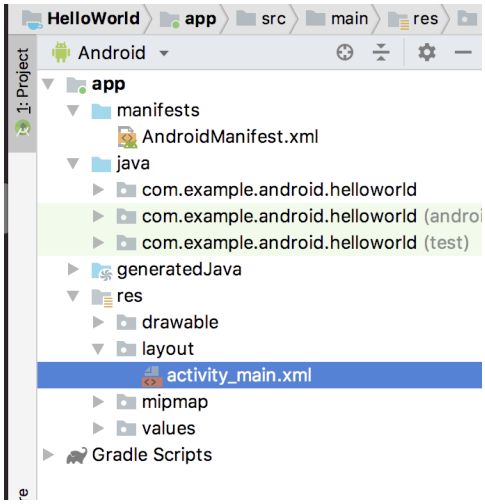
Step 3: Explore the res folder

1. In the **Project > Android** pane, expand the **res** folder.

   The **res** folder holds resources. *Resources* in Android are static content used in your apps. Resources include images, text strings, screen layouts, styles, and values such as hexadecimal colors or standard dimensions.

   Android apps separate Kotlin code and resources as much as possible. That makes it much easier to find all the strings or icons that are used in the app's UI. Also, when you change one of these resource files, the change takes effect everywhere that the file is used in the app.

2. Within the **res** folder, expand the **layout** folder to see the `activity_main.xml` file.

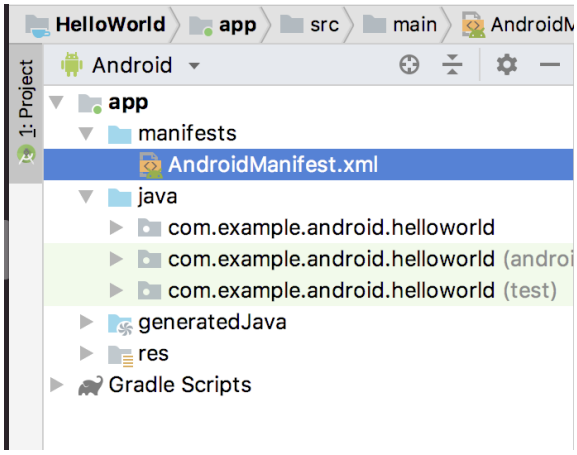


Your `Activity` is usually associated with a UI layout file, defined as an XML file in the `res/layout` directory. That layout file is usually named after its activity. In this case, the activity name is `MainActivity`, so the associated layout is `activity_main`.

Step 4: Explore the manifests folder and AndroidManifest.xml

The `manifests` folder contains files that provide essential information about your app to the Android system.

1. Expand the **manifests** folder and double-click **AndroidManifest.xml** to open it.
   The `AndroidManifest.xml` file includes details that the Android system needs in order to run your app, including what activities are part of the app.



2. Note that `MainActivity` is referenced in the `<activity>` element. Any `Activity` in your app must be declared in the manifest. Here's an example for `MainActivity`:

```
<activity android:name=".MainActivity">
   <intent-filter>
       <action android:name="android.intent.action.MAIN"/>

       <category android:name="android.intent.category.LAUNCHER"/>
   </intent-filter>
</activity>
```

3. Note the `<intent-filter>` element inside `<activity>`. The `<action>` and `<category>` elements in this intent filter tell Android where to start the app when the user clicks the launcher icon. You learn more about intent filters in a later codelab.

The `AndroidManifest.xml` file is also the place where you would define any permissions that your app needed. Permissions include the ability for your app to read phone contacts, send data over the internet, or access

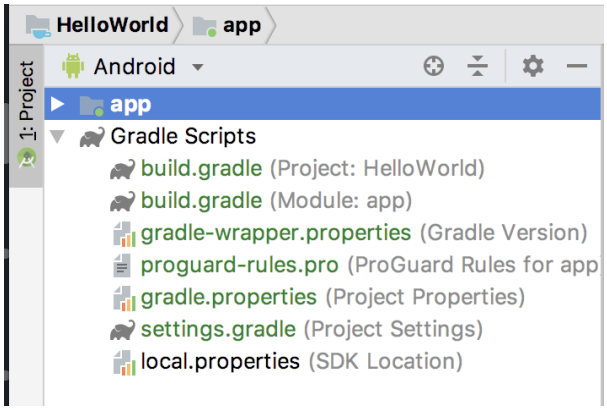
hardware such as the device's camera.

5. Task: Explore the Gradle Scripts folder

Gradle is a build automation system that uses a domain-specific language to describe the app's project structure, configuration, and dependencies. When you compile and run your app, you see information about the Gradle build running. You also see information about the Android Package Kit (APK) being installed. (*APK* is the package file format that the Android operating system uses to distribute and install mobile apps.)

Explore the Gradle system:

1. Expand the **Gradle Scripts** folder. In the **Project > Android** pane, this folder contains all the files that the build system needs.



2. Look for the **build.gradle(Project: HelloWorld)** file.

   This file contains the configuration options that are common to all the modules that make up your project. Every Android Studio project contains a single, top-level Gradle build file. This file defines the Gradle repositories and dependencies that are common to all modules in the project.

3. Look for the **build.gradle(Module:app)** file.

   In addition to the project-level `build.gradle` file, each module has a `build.gradle` file of its own. The module-level `build.gradle` file allows you to configure build settings for each module. (The HelloWorld app has only one module, the module for the app itself.) This `build.gradle` file is the one you most often edit when changing app-level build configurations. For example, you edit this `build.gradle` file when you change the SDK level that your app supports, or when you declare new dependencies in the `dependencies` section. You learn more about both these things in a later codelab.

6. Task: Run your app on a virtual device (emulator)

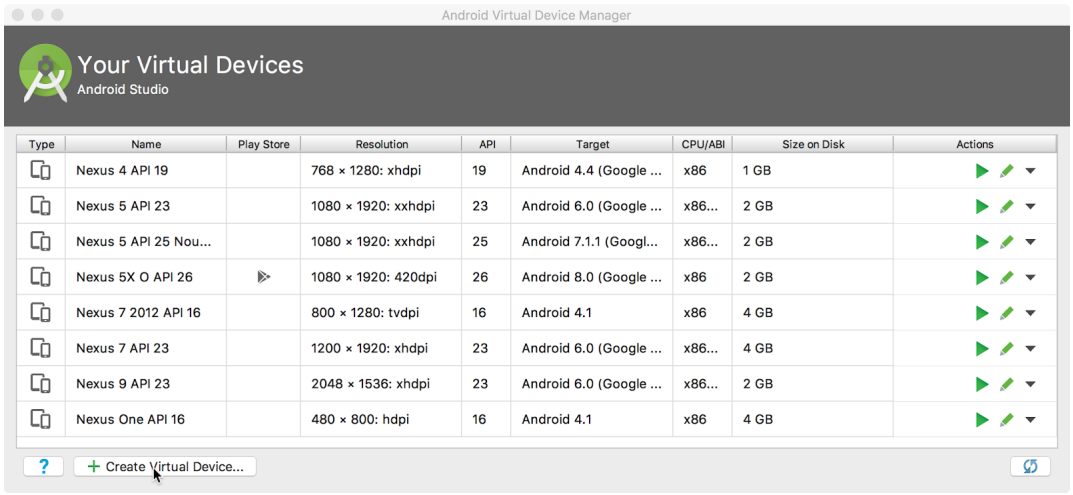
In this task, you use the Android Virtual Device (AVD) manager to create a virtual device (an emulator). The virtual device simulates the configuration for a particular type of Android device. Then you use that virtual device to run the app.

The Android Emulator is an independent application, 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.

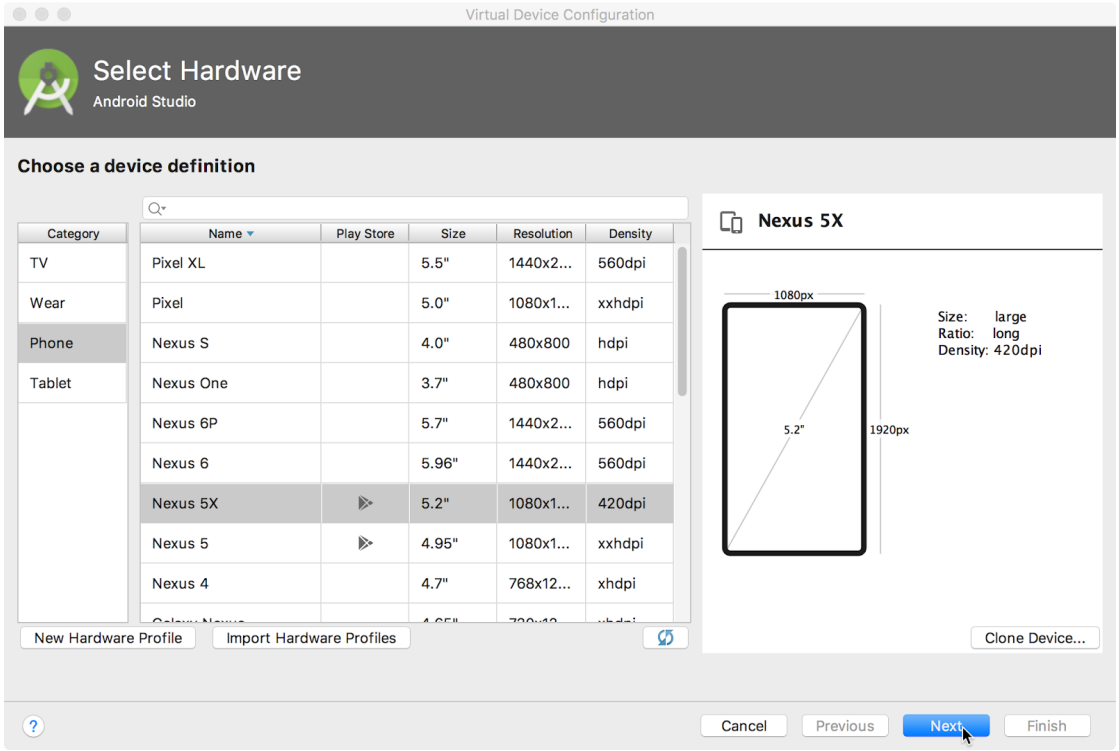
Step 1: Create an Android virtual device (AVD)

To run an emulator on your computer, you have to create a configuration that describes the virtual device.

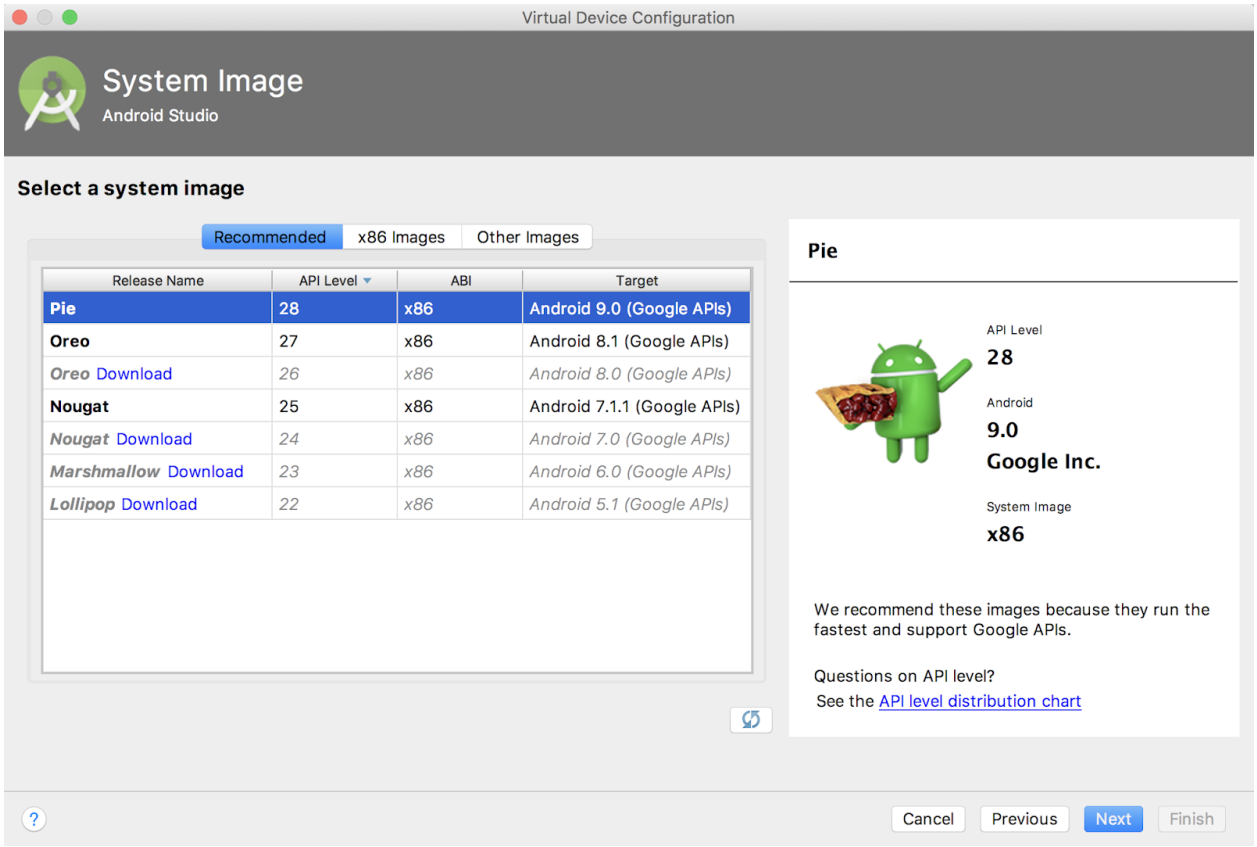
1. In Android Studio, select **Tools > AVD Manager**, or click the **AVD Manager** icon  in the toolbar. The **Your Virtual Devices** dialog appears. If you've already created virtual devices, the dialog shows them (as shown in the figure below), Otherwise, you see a blank list.



2. Click **+Create Virtual Device** at the bottom left of the dialog. The **Select Hardware** dialog appears, showing a list of pre-configured hardware devices. For each device, the table provides a column for its diagonal display size (**Size**), screen resolution in pixels (**Resolution**), and pixel density (**Density**).



3. Select a device such as **Nexus 5x** or **Pixel XL**, and click **Next**. The **System Image** dialog appears.
4. Click the **Recommended** tab, and choose which version of the Android system to run on the virtual device (such as **Pie**).



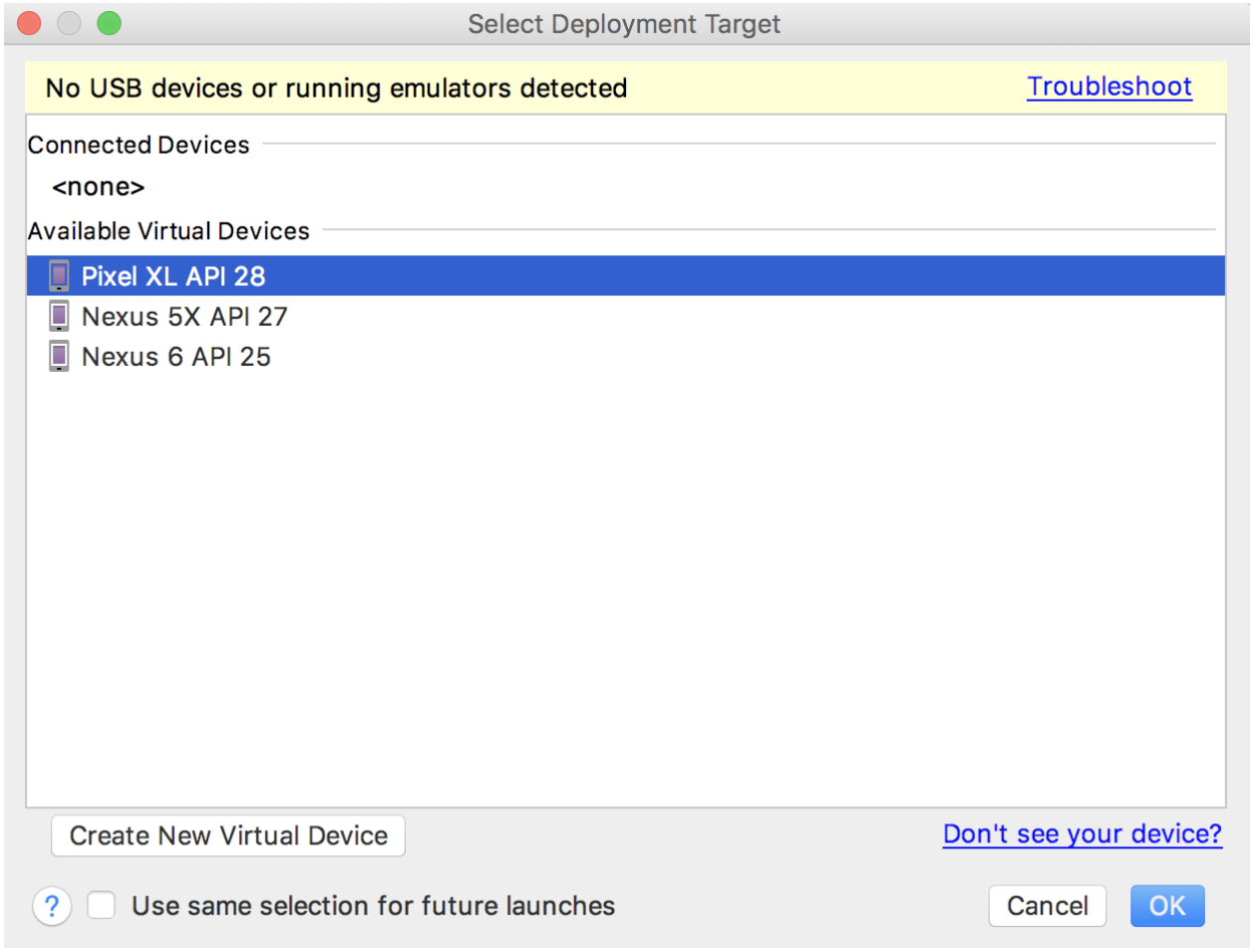
> **Note:** Many more versions of the Android system are available than are shown in the **Recommended** tab. To see them, look at the **x86 Images** and **Other Images** tabs.
>
> These images use a lot of disk space, so only a few are part of your original installation. If a **Download** link is visible next to a system image you want to use, that image is not installed. Click the link to start the download, which can take a long time. When the download is complete, click **Finish**.

5. After you choose a system image, click **Next**. The **Android Virtual Device (AVD)** dialog opens. Check your configuration and click **Finish**.

Step 2: Run the app on the virtual device

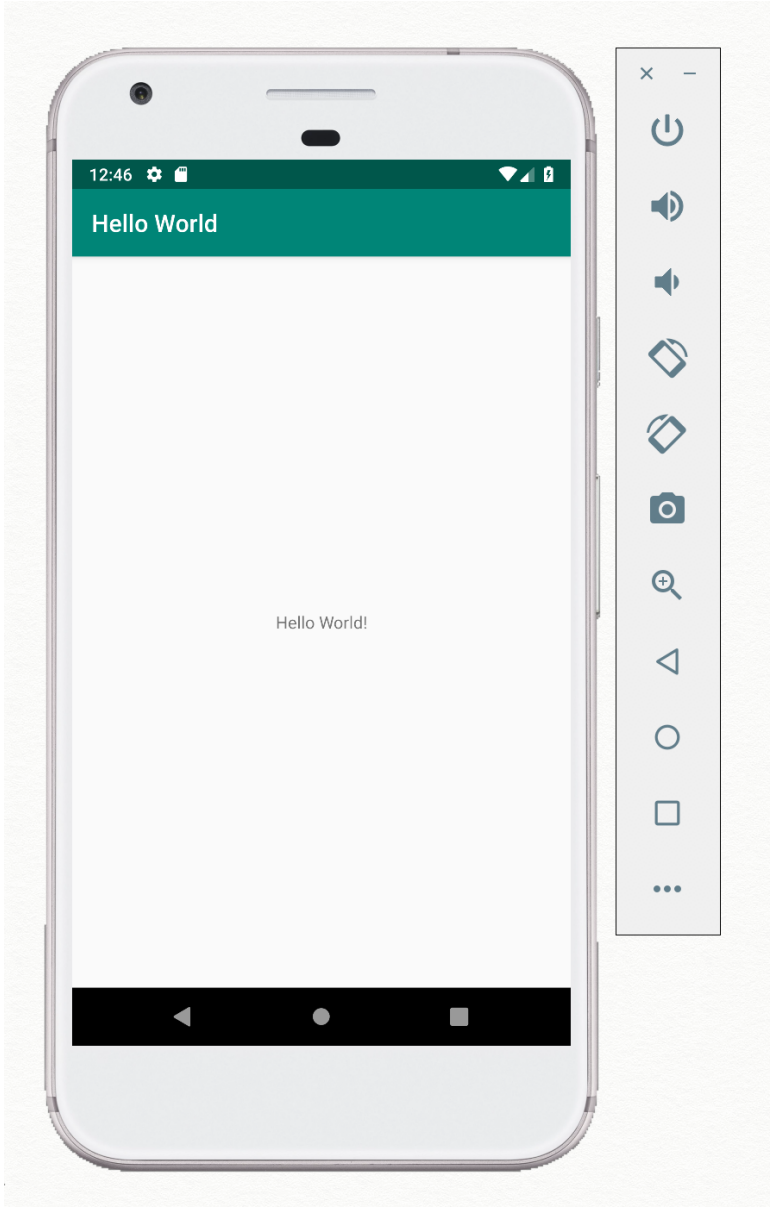
In this task, you finally run your new app.

1. In Android Studio, select **Run > Run app** or click the **Run** icon  in the toolbar. The **Select Deployment Target** dialog appears and warns you that no devices are available. You see this warning if you do not have a physical device connected to your development computer, or if you have not yet launched a virtual device.
2. In the **Select Deployment Target** dialog, under **Available Virtual Devices**, select the virtual device you created. Click **OK**.



The emulator starts and boots just like a physical device. Depending on the speed of your computer, this process may take a while. Your app builds, and when the emulator is ready, Android Studio uploads the app APK to the emulator and runs it.

You should see the HelloWorld app as shown in the following figure.


12:46
Hello World
Hello World!

7. Task: Run your app on a physical device

In this task, you run your app on a physical mobile device such as a phone or tablet, if you have one. Always test your apps on both virtual and physical devices.

What you need:

- An Android device such as a phone or tablet.
- A USB data cable to connect your Android device to your computer via the USB port.
- If you are using a Linux or Windows system, you may need to perform extra steps. See the Run apps on a hardware device documentation. You may also need to install the appropriate USB driver for your device. For Windows-based USB drivers, see Install OEM USB drivers.

Step 1: Turn on USB debugging

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

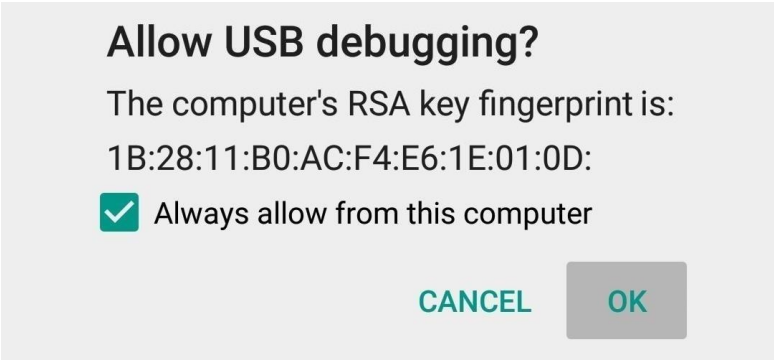
On Android 4.2 (Jellybean) and higher, the **Developer options** settings are hidden by default. To show developer options and enable USB debugging:

1. On your device, open **Settings**, search for **About phone**, tap on **About phone**, and tap **Build number**seven times.
2. Return to the previous page (**Settings / System**). **Developer options** appears in the list. Tap **Developer options**.
3. Select **USB debugging**.

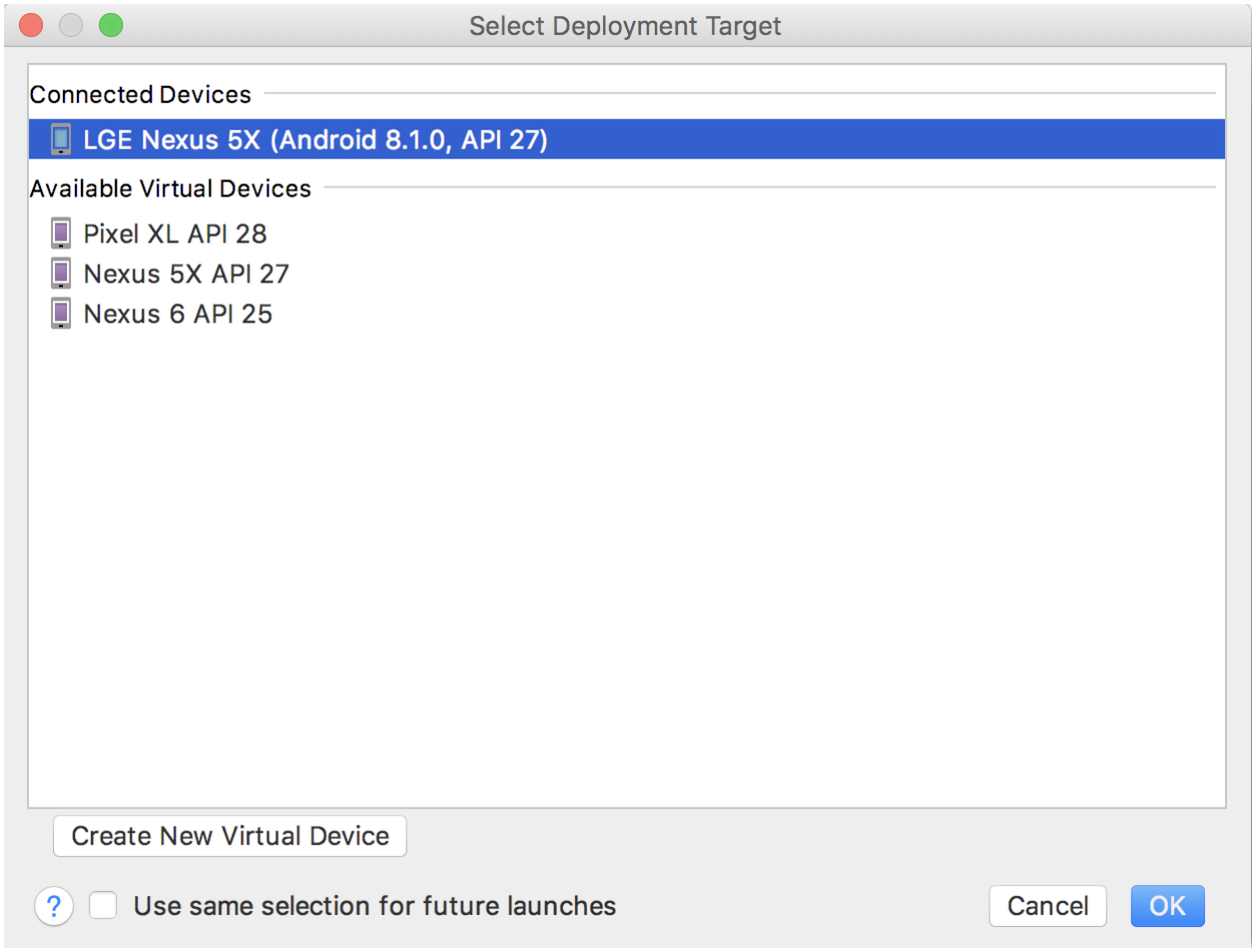
Step 2: Run your app on the Android device

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.



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  . The **Select Deployment Target** dialog opens with the list of available emulators and connected devices. You should see your physical device along with any emulators.



4. Select your device, and click **OK**. Android Studio installs the app on your device and runs it.

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 the **Settings** app.
3. Tap **Revoke USB debugging authorizations**.
4. Reconnect the device to your computer.
5. When prompted, grant authorizations.

You may need to install the appropriate USB driver for your device. See the Run apps on a hardware device.

Android Kotlin Fundamentals 01.2: Anatomy of Basic Android Project

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

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

Introduction

Thus far you've set everything up and Android Studio has created a lot of code for you. Before you modify all that code, it's important to know what you just created and how to navigate the source files of an Android app.

In this codelab, you learn more about the major components of an Android app and add simple interactivity to an app with a button.

What you should already know

- How to install and open Android Studio.
- How to create a new app project.
- How to run an app on an emulator or a physical device.

What you'll learn

How to edit the app's layout file.

How to create an app with interactive behavior.

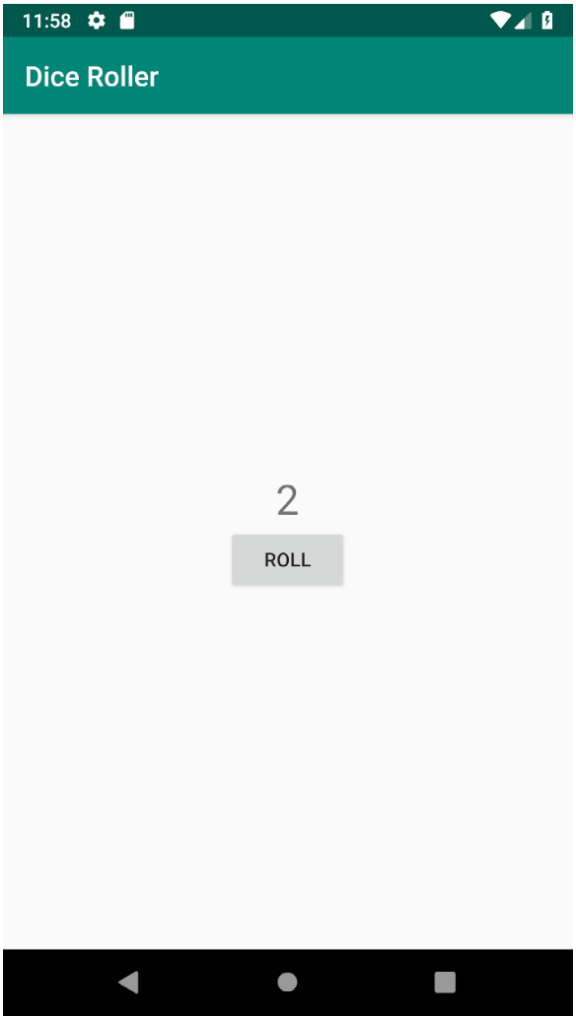
A lot of new terminology. Check out the Vocabulary Glossary for friendly explanations of terms and concepts.

What you'll **do**

- Explore the `MainActivity` Kotlin file and the activity's layout file.
- Edit the activity's layout in XML.
- Add a `Button` element to the activity's layout.
- Extract hardcoded strings into a file of string resources.
- Implement click-handler methods to display messages on the screen when the user taps a `Button`.

2. App overview

In this codelab, you create a new app project called DiceRoller and add basic interactivity with a button. Each time the button is clicked, the value of the displayed text changes. The final DiceRoller app for this codelab looks like this:



3. Task: Explore the activity and layout files

In the last codelab, you learned about the main parts of an app project, including the `java` and `res` directories. In this task, you focus on the two most important files that make up your app: The `MainActivity` Kotlin file, and the `activity_main.xml` layout file.

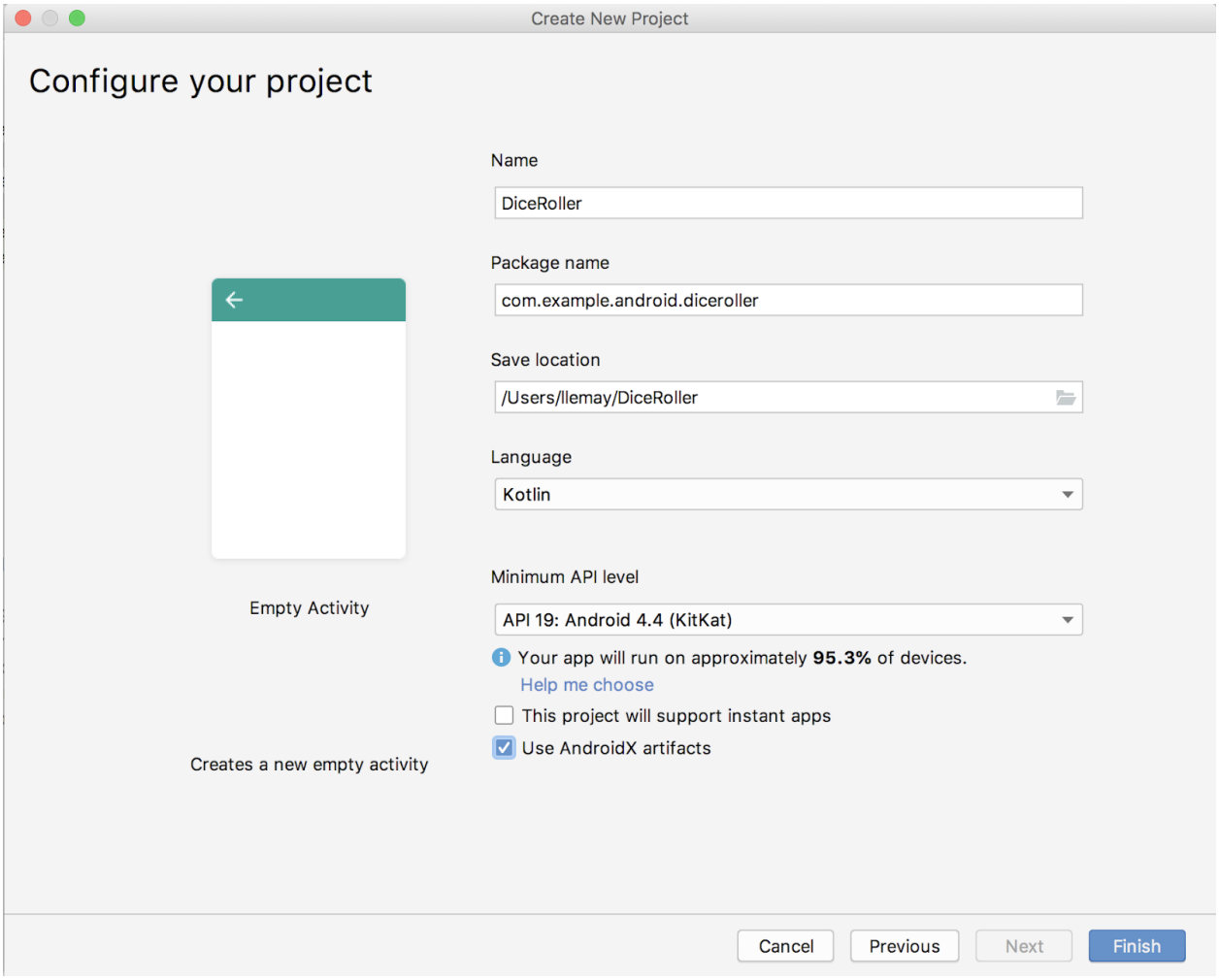
Step 1: Examine MainActivity

`MainActivity` is an example of an `Activity`. An `Activity` is a core Android class that draws an Android app user interface (UI) and receives input events. When your app launches, it launches the activity specified in the `AndroidManifest.xml` file.

Many programming languages define a main method that starts the program. Android apps don't have a main method. Instead, the `AndroidManifest.xml` file indicates that `MainActivity` should be launched when the user taps the app's launcher icon. To launch an activity, the Android OS uses the information in the manifest to set up the environment for the app and construct the `MainActivity`. Then the `MainActivity` does some setup in turn.

Each activity has an associated layout file. The activity and the layout are connected by a process known as *layout inflation*. When the activity starts, the views that are defined in the XML layout files are turned into (or "inflated" into) Kotlin view objects in memory. Once this happens, the activity can draw these objects to the screen and also dynamically modify them.

1. In Android Studio, select **File > New > New Project** to create a new project. Use the Empty activity and click **Next**.
2. Call the project **DiceRoller**, and verify all the other values for project name project location. Make sure "Use AndroidX Artifacts" is checked. Click **Finish**.



3. In the **Project > Android** pane, expand **java > com.example.android.diceroller**. Double-click **MainActivity**. The code editor shows the code in `MainActivity`.

4. Below the package name and import statements is the class declaration for `MainActivity`. The `MainActivity` class extends `AppCompatActivity`.

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

> **`AppCompatActivity`** is a subclass of **`Activity`** that supports all modern Android features while providing backward compatibility with older versions of Android. To make your app available to the largest number of devices and users possible, always use **`AppCompatActivity`**.

5. Notice the `onCreate()` method. Activities do not use a constructor to initialize the object. Instead, a series of predefined methods (called "lifecycle methods") are called as part of the activity setup. One of those lifecycle methods is `onCreate()`, which you always override in your own app. You learn more about the lifecycle methods in a later codelab.

In `onCreate()`, you specify which layout is associated with the activity, and you inflate the layout. The `setContentView()` method does both those things.

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

The `setContentView()` method references the layout using `R.layout.activity_main`, which is actually an integer reference. The `R` class is generated when you build your app. The `R` class includes all the app's assets, including the contents of the `res` directory.

In this case, `R.layout.activity_main` refers to the generated `R` class, the `layout` folder, and the `activity_main.xml` layout file. (Resources do not include file extensions.) You'll refer to many of the app's resources (including images, strings, and elements within the layout file) using similar references in the `R` class.

Step 2: Examine and explore the app layout file

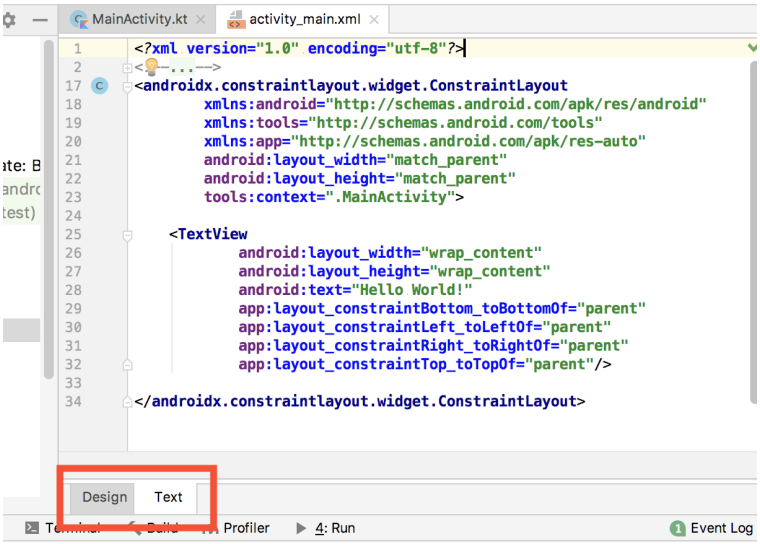
All the activities in your app have an associated layout file in the app's `res/layout` directory. A *layout file* is an XML file that expresses what an activity actually looks like. A layout file does this by defining views and defining where the views appear on the screen.

Views are things like text, images, and buttons that extend the `View` class. There are many types of views, including `TextView`, `Button`, `ImageView`, and `CheckBox`.

In this task, you examine and modify the app layout file.

1. In the **Project > Android** pane, expand **res > layout** and double-click **activity_main.xml**. The layout design editor opens. Android Studio includes this editor, which lets you build your app's layout in a visual way and preview the layout design. You learn more about the design editor in a later codelab.

2. To view the layout file as XML, click the **Text** tab at the bottom of the window.



3. Delete all the existing XML code in the layout editor. The default layout you get with a new project is a good starting point if you're working with the Android Studio design editor. For this lesson you'll work with the underlying XML to build a new layout from scratch.
4. Copy and paste this code into the layout:

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

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>
```

Now examine the code:

1. The top-level or root element of the layout is a `<LinearLayout>` element. The `LinearLayout` view is a `ViewGroup` . View groups are containers that hold other views and help specify the views' positions on the screen.

All the views and view groups you add to your layout are organized in a view *hierarchy*, with the topmost XML element as the root of that hierarchy. The root view can contain other views and view groups, and the contained view groups can contain other views and view groups. When your app runs the view hierarchy in your XML layout file becomes a hierarchy of objects when the layout is inflated. In this case the root view group is a linear layout, which organizes its child views linearly, one after another (either vertically or horizontally).

The default root you get for a new Android project is a `ConstraintLayout` , which works well in coordination with the design editor. For this app, you use a `LinearLayout` view group, which is simpler than the constraint layout. You learn a lot more about view groups and constraint layout in the next lesson. 2. Inside the `LinearLayout` tag, notice the `android:layout_width` attribute. The width of this `LinearLayout` is set to `match parent` , which makes it the same width as its parent. As this is the root view, the layout expands to the full width of the screen. 3. Notice the `android:layout_height` attribute, which is set to `wrap_content` . This attribute makes the height of the `LinearLayout` match the combined height of all the views it contains, which for now is only the `TextView` . 4.

Examine the `<TextView>` element. This `TextView`, which displays text, is the only visual element in your DiceRoller app. The `android:text` attribute holds the actual string to display, in this case the string `"Hello World!"` 5. Notice the `android:layout_width` and `android:layout_height` attributes in the `<TextView>` element, which are both set to `wrap_content`. The content of text view is the text itself, so the view will take up only the space required for the text.

4. Task: Add a button

The dice-rolling app isn't very useful without a way for the user to roll the dice and see what they rolled. To start, add a button to the layout to roll the dice, and add text that shows the dice value that the user rolled.

Step 1: Add a button to the layout

1. Add a `Button` element to the layout below the text view by entering **<Button** and then press Return. A `Button` block appears that ends with `/>` and includes the `layout_width` and `layout_height` attributes.

```
<Button
    android:layout_width=""
    android:layout_height="" />
```

2. Set both the `layout_width` and `layout_height` attributes to `"wrap_content"`. With these values the button is the same width and height as the text label it contains.
3. Add an `android:text` attribute to the button, and give it a value of "Roll". The Button element now looks like this:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Roll" />
```

For `Button` views the `text` attribute is the label of the button. In the layout editor, the attribute is highlighted in yellow, which indicates a tip or a warning. In this case, the yellow highlighting is because the string `"Roll"` is hardcoded in the button label, but the string should be a resource. You learn about string resources in the next section.

Step 2: Extract string resources

Instead of hardcoding strings in your layout or code files, it's a best practice to put all your app strings into a separate file. This file is called `strings.xml`, and it is located among the app's resources, in the `res/values/` directory.

Having the strings in a separate file makes it easier to manage them, especially if you use these strings more than once. Also, string resources are mandatory for translating and localizing your app, because you need to create a string resource file for each language.

Android Studio helps you remember to put your strings into a resource file with hints and warnings.

1. Click once on the "Roll" string in the `android:text` attribute of the `<Button>` tag.
2. Press `Alt+Enter` (`Option+Enter` in macOS) and select **Extract string resource** from the popup menu.
3. Enter `roll_label` for the **Resource name**.
4. Click **OK**. A string resource is created in the `res/values/string.xml` file, and the string in the Button element is replaced with a reference to that resource: `android:text="@string/roll_label"`
5. In the **Project > Android** pane, expand **res > values**, and then double-click **strings.xml** to see your string resources in the `strings.xml` file:

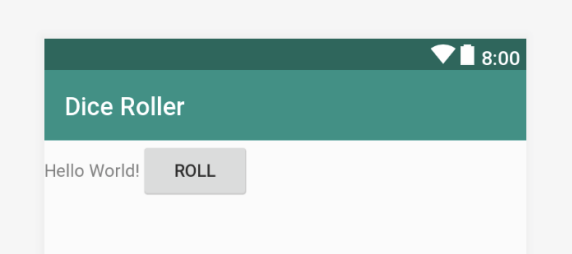
```
<resources>
    <string name="app_name">DiceRoller</string>
    <string name="roll_label">Roll</string>
</resources>
```

Tip: In addition to the string you just added, the `strings.xml` file also includes the app name. The app name appears in the app bar at the top of the screen if you start your app project using the Empty Template. You can change the app name by editing the `app_name` resource.

Step 3: Style and position views

Your layout now contains one `TextView` and one `Button` view. In this task, you arrange the views within the view group to look more attractive.

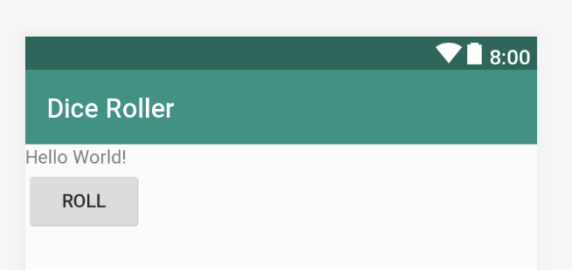
1. Click the **Design** tab to see a preview of the layout. Right now both views are next to each other and pushed up into the top of the screen.



2. Click the **Text** tab to return to the XML editor. Add the `android:orientation` attribute to the `LinearLayout` tag, and give it a value of `"vertical"`. The `<LinearLayout>` element should now look like this:

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

The `LinearLayout` view group positions the views it contains one after another in a line, either horizontally in a row, or vertically in a stack. Horizontal is the default. Because you want the `TextView` stacked on top of the `Button`, you set the orientation to vertical. The design now looks something like this, with the button below the text:



3. Add the `android:layout_gravity` attribute to both the `TextView` and the `Button`, and give it the value `"center_horizontal"`. This aligns both views along the center of the horizontal axis. The TextView and Button elements should now look like this:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Hello World!" />

<Button
```

```
    android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center_horizontal"
     android:text="@string/roll_label" />
```

4. Add the `android:layout_gravity` attribute to the linear layout, and give it the value of `"center_vertical"`. Your `LinearLayout` element should now look like this:

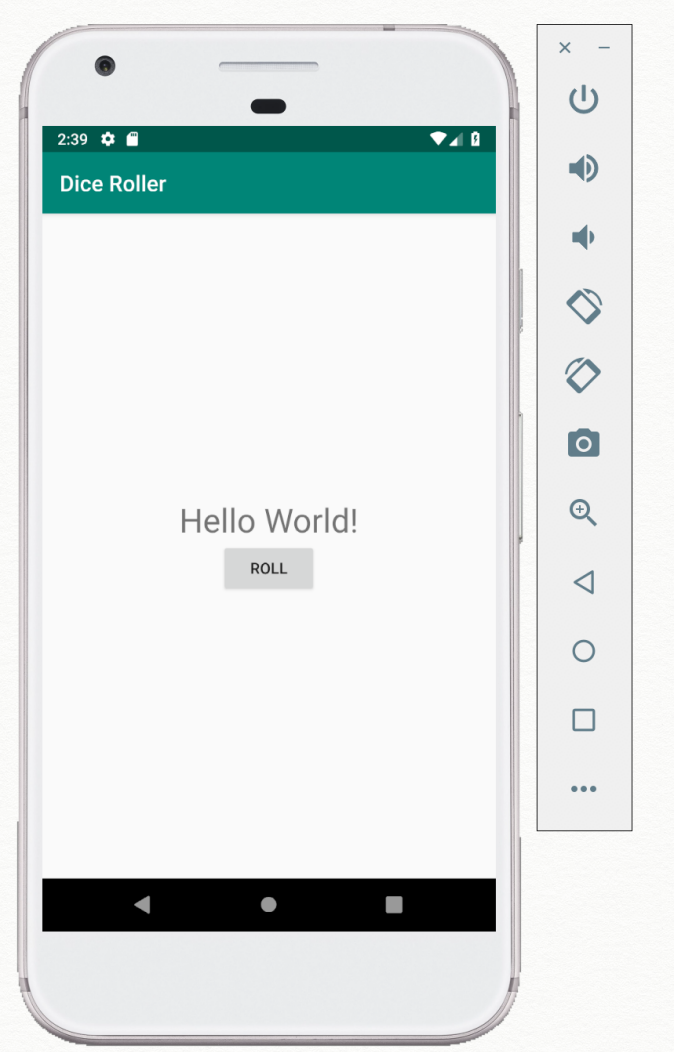
```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:layout_gravity="center_vertical"
    tools:context=".MainActivity">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

> **Note:** If you add the **`center_vertical`** gravity to both the button and text views (instead of **`center_horizontal`**), the views are both centered horizontally and vertically in the middle of the layout. That is, the views are on top of each other.
>
> To center all the child elements at once, use **`center_vertical`** on the parent (the **`LinearLayout`**element), as shown above.

5. To increase the size of the text in the text view, add the `android:textSize` attribute to the `<TextView>` element with the value `"30sp"`. The *sp* abbreviation stands for *scalable pixels*, which is a measure for sizing text independently of the device's display quality. The TextView element should now look like this:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:textSize="30sp"
    android:text="Hello World!" />
```

6. Compile and run your app.



Now both the text and button are nicely placed, and there is larger text in the text view. The button doesn't have any functionality yet, so nothing happens when you click it. You work on that next.

Step 4: Get a reference to the button in code

The Kotlin code in `MainActivity` is responsible for defining the interactive parts of your app, such as what happens when you tap a button. To write a function that executes when the button is clicked, you need to get a reference to the Button object in your inflated layout in MainActivity. To get a reference to the button:

- Assign the `Button` an ID in the XML file.
- Use the `findViewById()` method in your code to get a reference to the `View` with a specific ID.

Once you have a reference to the `Button` view, you can call methods on that view to dynamically change it as the app runs. For example, you can add a click handler that executes code when the button is tapped.

1. Open the `activity_main.xml` layout file, if it is not already open, and click the **Text** tab.
2. Add the `android:id` attribute to the button, and give it a name (in this case, "`@+id/roll_button`"). Your `<Button>` element now looks like this:

```
<Button
    android:id="@+id/roll_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="@string/roll_label" />
```

When you create an ID for a view in the XML layout file, Android Studio creates an integer constant with that ID's name in the generated `R` class. So if you name a view `roll_button`, Android Studio generates and creates an integer constant called `roll_button` in the `R` class. The `"@+id"` prefix for the ID name tells the compiler to add that ID constant to the R class. All the view IDs in your XML file must have this prefix..

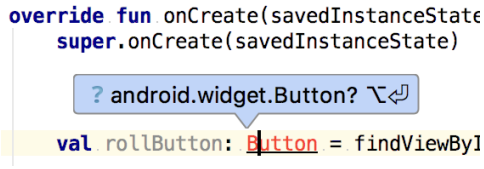
3. Open the `MainActivity` Kotlin file. Inside `onCreate()`, after `setContentView()`, add this line:

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

Use the `findViewById()` method to get a `View` reference for the view that you defined in the XML class. In this case, you get the `Button` reference from the `R` class and the ID `roll_button`, and you assign that reference to the `rollButton` variable.

> **Note:** If you type the line instead of copying and pasting it, you'll notice that Android Studio provides an autocomplete hint for the ID name once you start typing.

4. Notice that Android Studio highlights the `Button` class in red and underlines it, to indicate it is an unresolved reference and that you need to import this class before you can use it. A tooltip indicating the fully qualified class name may also appear:



5. Press `Alt+Enter` (`Option+Enter` on a Mac), to accept the fully qualified class name.

> **Tip**: You can configure Android Studio to automatically add import statements for classes if the meaning is unambiguous. The settings panel for **Editor > General >Auto Import** specifies how imports are handled.

Step 5: Add a click handler to display a toast

A *click handler* is a method that is invoked each time the user clicks or taps on a clickable UI element, such as a button. To create a click handler you need:

- A method that performs some operation.
- The `setOnClickListener()` method, which connects the `Button` to the handler method.

In this task, you create a click-handler method to display a `Toast`. (A *toast* is a message that pops up the screen for a short time.) You connect the click-handler method to the `Button`.

1. In your `MainActivity` class after `onCreate()`, create a private function called `rollDice()`.

```
private fun rollDice() {

}
```

2. Add this line to the `rollDice()` method to display a `Toast` when `rollDice()` is called:

```
Toast.makeText(this, "button clicked",
    Toast.LENGTH_SHORT).show()
```

To create a toast, call the `Toast.makeText()` method. This method requires three things:

- A `Context` object. The `Context` object allows you to communicate with and get information about the current state of the Android OS. You need a `Context` here so that the `Toast` object can tell the OS to display the toast. Because `AppCompatActivity` is a subclass of `Context`, you can just use the keyword `this` for the context.
- The message to be shown, here `"button clicked"`.
- the duration to show the message. The `show()` method at the end displays the toast.

3. In `onCreate()`, after the call to `findViewById()` add this line to assign `rollDice()` as a click handler to the `rollButton` object:

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

The full definition of your `MainActivity` class now looks like this:

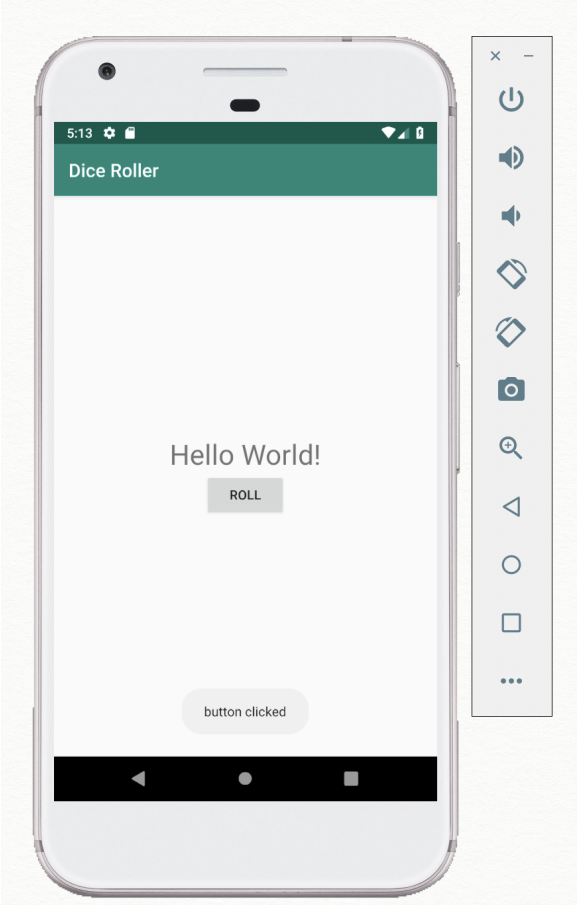
```
class MainActivity : AppCompatActivity() {

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

        val rollButton: Button = findViewById(R.id.roll_button)
        rollButton.setOnClickListener { rollDice() }
    }

    private fun rollDice() {
        Toast.makeText(this, "button clicked",
            Toast.LENGTH_SHORT).show()
    }
}
```

4. Compile and run your app. Each time you tap the button, a toast should appear.


5:13
Dice Roller
Hello World!
ROLL
button clicked

5. Task: Change the text

In this task, you modify the `rollDice()` method to change the text in the `TextView`. For the first step, you change that text from `"Hello World!"` to the string `"Dice Rolled!"`. For the second step, you display a random number between one and six.

Step 1: Display a string

1. Open `activity_main.xml`, and add an ID to the `TextView`.

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

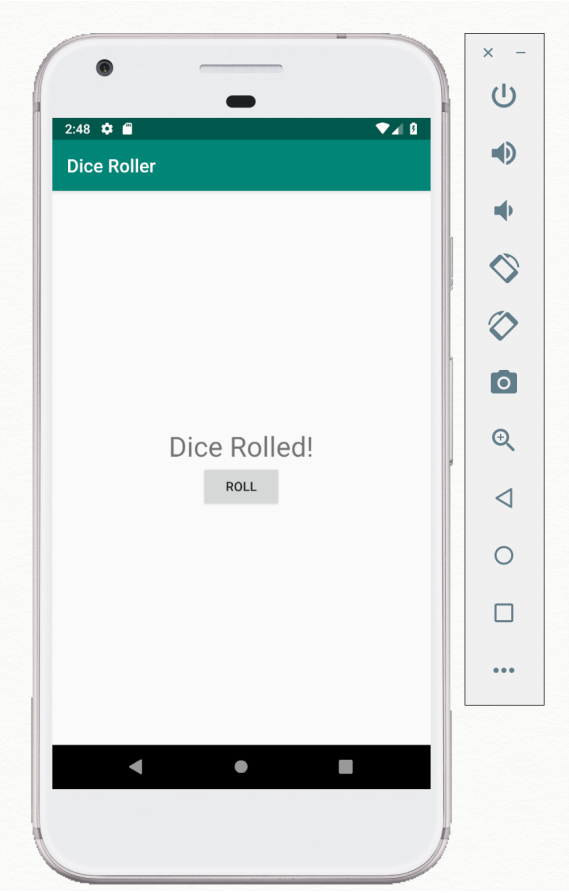
2. Open `MainActivity`. In the `rollDice()` method, comment out the line to display the `Toast`.
3. Use the `findViewById()` method to get a reference to the `TextView` by its ID. Assign the reference to a `resultText` variable.

```
val resultText: TextView = findViewById(R.id.result_text)
```

4. Assign a new string to the `resultText.text` property to change the displayed text. You can ignore the hint to extract that string into a resource; this is just a temporary string.

```
resultText.text = "Dice Rolled!"
```

5. Compile and run the app. Note that tapping the **Roll** button now updates the `TextView`.



Step 2: Display a random number

Finally, in this task you add randomness to the button click, to simulate the roll of the dice. Each time the button is clicked or tapped your code picks a random number from 1 to 6 and updates the `TextView`. The task of generating a random number isn't Android-specific, and you use the `Random` function on the range to do it.

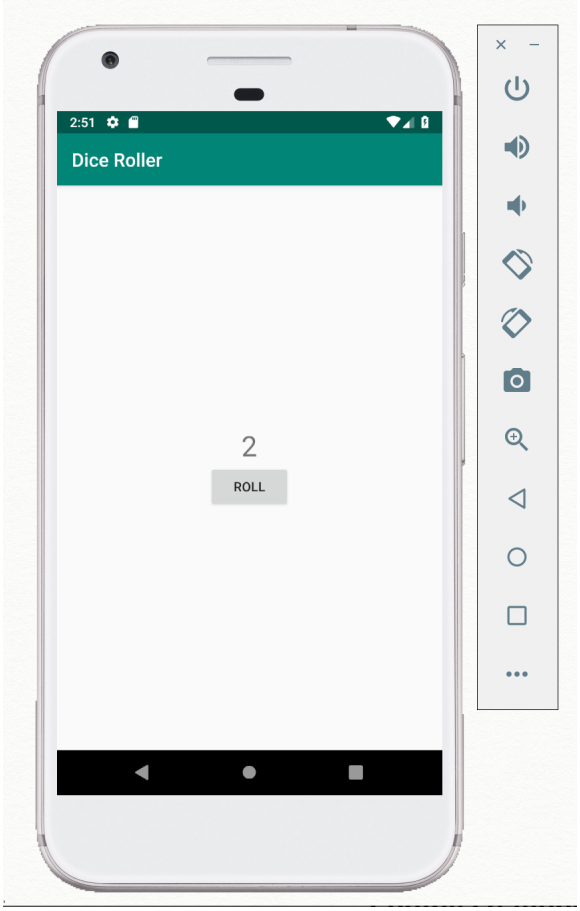
1. At the top of the `rollDice()` method, use the `(1..6).random()` method to get a random number between 1 and 6:

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

2. Set the `text` property to the value of the random integer, as a string:

```
resultText.text = randomInt.toString()
```

3. Compile and run the app. Each time you tap the **Roll** button, the number in the text view changes.



Android Kotlin Fundamentals 01.3: Image resources and compatibility

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

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

Introduction

In this codelab, you improve the DiceRoller app from the last codelab and learn how to add and use image resources in your app. You also learn about app compatibility with different Android versions and how the Android Jetpack can help.

What you should already know

- How to create a new app project, and run an app on an emulator or a physical device.
- The basic components of an app project, including the resource (`res`) directory and Gradle build files.
- How to edit the app's layout file.
- How to find and modify view objects in your app's code.

What you'll learn

How to add files to your app's resources.

How to use images in your app's layout.

How to find views more efficiently in your app's code.

How to use placeholder images in your app's design with XML namespaces.

About Android API levels for your app, and how to understand the minimum, targeted, and compiled API levels.

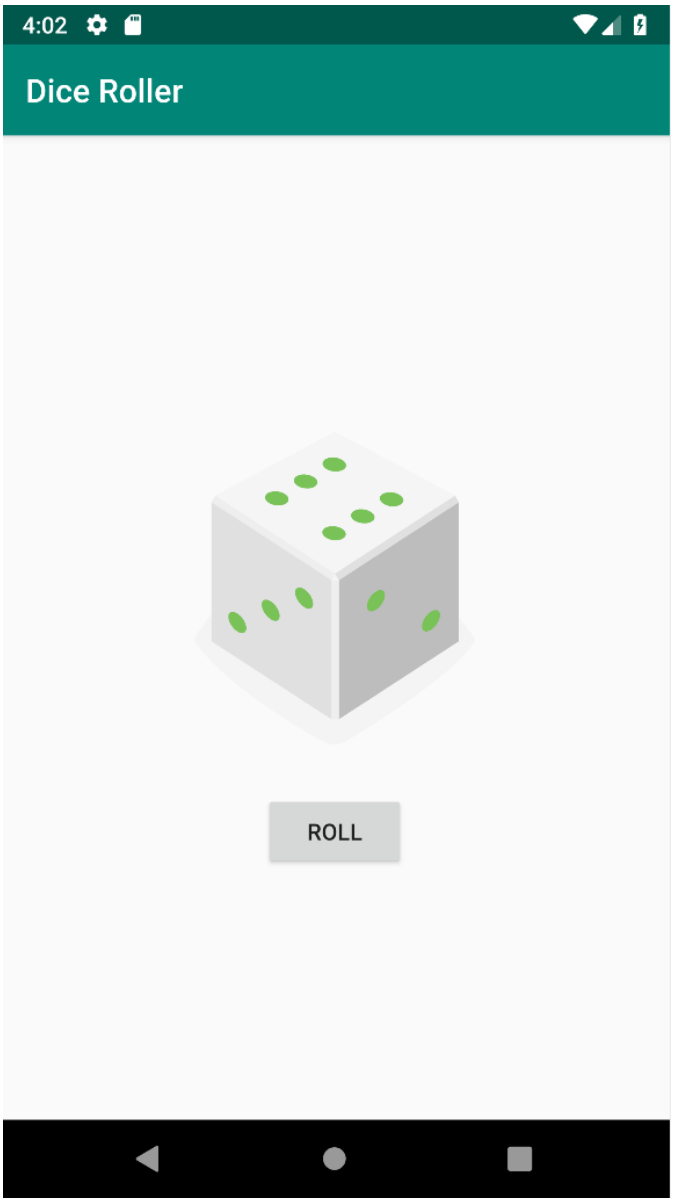
How to use the Jetpack libraries in your app to support older versions of Android.

What you'll **do**

- Modify the DiceRoller app from the last codelab to include images for the dice value, rather than a number.
- Add image files to your app's resources.
- Update the app's layout and code to use images for the dice value, rather than a number.
- Update your code to find views more efficiently.
- Update your code to use an empty image when the app starts.
- Update your app to use the Android Jetpack libraries for backward-compatibility with older versions of Android.

2. App overview

In this codelab, you build on the DiceRoller app you started in the previous codelab, and you add dice images that change when the dice is rolled. The final DiceRoller app looks like this:



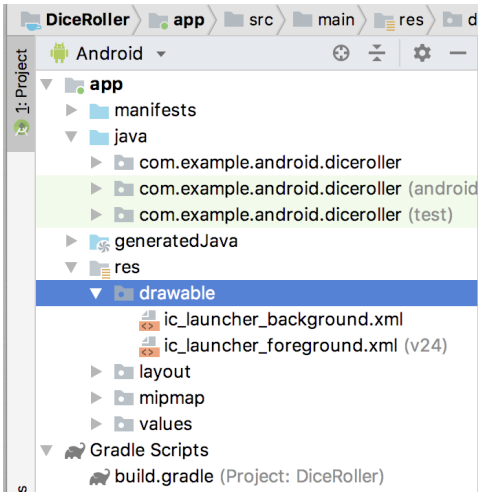
If you did not work through the last codelab, you can download the starting app here: DiceRoller.

3. Task: Add and update image resources

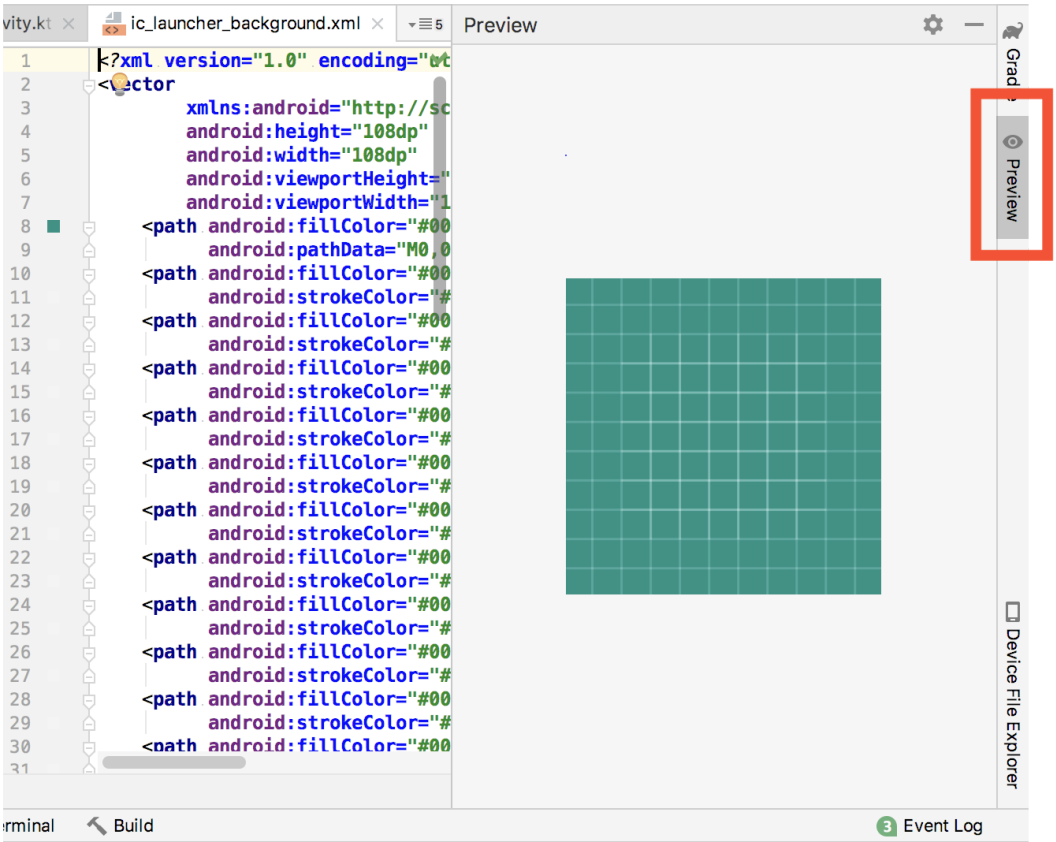
At the end of the last codelab, you had an app that updates a text view with a number between 1 and 6 each time the user taps a button. However, the app is called DiceRoller, not 1-6 Number Generator, so it would be nice if the dice actually looked like dice. In this task, you add some dice images to your app. Then instead of updating text when the button is pressed, you swap in a different image for each roll result.

Step 1: **Add the images**

1. Open the DiceRoller app project in Android Studio if it is not already open. If you did not work through the last codelab, you can download the app here: DiceRoller.
2. In the Project > Android view, expand the **res** folder and then expand **drawable**.



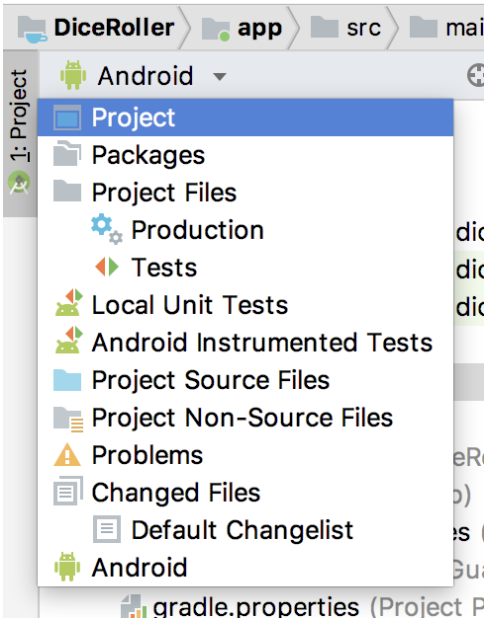
Your app uses many different resources including images and icons, colors, strings, and XML layouts. All those resources are stored in the `res` folder. The `drawable` folder is where you should put all the image resources for your app. Already in the `drawable` folder you can find the resources for the app's launcher icons. 3. Double-click **ic_launcher_background.xml**. Note that these are XML files that describe the icon as a vector image. Vectors enable your images to be drawn at many different sizes and resolutions. Bitmap images such as PNG or GIF may need to be scaled for different devices, which can result in some loss of quality. 4. Click **Preview** in the right column of the XML editor to view the vector drawable in visual form.



5. Download the dice images for your app from DiceImages.zip. Unzip the archive. You should have a folder of XML files that looks like this:

Name	Date Modified	Size	Kind
dice_1.xml	Aug 30, 2018 at 6:21 PM	3 KB	XML Document
dice_2.xml	Aug 30, 2018 at 6:21 PM	3 KB	XML Document
dice_3.xml	Aug 30, 2018 at 6:21 PM	3 KB	XML Document
dice_4.xml	Aug 30, 2018 at 6:21 PM	3 KB	XML Document
dice_5.xml	Aug 30, 2018 at 6:21 PM	4 KB	XML Document
dice_6.xml	Aug 30, 2018 at 6:21 PM	4 KB	XML Document
empty_dice.xml	Aug 30, 2018 at 6:21 PM	921 bytes	XML Document

6. In Android Studio, click the drop-down menu at the top of the project view that currently says **Android**, and choose **Project**. The screenshot below shows what the structure of your app looks like in the file system.

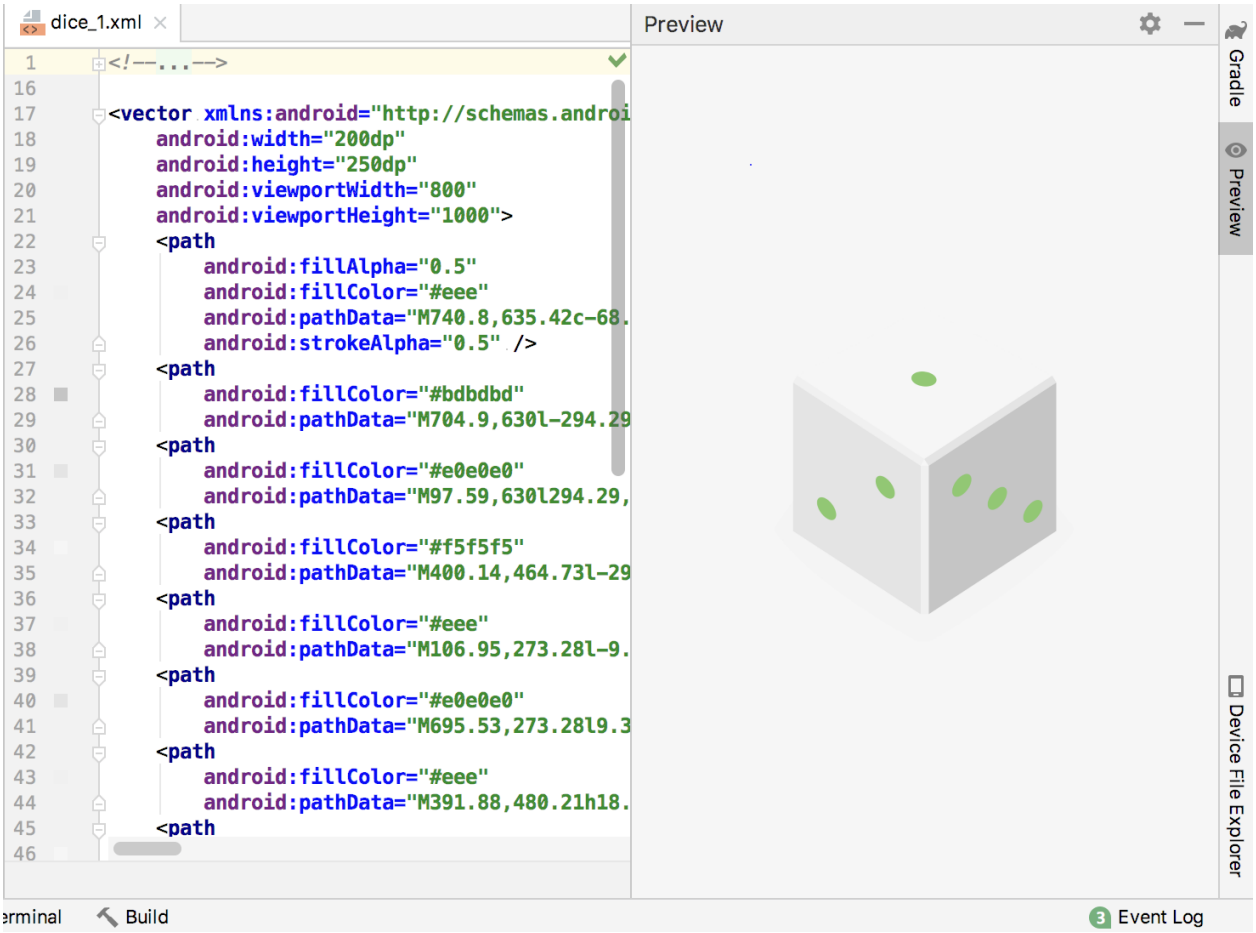


7. Expand **DiceRoller > app > src > main > res > drawable**.
8. Drag all the individual XML files from the `DiceImages` folder into Android Studio and onto the **drawable** folder. Click **OK**.

> **Note:** Make sure you drop the files onto the **drawable** folder and not the **drawable24** folder. You learn more about this folder and what the others are used for later.

Also, do not include the `DiceImages` folder itself. Drag only the XML files.

9. Switch the project back to **Android** view, and notice that your dice image XML files are in the drawable folder.
10. Double-click `dice_1.xml`, and notice the XML code for this image. Click the **Preview** button to get a preview of what this vector drawable actually looks like.



Step 2: **Update the layout to use images**

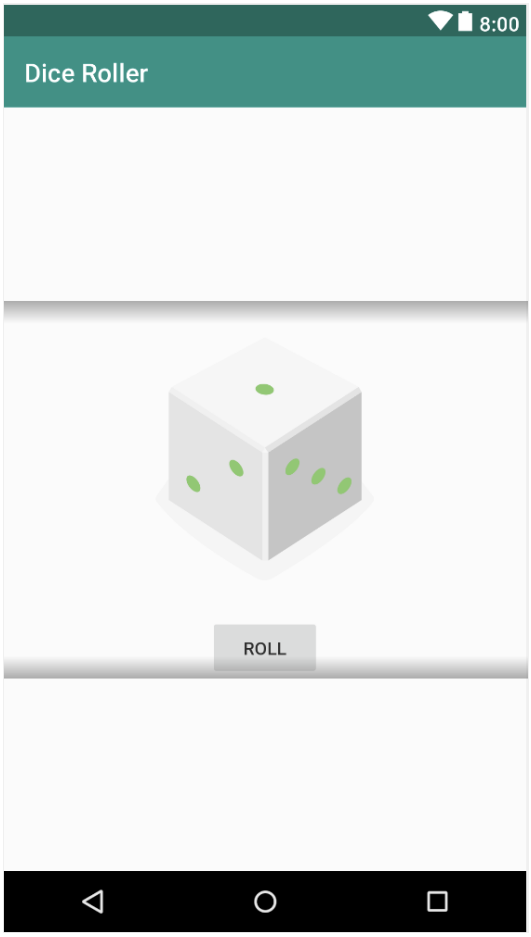
Now that you have the dice image files in your `res/drawables` folder, you can access those files from your app's layout and code. In this step, you replace the `TextView` that displays the numbers with an `ImageView` to display the images.

1. Open the `activity_main.xml` layout file if it is not already open. Click the **Text** tab to view the layout's XML code.
2. Delete the `<TextView>` element.
3. Add an `<ImageView>` element with these attributes:

```
<ImageView
    android:id="@+id/dice_image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:src="@drawable/dice_1" />
```

You use an `ImageView` to display an image in your layout. The only new attribute for this element is `android:src`, to indicate the source resource for the image. In this case, an image source of `@drawable/dice_1` means Android should look in the drawable resources (`res/drawable`) for the image named `dice_1`.

4. Click the **Preview** button to preview the layout. It should look like this:



Step 3: **Update the code**

1. Open `MainActivity`. Here's what the `rollDice()` function looks like so far:

```
private fun rollDice() {
    val randomInt = (1..6).random()

    val resultText: TextView = findViewById(R.id.result_text)
    resultText.text = randomInt.toString()
}
```

Notice that the reference to `R.id.result_text` may be highlighted in red—that's because you deleted the `TextView` from the layout, and that ID no longer exists.

2. Delete the two lines at the end of the function that define the `resultText` variable and set its text property. You're no longer using a `TextView` in the layout, so you don't need either line.
3. Use `findViewByID()` to get a reference to the new `ImageView` in the layout by ID (`R.id.dice_image`), and assign that view to a new `diceImage` variable:

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

4. Add a `when` block to choose a specific dice image based on the value of `randomInteger`:

```
val drawableResource = when (randomInt) {
    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
}
```

As with the IDs you can reference the dice images in the drawable folder with the values in the `R` class. Here `R.drawable` refers to the app's drawable folder, and `dice_1` is a specific dice image resource within that folder.

5. Update the source of the `ImageView` with the `setImageResource()` method and the reference to the dice image you just found.

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

6. Compile and run the app. Now when you click the **Roll** button, the image should update with the appropriate image.

4. Task: Find views efficiently

Everything in your app works, but there's more to developing apps than just having code that works. You should also understand how to write performant, well-behaving apps. This means your apps should run well, even if your user doesn't have the most expensive Android device or the best network connectivity. Your apps should also continue to run smoothly as you add more features, and your code should be readable and well organized.

In this task, you learn about one way to make your app more efficient.

1. Open `MainActivity`, if it is not already open. In the `rollDice()` method, note the declaration for the `diceImage` variable:

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

Because `rollDice()` is the click handler for the **Roll** button, every time the user taps that button, your app calls `findViewById()` and gets another reference to this `ImageView`. Ideally, you should minimize the number of calls to `findViewById()`, because the Android system is searching the entire view hierarchy each time, and that's an expensive operation.

In a small app like this one, it's not a huge problem. If you're running a more complicated app on a slower phone, continually calling `findViewById()` could cause your app to lag. Instead it is a best practice to just call `findViewById()` once and store the `View` object in a field. Keeping the reference to the `ImageView` in a field allows the system to access the `View` directly at any time, which improves performance.

2. At the top of the class, before `onCreate()`, create a field to hold the `ImageView`.

```
var diceImage : ImageView? = null
```

Ideally you would initialize this variable up here when it's declared, or in a constructor—but Android activities don't use constructors. In fact, the views in the layout are not accessible objects in memory at all until after they have been inflated in the `onCreate()` method, by the call to `setContentView()`. You can't initialize the `diceImage` variable at all until that happens.

One option is to define the `diceImage` variable as nullable, as in this example. Set it to `null` when it's declared, and then assign it to the real `ImageView` in `onCreate()` with `findViewById()`. This will complicate your code, however, because now you have to check for the `null` value every time you want to use `diceImage`. There's a better way.

3. Change the `diceImage` declaration to use the `lateinit` keyword, and remove the `null` assignment:

```
lateinit var diceImage : ImageView
```

The `lateinit` keyword promises the Kotlin compiler that the variable will be initialized before the code calls any operations on it. Therefore we don't need to initialize the variable to `null` here, and we can treat it as a non-nullable variable when we use it. It is a best practice to use `lateinit` with fields that hold views in just this way.

4. In `onCreate()`, after the `setContentView()` method, use `findViewById()` to get the `ImageView`.

```
diceImage = findViewById(R.id.dice_image)
```

5. Delete the old line in `rollDice()` that declares and gets the `ImageView`. You replaced this line with the field declaration earlier.

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

6. Run the app again to see that it still works as expected.

5. Task: Use a default image

Right now you are using `dice_1` as the initial image for the dice. Instead, say, you wanted to display no image at all until the dice is rolled for the first time. There are a few ways to accomplish this.

1. Open `activity_main.xml` in the **Text** tab.
2. In the `<ImageView>` element, set the `android:src` attribute to `"@drawable/empty_dice"` :

```
android:src="@drawable/empty_dice"
```

The `empty_dice` image was one of the images you downloaded and added to the `drawable` folder. It's the same size as the other dice images, only it's empty. This image is the one that will be shown when the app first starts.

3. Click the **Design** tab. The dice image is empty now, but it's also not visible in the preview either.

It's fairly common that the contents of a design might be defined dynamically at runtime—for example, any app that grabs data from the internet should probably start with a blank or empty screen. But it's helpful when you're designing an app to have some sort of placeholder data in the layout so you know what you're laying out. 4. In `activity_main.xml` , copy the `android:src` line, and paste a second copy. Change the word "android" to "tools", so your two attributes look like this:

```
android:src="@drawable/empty_dice"
tools:src="@drawable/empty_dice" />
```

Here you've changed the XML namespace of this attribute from the default `android` namespace to the `tools` namespace. The `tools` namespace is used when you want to define placeholder content that is only

used in the preview or the design editor in Android Studio. Attributes using the `tools` namespace are removed when you compile the app.

Namespaces are used to help resolve ambiguity when referring to attributes that have the same name. For example, both these attributes in the `<ImageView>` tag have the same name (`src`), but the namespace is different.

5. Examine the `<LinearLayout>` element at the root of the layout file, and notice the two namespaces defined here.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    ...
```

6. Change the `tools:src` attribute in the `ImageView` tag to be `dice_1` instead of `empty_dice` :

```
android:src="@drawable/empty_dice"
tools:src="@drawable/dice_1" />
```

Notice that the `dice_1` image is in place now as the placeholder image in the preview.

7. Compile and run the app. Notice that the dice image is empty in the actual app until you click or tap **Roll**.

6. Task: Understand API levels and compatibility

One of the great things about developing for Android is the sheer number of devices your code can run on—from the Nexus One to the Pixel, to form factors like tablets, to Pixelbooks, to watches, TVs, and cars.

When you write for Android, you don't write completely separate apps for each of these different devices—even apps that run on radically different form factors such as watches and TVs can share code. But there are still constraints and compatibility strategies that you need to be aware of to support all of this.

In this task, you learn how to target your app for specific Android API levels (versions), and how to use the Android Jetpack libraries to support older devices.

Step 1: **Explore API levels**

In the previous codelab, when you created your project, you indicated the specific Android API level that your app should support. The Android OS has different version numbers named after tasty treats which are in alphabetical order. Each OS version ships with new features and functionality. For example, Android Oreo shipped with support for Picture-in-picture apps, while Android Pie introduced Slices. The API levels correspond to the Android versions. For example, API 19 corresponds to Android 4.4 (KitKat).

Due to a number of factors, including what the hardware can support, whether users choose to update their devices, and whether manufacturers support different OS levels, users inevitably end up with devices that run different OS versions.

When you create your app project, you specify the minimum API level that your app supports. That is, you specify the oldest Android version your app supports. Your app also has a level to which it is compiled, and a level that it targets. Each of these levels is a configuration parameter in your Gradle build files.

1. Expand the **Gradle Scripts** folder, and open the **build.gradle (Module: app)** file.

This file defines build parameters and dependencies specific to the app module. The **build.gradle (Project: DiceRoller)** file defines build parameters for the project as a whole. In many cases, your app module is the only module in your project, so this division may seem arbitrary. But if your app becomes more complex and you split it into several parts, or if your app supports platforms like Android watch, you may encounter different modules in the same project. 2. Examine the `android` section towards the top of the `build.gradle` file. (The sample below is not the entire section, but it contains what you're most interested in for this codelab.)

```
android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.example.android.diceroller"
        minSdkVersion 19
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
    }
```

3. Examine the `compileSdkVersion` parameter.

```
compileSdkVersion 28
```

This parameter specifies the Android API level that Gradle should use to compile your app. This is the newest version of Android your app can support. That is, your app can use the API features included in this API level and lower. In this case your app supports API 28, which corresponds to Android 9 (Pie).

4. Examine the `targetSdkVersion` parameter, which is inside the `defaultConfig` section:

```
targetSdkVersion 28
```

This value is the most recent API that you have tested your app against. In many cases this is the same value as `compileSdkVersion`.

5. Examine the `minSdkVersion` parameter.

```
minSdkVersion 19
```

This parameter is the most important of the three, as it determines the oldest version of Android on which your app will run. Devices that run the Android OS older than this API level cannot run your app at all.

Choosing the minimum API level for your app can be challenging. Set the API level too low, and you miss out on newer features of the Android OS. Set it too high, and your app may only run on newer devices.

When you set up your project and you come to the place where you define the minimum API level for your app, click **Help me choose** to see the **API Version Distribution** dialog. The dialog gives information about how many devices use different OS levels, and features that were added or changed in the OS levels. You can also check out the Android documentation release notes and dashboard, which have further information about the implications of supporting different API levels.

Step 2: **Explore compatibility**

Writing for different Android API levels is a common challenge that app developers face, so the Android framework team has done a lot of work to help you out.

In 2011, the team released the first support library, a Google-developed library that offers backward-compatible classes and helpful functions. In 2018, Google announced Android Jetpack, which is a collection of libraries that includes many of the previous classes and functions of the support library, while also expanding on the support library.

1. Open `MainActivity`.
2. Notice that your `MainActivity` class extends not from `Activity` itself, but from `AppCompatActivity`.

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

`AppCompatActivity` is a compatibility class that ensures your activity looks the same across different platforms OS levels.

3. Click the **+** symbol next to the line that starts with `import` to expand the imports for your class. Note that the `AppCompatActivity` class is imported from the `androidx.appcompat.app` package. The namespace for the Android Jetpack libraries is `androidx`.
4. Open **build.gradle (Module: app)** and scroll down to the dependencies section.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.0-beta01'
    implementation 'androidx.core:core-ktx:1.0.1'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.0-alpha4'
    androidTestImplementation
         'androidx.test.espresso:espresso-core:3.1.0-alpha4'
}
```

Notice the dependency on the `appcompat` library, which is part of `androidx` and contains the `AppCompatActivity` class.

> **Tip:** In general, if your app *can* use a compatibility class from the Jetpack libraries, it *should* use one of those classes, because those classes provide support for the largest possible number of features and devices.

Step 3: **Add compatibility for vector drawables**

You're going to use your new knowledge about namespaces, Gradle, and compatibility to make one final adjustment to your app, which will optimize your app size on older platforms.

1. Expand the **res** folder and then expand **drawable**. Double click one of the dice images.

As you learned earlier, all the dice images are actually XML files that define the colors and shapes for the dice. These kinds of files are called *vector drawables*. The nice thing about vector drawables versus bitmap image formats like PNG is that vector drawables can scale without losing quality. Also, a vector drawable is usually a much smaller file than the same image in a bitmap format.

An important thing to note about vector drawables is that they are supported in API 21 onwards. But your app's minimum SDK is set to API 19. If you tried your app on an API 19 device or emulator, you'd see that the app seems to build and run just fine. So how does this work?

When you build your app, the Gradle build process generates a PNG file from each of the vector files, and those PNG files are used on any Android device below 21. These extra PNG files increase the size of your app. Unnecessarily large apps aren't great—they make downloads slower for users and take up more of their devices' limited space. Large apps also have a higher chance of being uninstalled, and of users failing to download or canceling downloads of those apps.

The good news is that there is an Android X compatibility library for vector drawables all the way back to API level 7. 2. Open **build.gradle (Module: app)**. Add this line to the `defaultConfig` section:

```
vectorDrawables.useSupportLibrary = true
```

3. Click the **Sync Now** button. Every time that a `build.gradle` file is modified, you need to sync the build files with the project.
4. Open the `activity_main.xml` layout file. Add this namespace to the root `<LinearLayout>` tag, underneath the `tools` namespace:

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

The `app` namespace is for attributes that come from either your custom code or from libraries and not the core Android framework.

5. Change the `android:src` attribute in the `<ImageView>` element to be `app:srcCompat`.

```
app:srcCompat="@drawable/empty_dice"
```

The `app:srcCompat` attribute uses the Android X library to support vector drawables in older versions of Android, back to API level 7.

6. Build and run your app. You won't see anything different on the screen, but now your app doesn't need to use generated PNG files for the dice images no matter where the runs, which means a smaller app file.

Android Kotlin Fundamentals 01.4: Learn to help yourself

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

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

Introduction

In this codelab, you learn about resources that are helpful to Kotlin Android developers, including templates, documentation, videos, and sample apps.

What you should already know

- The basic workflow of Android Studio.
- How to use the Layout Editor in Android Studio.

What you'll learn

Where to find Kotlin and Android developer information and resources.

How to change the launcher icon in an app.

How to look for help when you're building Android apps using Kotlin.

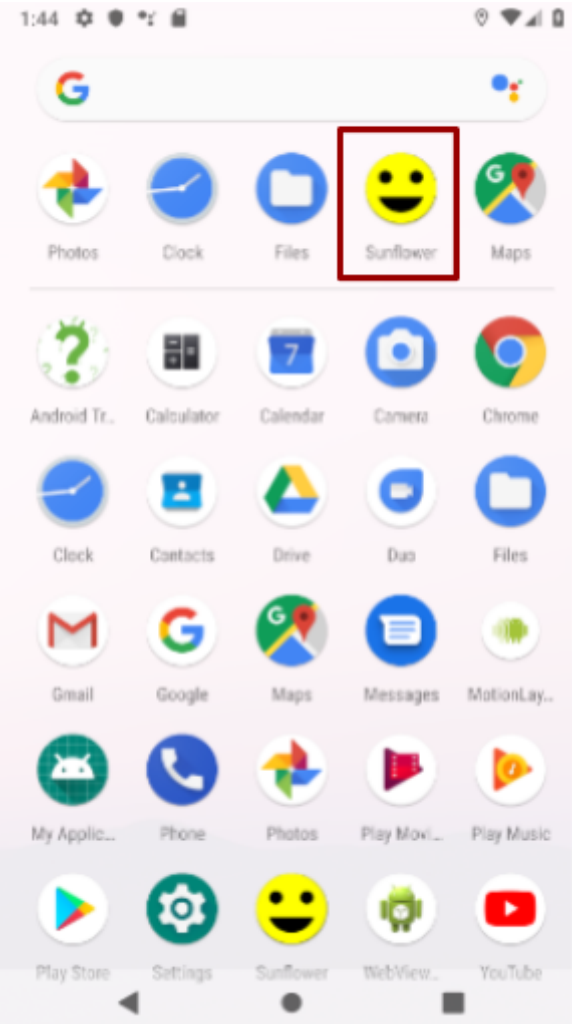
What you'll do

- Explore some of the resources available to Kotlin Android developers of all levels.
- Download and explore a Kotlin Android sample app.
- Change an app's launcher icon.

2. App overview

In this codelab, you learn about templates, samples, documentation, and other resources that are available for Kotlin Android developers.

First you create a simple app from an Android Studio template and modify the app. Then you download and explore the Android Sunflower sample app. You replace the sample app's launcher icon (a sunflower) with a clip-art image asset that's available within Android Studio (a smiley face).



3. Task: Use project templates

Android Studio provides templates for common and recommended app and activity designs. Built-in templates save you time, and they help you follow design best practices.

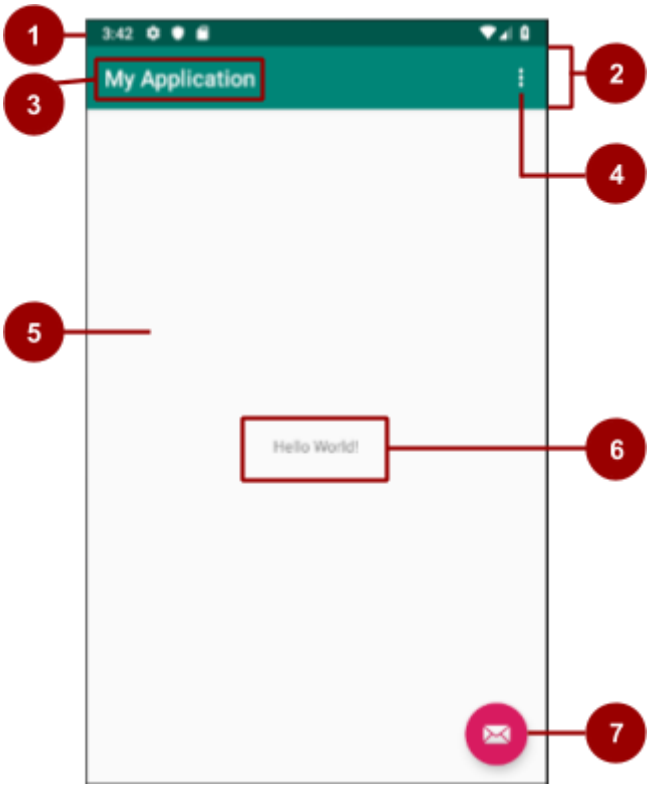
Each template incorporates a skeleton activity and user interface. You've already used the Empty Activity template in this course. The Basic Activity template has more features and incorporates recommended app features, such as the options menu that appears in the app bar on Android-powered devices.

Step **1: Explore the Basic Activity architecture**

1. In Android Studio, create a project.
2. In the **Choose your project** dialog, select the Basic Activity template, and click **Next**.
3. In the **Configure your project** dialog, name the app whatever you'd like. Select **Kotlin** for the language, and select the **Use AndroidX artifacts** checkbox. Click **Finish**.
4. Build the app and run it on an emulator or Android-powered device.
5. Identify the labeled parts in the figure and table below. Find their equivalents on your device or emulator screen. Inspect the corresponding Kotlin code and XML files described in the table.

Being familiar with the Kotlin source code and XML files will help you extend and customize the Basic Activity template for your own needs.

Architecture of the Basic Activity template



#	UI description	Code reference
1	Status bar, which the Android system provides and controls.	Not visible in the template code, b code in `MainActivity.kt` to hide
2	The app bar, also called the *action bar*, provides visual structure, standardized visual elements, and navigation.	In `activity_main.xml`, look for To the `AppBarLayout` in the template the appearance of the app bar, ch
3	The app name is initially derived from your package name, but you can	In `AndroidManifest.xml`, look for

	change it to anything you want.	in `strings.xml` .
4	The options-menu overflow button holds menu items for the activity. The overflow button also holds global menu options such as **Search** and **Settings** for the app. Your app menu items go into this menu.	In `MainActivity.kt` , the `onOptior` menu item.To see the options-me is specified in this template.
5	The CoordinatorLayout `ViewGroup` is a layout that provides mechanisms for UI elements to interact. Your app's UI goes inside the `content_main.xml` file, which is included within this `ViewGroup` .	In `activity_main.xml` , look for th layout includes the `content_main` from the views unique to your app
6	The template uses a TextView to display "Hello World". You replace this `TextView` with your app's UI elements.	The "Hello World" text view is in th file.
7	Floating action button (FAB)	In `activity_main.xml` , look for th clip-art icon. `MainActivity.kt` inc

Step 2: Customize the app that the template produces

Change the appearance of the app produced by the Basic Activity template. For example, you can change the color of the app bar to match the status bar. (On some devices, the status bar is a darker shade of the same primary color that the app bar uses.)

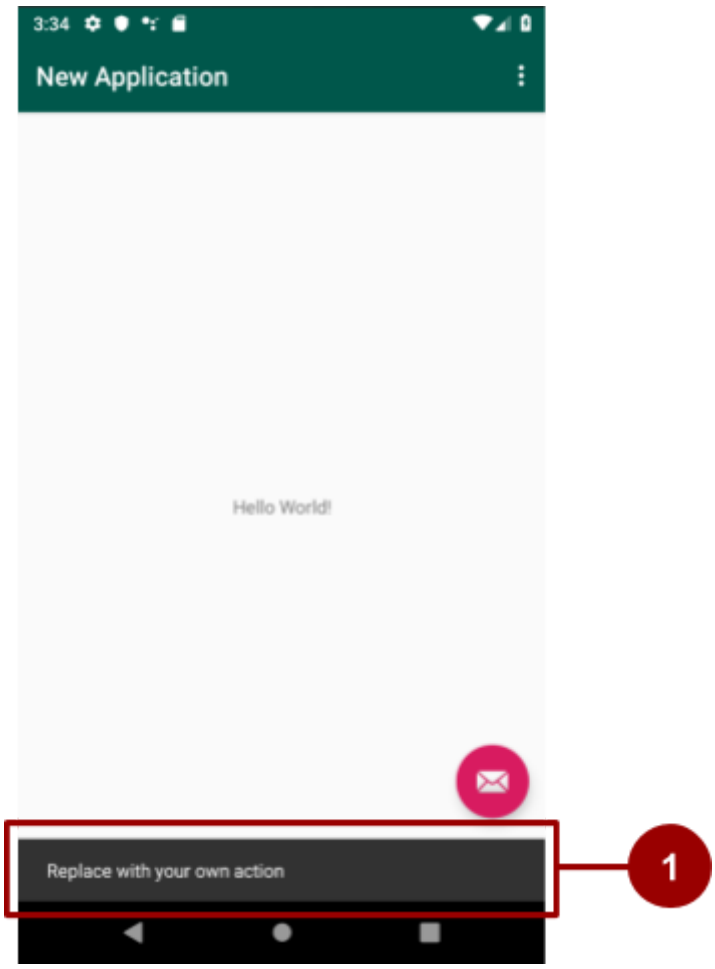
1. Change the name of the app that the app bar displays. To do this, change the `app_name` string resource in the **res > values > strings.xml** file to the following:

```
<string name="app_name">New Application</string>
```

2. Change the color of the app bar (`Toolbar`) in the **res > layout > activity_main.xml** by changing the `android:background` attribute to `"?attr/colorPrimaryDark"` . This value sets the app bar color to a darker primary color that matches the status bar:

```
android:background="?attr/colorPrimaryDark"
```

3. Run the app. The app's new name appears in the status bar, and the background color of the app bar is darker and matches the color of the status bar. When you click the FAB, a snackbar appears, shown as 1 in the screenshot below.



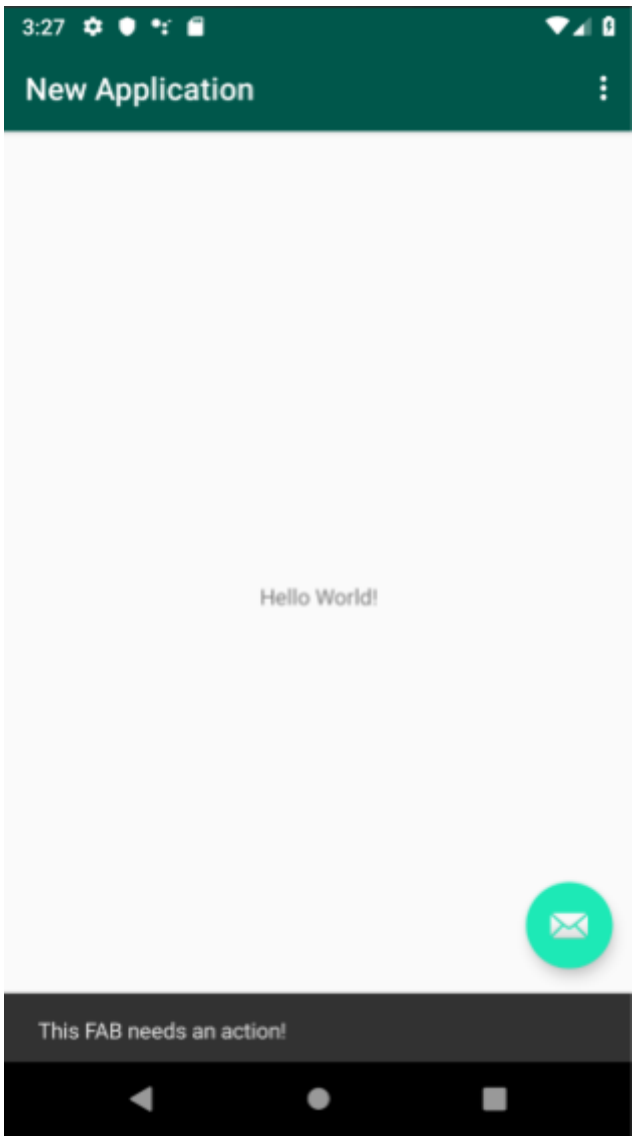
4. Change the snackbar text. To do this, open `MainActivity` and look for the stub code in `onCreate()` that sets an `onClick()` listener for the button. Change `"Replace with your own action"` to something else. For example:

```
fab.setOnClickListener { view ->
   Snackbar.make(view, "This FAB needs an action!", Snackbar.LENGTH_LONG)
       .setAction("Action", null).show()
}
```

5. The FAB uses the app's accent color, so one way to change the FAB's color is to change the accent color. To change accent color, open the **res > values > colors.xml** file and change the `colorAccent` attribute, as shown below. (For help choosing colors, see the Material Design color system.)

```
<color name="colorAccent">#1DE9B6</color>
```

6. Run the app. The FAB uses the new color, and the snackbar text has changed.



Tip: For details on the XML syntax for accessing resources, see Accessing your app resources.

Step **3: Explore how to add activities using templates**

For the codelabs in this course so far, you've used the Empty Activity and Basic Activity templates to start new projects. You can also use activity templates when creating activities after your project has been created.

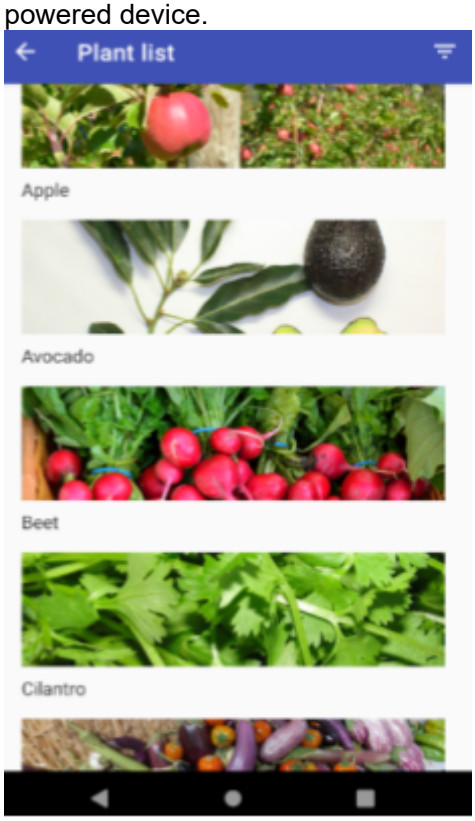
1. Create an app project or choose an existing project.
2. In the **Project > Android** pane, right-click on the **java** folder.
3. Select **New > Activity > Gallery**.
4. Add an activity to the app by selecting one of the `Activity` templates. For example, select **Navigation Drawer Activity** to add an `Activity` that has a navigation drawer.
5. To display the activity in the layout editor, double-click the activity's layout file (for example `activity_main2.xml`). Use the **Design** tab and the **Text** tab to switch between the activity's layout preview and layout code.

4. Task: Learn from sample code

The Google Samples repositories on GitHub provide Kotlin Android code samples that you can study, copy, and incorporate into your projects.

Step **1: Download and run a Kotlin Android code sample**

1. In a browser, navigate to github.com/android.
2. For **Language**, select **Kotlin**.
3. Select a Kotlin Android sample app that's been modified recently and download the app's project code. For this example, download the zip file for the android-sunflower app, which demonstrates some of the components of Android Jetpack.
4. In Android Studio, open the android-sunflower-master project.
5. Accept any updates that Android Studio recommends, then run the app on an emulator or Android-powered device.

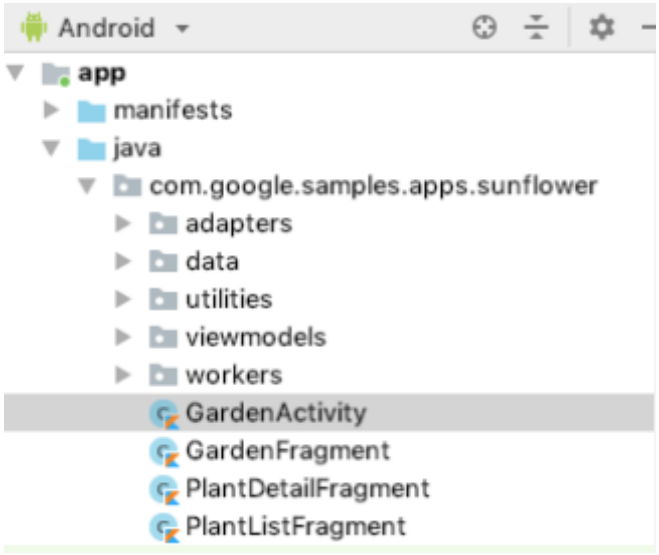


> **Note:** The samples contained in the Google Samples repo on GitHub are meant as a starting point for further development. We encourage you to design and build your own ideas into these samples.

Step 2: Explore a Kotlin Android code sample

Now that you have the Android Sunflower sample app open in Android Studio, learn about the app and explore its project files.

1. For information about what a sample app is demonstrating, visit the app's README file in GitHub. For this example, see the Android Sunflower README.
2. In Android Studio, open one of the Kotlin activity files in the app, for example `GardenActivity.kt`.



3. In `GardenActivity.kt`, find a class, type, or procedure that you're not familiar with and look it up in the Android Developer documentation. For example, to learn more about the `setContentView()` method, search on developer.android.com to find `setContentView()`.

Step 3: Change the launcher icon

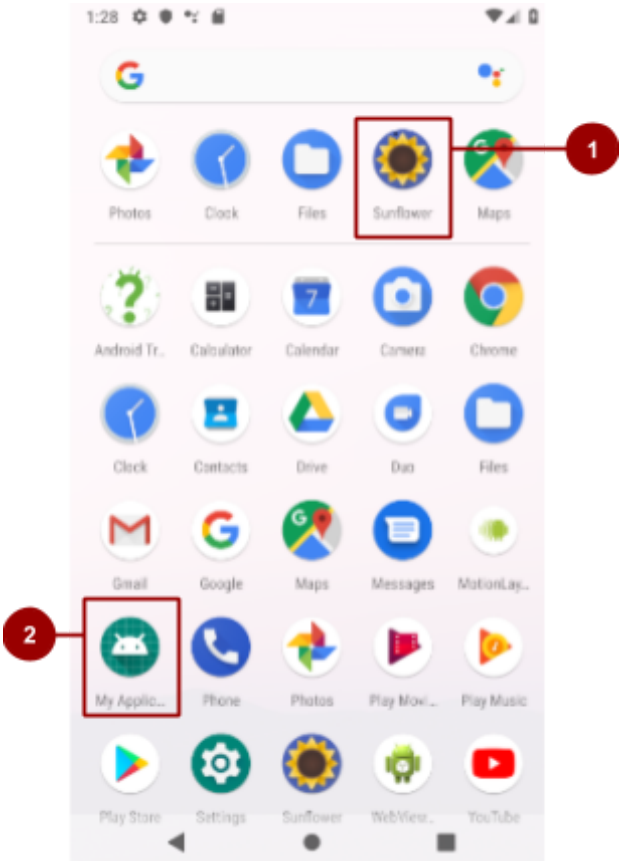
In this step, you change the launcher icon for the Android Sunflower sample app. You add a clip-art image and use it to replace the current Android Sunflower launcher icon.

Launcher icons

Each app that you create with Android Studio starts with a default launcher icon that represents the app. Launcher icons are sometimes called *app icons* or *product icons*.

If you publish an app on Google Play, the app's launcher icon appears in the app's listing and search results in the Google Play store.

After an app is installed on an Android-powered device, the app's launcher icon appears on the device's home screen and elsewhere on the device. For example, the Android Sunflower app's launcher icon appears in the device's **Search Apps** window, shown as 1 in the screenshot below. The default launcher icon, shown as 2 below, is used initially for all app projects that you create in Android Studio.

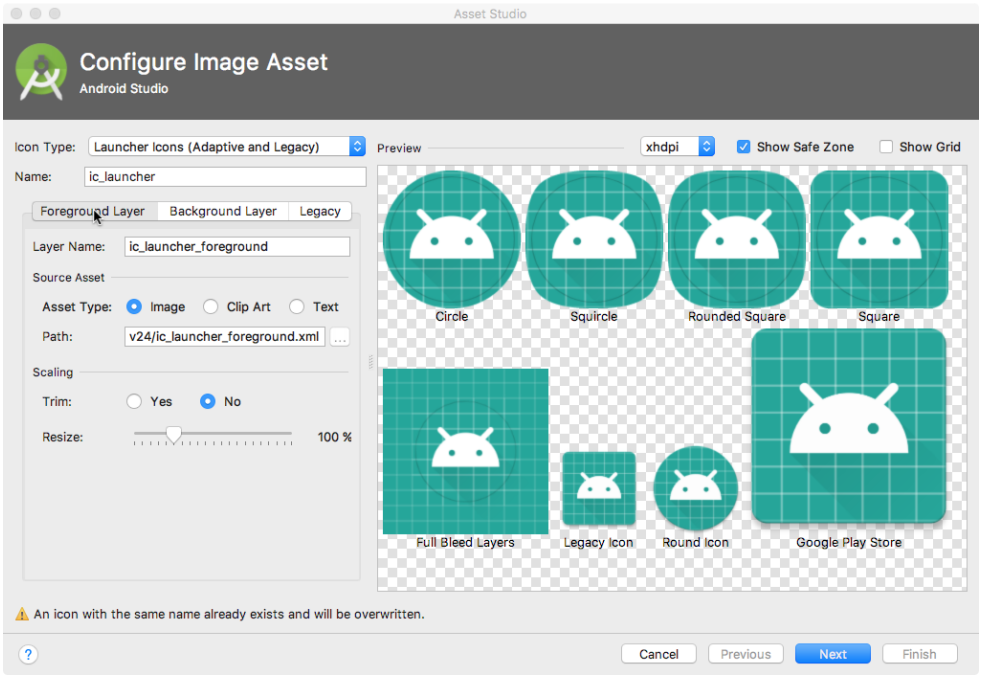


Changing the launcher icon

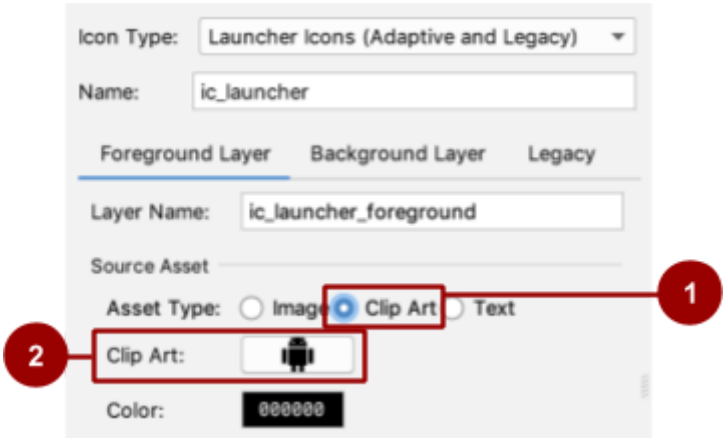
Going through the process of changing the launcher icon introduces you to Android Studio's image asset features.

In Android Studio, here's how to change the Android Sunflower app's launcher icon:

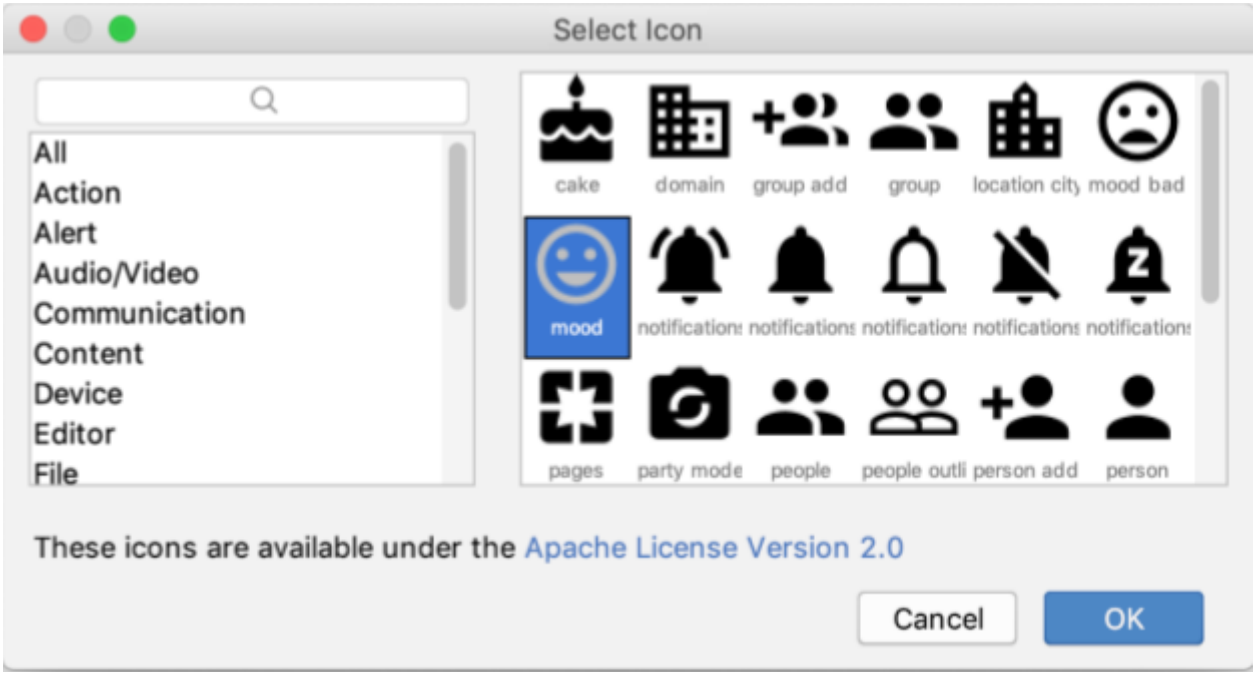
1. In the **Project > Android** pane, right-click (or Control+click) the **res** folder. Select **New > Image Asset**. The **Configure Image Asset** dialog appears.



2. In the **Icon Type** field, select **Launcher Icons (Adaptive & Legacy)** if it's not already selected. Click the **Foreground Layer** tab.
3. For the **Asset Type**, select **Clip Art**, shown as 1 in the screenshot below.

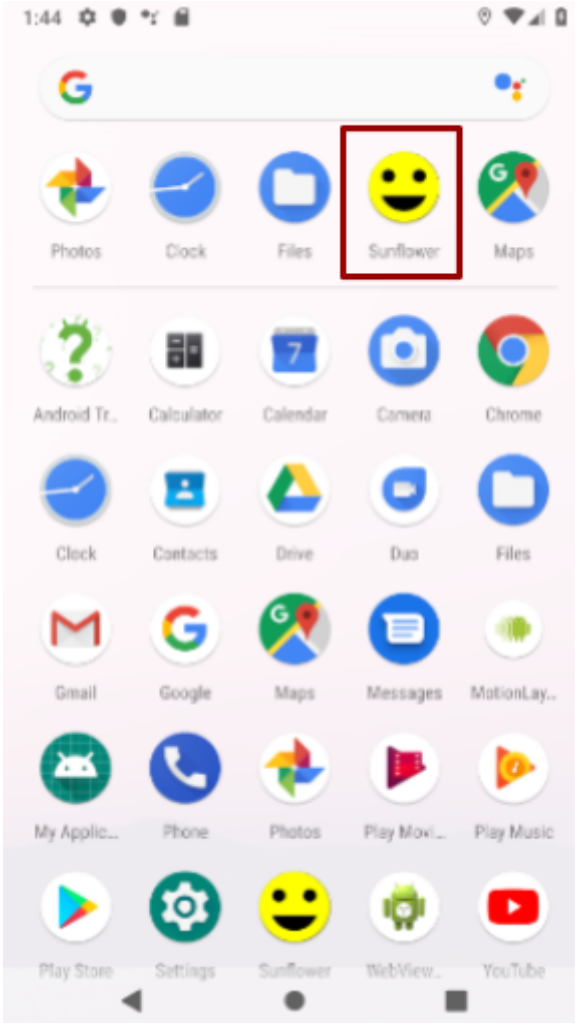


4. Click the robot icon in the **Clip Art** field, shown as 2 in the screenshot above. The **Select Icon** dialog appears, showing the Material Design icon set. 5. Browse the **Select Icon** dialog, or search for an icon by name. Select an icon, such as the **mood** icon to suggest a good mood. Click **OK**.



6. In the **Configure Image Asset** dialog, click the **Background Layer** tab. For the **Asset Type**, select **Color**. Click the color chip and select a color to use as the background layer for the icon.
7. Click the **Legacy** tab and review the default settings. Confirm that you want to generate legacy, round, and Google Play Store icons. Click **Next**.
8. The **Confirm Icon Path** dialog appears, showing where icon files are being added and overwritten. Click **Finish**.
9. Run the app on an AVD emulator or Android-powered device.

Android Studio automatically adds the launcher images to the **mipmap** directories for the different screen densities. The Android Sunflower app now uses the new clip-art icon as its launch icon.



> **Tip**: To learn about designing effective launcher icons, see the Material Design Product icons guide.

4. Run the app again. Make sure the new launcher icon appears in the Search Apps screen.

5. Task: Explore docs and other resources

Step 1: Explore the official Android documentation

Explore a few of the most useful Android documentation sites and become familiar with what's available:

1. Go to developer.android.com. This official Android developer documentation is kept current by Google.
2. Go to developer.android.com/design/. This site offers guidelines for designing the look and functionality of high-quality Android apps.
3. Go to material.io, which is a site about Material Design. Material Design is a conceptual design philosophy that outlines how all apps, not just Android apps, should look and function on mobile devices. Navigate the links to learn more about Material Design. For example, to learn about the use of color, click the **Design** tab, then select **Color**.
4. Go to developer.android.com/docs/ to find API information, reference documentation, tutorials, tool guides, and code samples.
5. Go to developer.android.com/distribute/ to find information about publishing an app on Google Play. Google Play is Google's digital distribution system for apps developed with the Android SDK. Use the Google Play Console to grow your user base and startearning money.

Step 2: Explore content from the Android team and Google Search

1. Explore the Android Developer YouTube channel, which is a great source of tutorials and tips.
2. Visit the official Android blog, where the Android team posts news and tips.
3. Enter a question into Google Search, and the Google Search engine collects relevant results from various resources. For example, use Google Search to ask the question, "What is the most popular Android OS version in India?" You can even enter error messages in Google Search.

Step 3: **Search on Stack Overflow**

Stack Overflow is a community of programmers helping each other. If you run into a problem, chances are high that someone has already posted an answer.

1. Go to Stack Overflow.
2. In the search box, enter a question such as "How do I set up and use ADB over Wi-Fi?" You can search on Stack Overflow without registering, but if you want to post a new question or answer a question, you need to register.
3. In the search box, enter `[android]`. The `[]` brackets indicate that you want to search for posts that have been tagged as being about Android.
4. You can combine tags and search terms to make your search more specific. Try these searches:

- `[android] and [layout]`
- `[android] "hello world"`

> **Tip:** To learn more about the many ways in which you can search on Stack Overflow, see theStack Overflow help center.