1.0: Create Your First Android App

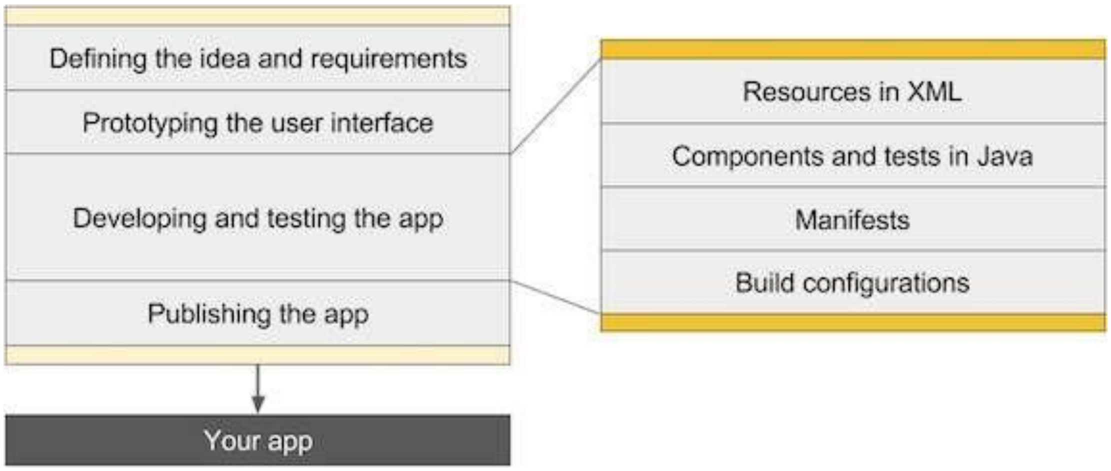
Contents:

This chapter explains how to create applications using the Integrated Development Environment (IDE) for Android Studio.

The development process

A project for an Android app starts with an idea and a list of the requirements needed to make it a reality. The project goes through design, development, and testing as it progresses.

The development process is depicted in the above graphic at a high level and includes the following steps:


Defining the idea and requirements
Prototyping the user interface
Developing and testing the app
Publishing the app
Your app
Resources in XML
Components and tests in Java
Manifests
Build configurations

- Defining the concept and the required components: Most apps begin with a concept for what they should be able to perform, supported by market and user research. The requirements for the app are established at this stage.
- Prototyping the user interface: Use sketches, mockups, and prototypes to depict how the user interface might appear and function.
- Developing and testing the app: An app is made up of a few different tasks. You can use Android Studio to complete the following tasks for each activity, in no particular order:
- Create the layout: Using the Extensible Markup Language (XML), arrange UI elements on the screen in a layout and assign string resources and menu items.
- Write the Java code: Use tools for testing and debugging, and create source code for the components and tests.
- Register the activity: Declare the activity in the manifest file.
- Define the build: Use tools for testing and debugging, and create source code for the components and tests.
- Publishing the app: Assemble the final APK (package file) and distribute it through channels such as the Google Play.

Using Android Studio

An integrated development environment for building apps for all Android devices, Android Studio offers tools for the development process' testing and publishing phases. The development environment consists of a flexible build system, numerous testing tools and frameworks, and code templates with sample code for typical app features.

Double-click the Android Studio application icon to launch it after the Android Studio IDE has been successfully installed. In the Welcome window, select Start a new Android Studio project. Give the project the same name as the app you intend to create. Keep in mind that apps published to the Google Play store must have a distinctive package name when selecting a unique company domain. Since domain names are distinctive, adding your name or the domain name of your business before the app's name should produce a sufficiently distinctive package name. You can accept the default example domain if you don't intend to publish the app. Be mindful that changing the package name later requires additional work.

Choose your project type

You can choose the type of project you want to create from the categories of device form factors that are displayed in the Templates pane of the New Project screen that appears. The project templates for phones and tablets, for instance, are shown in figure 1.

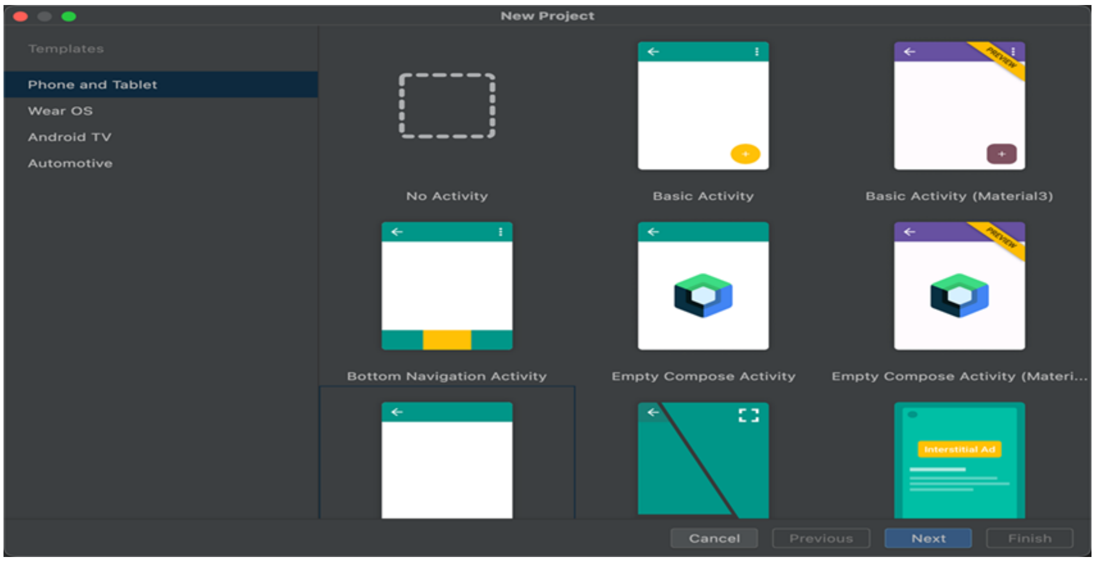
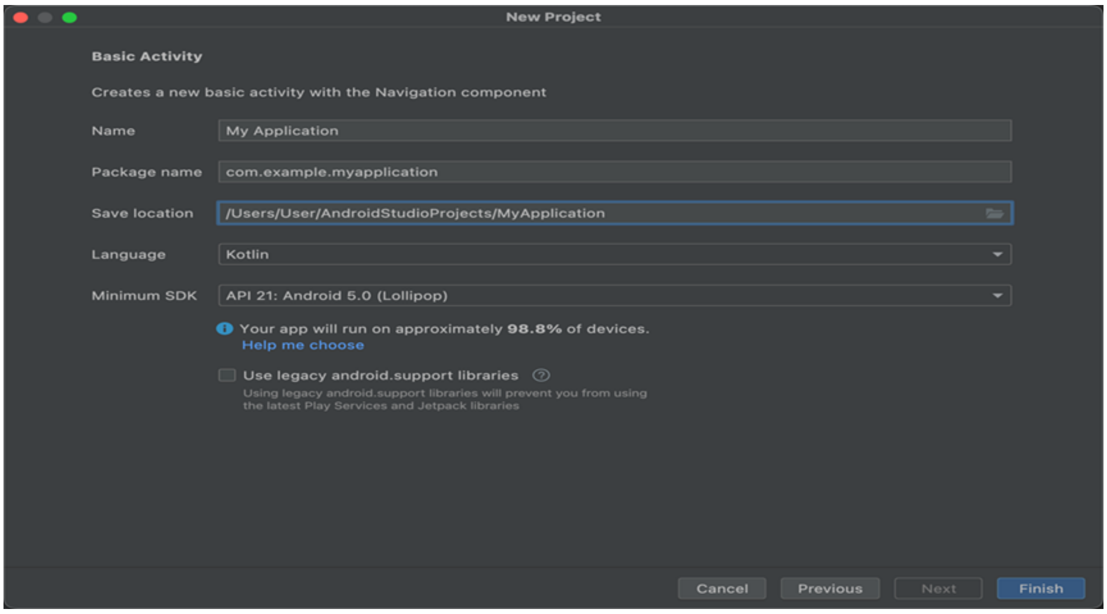


Figure 1. Choose the type of project you want to create on the New Project screen.

Selecting the type of project you want to create lets Android Studio include sample code and resources in your project to help you get started. Once you select your project type, click Next.

Configure your project

The next step in creating your project is to configure some settings, as shown in figure 2. If you're creating a Native C++ project, read Create a new project with C/C++ support to learn more about the options you need to configure.


New Project

Basic Activity

Creates a new basic activity with the Navigation component

Name: My Application

Package name: com.example.myapplication

Save location: /Users/User/AndroidStudioProjects/MyApplication

Language: Kotlin

Minimum SDK: API 21: Android 5.0 (Lollipop)

Your app will run on approximately **98.8%** of devices.
Help me choose

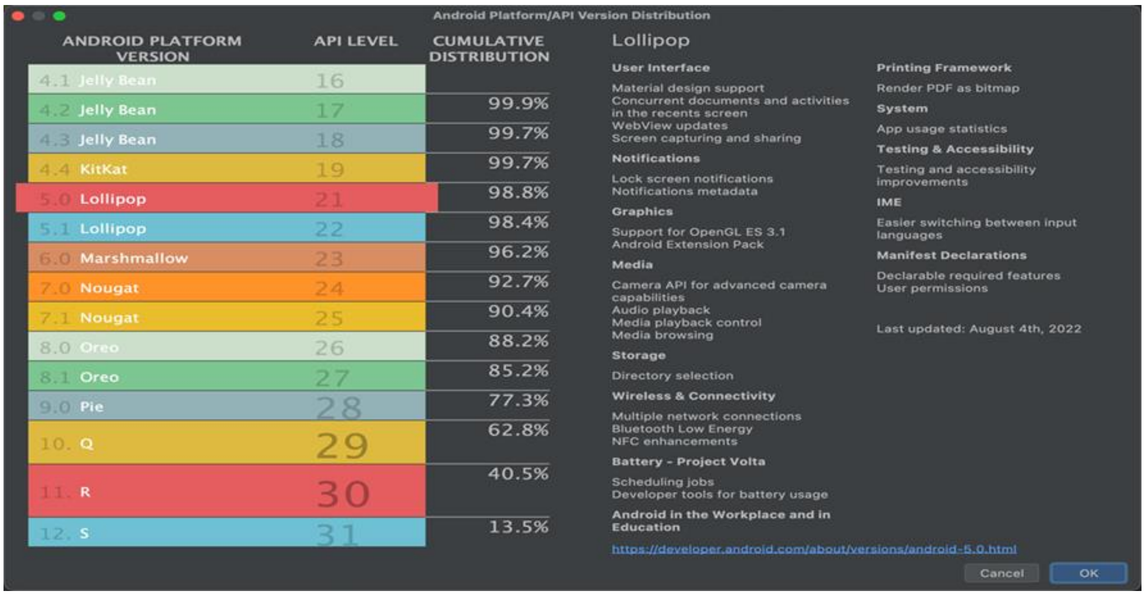
☐ Use legacy android.support libraries
Using legacy android.support libraries will prevent you from using the latest Play Services and Jetpack libraries

Figure 2. Configure your new project with a few settings.

1. Specify the **Name** of your project.
2. Specify the **Package name**. By default, this package name becomes your project's namespace (used to access your project resources) and your project's application ID (used as the ID for publishing). To learn more, see [Configure the app module](#).

3. Specify the **Save location** where you want to locally store your project.
4. Select the **Language**, Kotlin or Java, you want Android Studio to use when creating sample code for your new project. Keep in mind that you aren't limited to using only that language in the projec
5. Select the **Minimum API level** you want your app to support. When you select a lower API level, your app can't use as many modern Android APIs. However, a larger percentage of Android devices can run your app. The opposite is true when selecting a higher API level.

If you want to see more data to help you decide, click **Help me choose**. This displays a dialog showing the cumulative distribution for the API level you have selected and lets you see the impact of using different minimum API levels



Android Platform/API Version Distribution

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.1 Jelly Bean	16	
4.2 Jelly Bean	17	99.9%
4.3 Jelly Bean	18	99.7%
4.4 KitKat	19	99.7%
5.0 Lollipop	21	98.8%
5.1 Lollipop	22	98.4%
6.0 Marshmallow	23	96.2%
7.0 Nougat	24	92.7%
7.1 Nougat	25	90.4%
8.0 Oreo	26	88.2%
8.1 Oreo	27	85.2%
9.0 Pie	28	77.3%
10. Q	29	62.8%
11. R	30	40.5%
12. S	31	13.5%

Lollipop

User Interface
Material design support
Concurrent documents and activities in the recents screen
WebView updates
Screen capturing and sharing

Notifications
Lock screen notifications
Notifications metadata

Graphics
Support for OpenGL ES 3.1
Android Extension Pack

Media
Camera API for advanced camera capabilities
Audio playback
Media playback control
Media browsing

Storage
Directory selection

Wireless & Connectivity
Multiple network connections
Bluetooth Low Energy
NFC enhancements

Battery - Project Volta
Scheduling jobs
Developer tools for battery usage

Android in the Workplace and in Education

Printing Framework
Render PDF as bitmap

System
App usage statistics

Testing & Accessibility
Testing and accessibility improvements

IME
Easier switching between input languages

Manifest Declarations
Declarable required features
User permissions

Last updated: August 4th, 2022

https://developer.android.com/about/versions/android-5.0.html

6. Your project is configured to use AndroidX libraries by default, which replace the Android Support libraries. To use the legacy support libraries instead, select **Use legacy android.support libraries**. However, this is not recommended, as the legacy support libraries are no longer supported. To learn more, read the AndroidX overview.
7. When you're ready to create your project, click Finish. Android Studio creates your new project with some basic code and resources to get you started. If you decide to add support for a different device form factor later, you can add a module to your project. And if you want to share code and resources between modules, you can do so by creating an Android library.

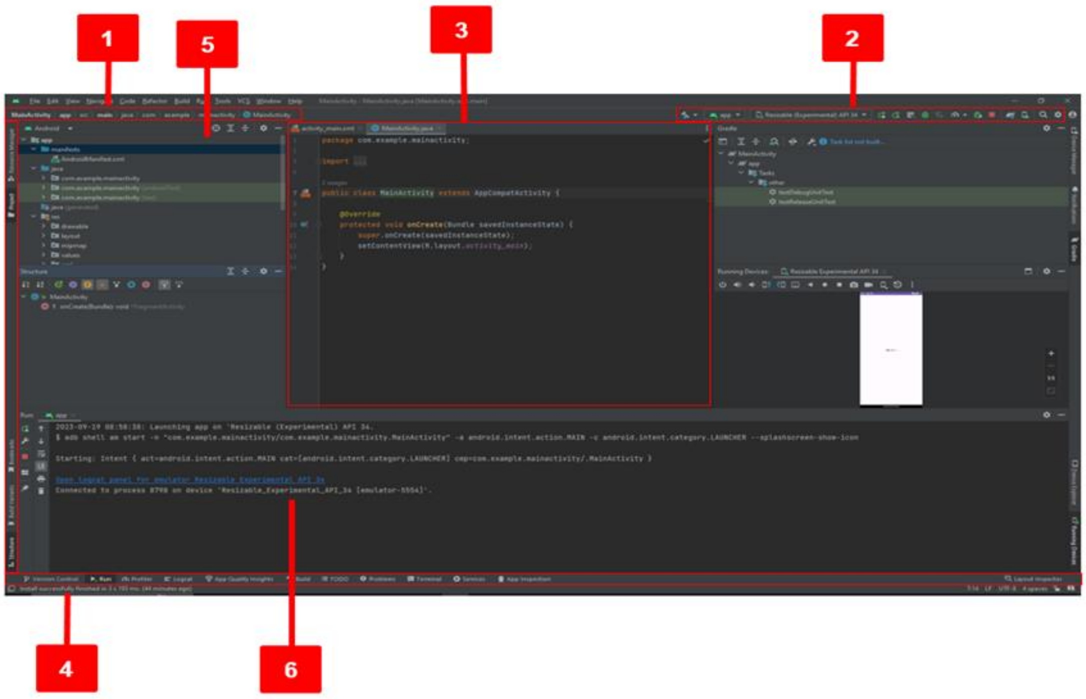
Import an existing project

To import an existing local project into Android Studio, proceed as follows:

1. Click File > New > Import Project.
2. In the window that appears, navigate to the root directory of the project you want to import.

Click OK.

Android Studio Pane



```
package com.example.mainactivity;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

```
2023-09-19 08:58:38: Launching app on 'Resizable (Experimental) API 34.
$ adb shell am start -n "com.example.mainactivity/com.example.mainactivity.MainActivity" -a android.intent.action.MAIN -c android.intent.category.LAUNCHER --splashscreen-show-icon

Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.example.mainactivity/.MainActivity }

Open logcat panel for emulator Resizable Experimental API 34
Connected to process 8798 on device 'Resizable_Experimental_API_34 [emulator-5554]'.
```

The Android Studio main window is made up of several logical areas, or *panes*, as shown in the figure below.

In the above figure:

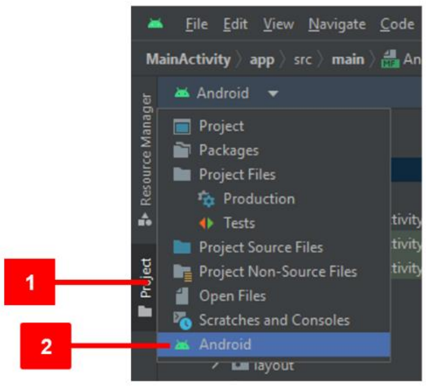
1. The **Toolbar**. The toolbar performs a variety of tasks, such as starting the Android app and accessing its tools.
2. The **Navigation Bar**. The navigation bar enables editing of open files and project navigation. It offers a more condensed perspective of the project structure.
3. The **Editor Pane**. This pane displays the project's selected file's contents. For instance, this pane displays the layout editor with tools to edit the layout after selecting a layout (as shown in the figure). This pane displays the code with editing tools after choosing a Java code file.
4. The **Status Bar*.*** The project's and Android Studio's status, as well as any alerts or messages, are shown in the status bar. The status bar displays the build progress.
5. The **Project Pane**. The project pane shows the project files and project hierarchy.
6. The **Monitor Pane**. The TODO list for managing tasks, the Android Monitor for tracking app execution (illustrated in the figure), the logcat for viewing log messages, and the Terminal application for carrying out Terminal operations are all accessible from the monitor pane.

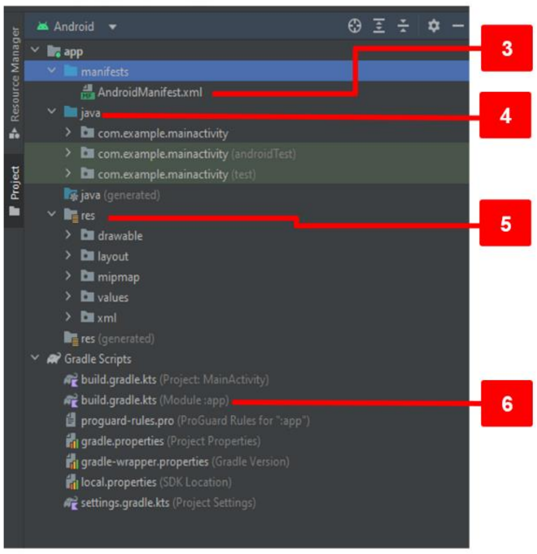
Tip: You can organize the main window to give yourself more screen space by hiding or moving panes. You can also use keyboard shortcuts to access most features. See Keyboard Shortcuts for a complete list.

Exploring a project

The AndroidManifest.xml file, component source-code files, and related resource files are all included in each project in Android Studio. As shown below, Android Studio presents your project files in the Project: Android view in the left tool pane by default, according to the file type. The view offers simple access to the important files for your project.

Click the vertical Project tab in the far left column of the Project pane and select Android from the pop-up menu at the top to return to this view after leaving another view, as shown in the illustration below.




Android
app
manifests
AndroidManifest.xml
java
com.example.mainactivity
com.example.mainactivity (androidTest)
com.example.mainactivity (test)
java (generated)
res
drawable
layout
mipmap
values
xml
res (generated)
Gradle Scripts
build.gradle.kts (Project: MainActivity)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules for ":app")
gradle.properties (Project Properties)
gradle-wrapper.properties (Gradle Version)
local.properties (SDK Location)
settings.gradle.kts (Project Settings)
3
4
5
6

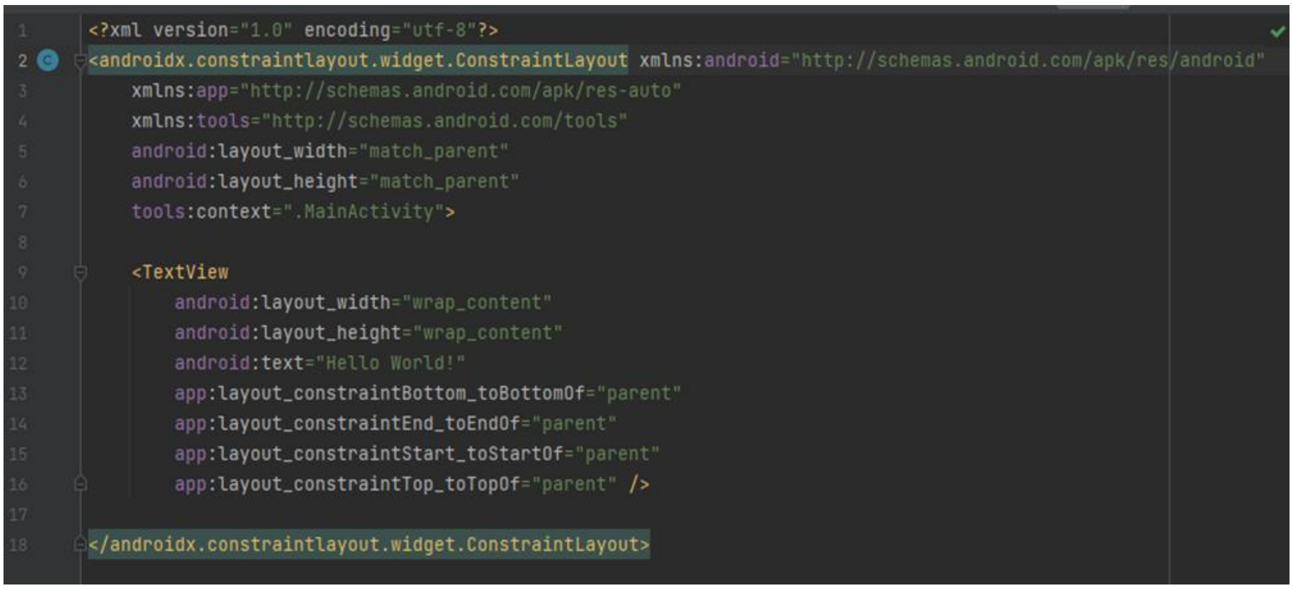
In the figure above:

1. The **Project** tab. Click to show the project view.
2. The **Android** selection in the project drop-down menu.
3. The **AndroidManifest.xml** file. Used for specifying information about the app for the Android runtime environment. The template you choose creates this file.
4. The **java** folder.Activities, tests, and other parts written in Java source code are contained in this folder. Each activity, service, and additional component has a Java class definition, typically in its own file. MainActivity is typically the name of the first activity (screen) that the user sees, which also initializes resources for the entire app.
5. The **res** folder. This folder holds resources, such as XML layouts, UI strings, and images. An activity usually is associated with an XML resource file that specifies the layout of its views. This file is usually named after its activity or function.
6. The **build.gradle (Module: App)** file. This file specifies the module's build configuration. The template you choose creates this file, which defines the build configuration, including the `minSdkVersion` attribute that declares the minimum version for the app, and the `targetSdkVersion` attribute that declares the highest (newest) version for which the app has been optimized. This file also includes a list of *dependencies*, which are libraries required by the code — such as the AppCompat library for supporting a wide range of Android versions.

Viewing the Android Manifest

Contains tests, activities, and other parts written in Java source code. Each activity, service, and additional component has a Java class definition, typically in its own file. MainActivity is typically the name of the first activity (screen) that the user sees, which also initializes resources for the entire app.

To view this file, expand the manifests folder in the Project: Android view, and double-click the file (**AndroidManifest.xml**). Its contents appear in the editing pane as shown in the figure below.



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

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

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

Android namespace and application tag

The Android Manifest is coded in XML and always uses the Android namespace:

```
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.android.helloworld">
```

The `package` expression shows the unique package name of the new app. Do not change this once the app is published.

```
<application
...
</application>
```

The `<application` tag, with its closing `</application>` tag, defines the manifest settings for the entire app.

Automatic backup

The `android:allowBackup` attribute enables automatic app data backup:

```
...
android:allowBackup="true"
...
```

The app can be `android:allowBackup` up and restored when the `android:allowBackup` attribute is set to true. When configuring apps, users put time and effort into it. All that careful configuration may be rendered useless by switching to a new device. Nearly all app data is automatically backed up by the system by default, and this is done without the developer having to add any additional app code. Devices running Android 6.0 and higher automatically create backups of app data to the cloud for apps whose target SDK version is Android 6.0 (API level 23) and higher because the `android:allowBackup` attribute defaults to true if omitted. The `android:allowBackup` attribute must be added explicitly and set to true for apps with an API level lower than 22.

The app icon

The `android:icon` attribute sets the icon for the app:

```
...
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
...
```

The `android:icon` attribute assigns an icon in the **mipmap** folder (inside the **res** folder in Project: Android view) to the app. The icon appears in the Launcher for launching the app. The icon is also used as the default icon for app components.

App label and string resources

As you can see in the previous figure, the `android:label` attribute shows the string `"Hello World"` highlighted. If you click on this string, it changes to show the string resource `@string/app_name`:

```
...
android:label="@string/app_name"
...
```

Tip: Ctrl-click or right-click `app_name` in the edit pane to see the context menu. Choose **Go To > Declaration** to see where the string resource is declared: in the strings.xml file. When you choose **Go To > Declaration** or open the file by double-clicking **strings.xml** in the Project: Android view (inside the **values** folder), its contents appear in the editing pane.

After opening the strings.xml file, you can see that the string name `app_name` is set to `Hello World`. By substituting a different string for the `Hello World` one, you can change the name of the app. Resources for strings are discussed in a different lesson.

The app theme

The `android:theme` attribute sets the app's theme, which defines the appearance of user interface elements such as text:

```
...
android:theme="@style/AppTheme">
...
```

The `theme` attribute is set to the standard theme `AppTheme`. Themes are described in a separate lesson.

Declaring the Android version

Different Android operating systems, such as Android 4.0 or Android 4.4, may be used on various devices. New APIs that were not present in the previous version may be added in each subsequent release. Each version specifies an API level to show which set of APIs is available. Android 1.0, for instance, is at API level 1, while Android 4.4 is at API level 19.

The API level allows a developer to declare the minimum version with which the app is compatible, using the `<uses-sdk>` manifest tag and its `minSdkVersion` attribute.

```
<manifest ... >
    <uses-sdk android:minSdkVersion="14" android:targetSdkVersion="19" />
    ...
</manifest>
```

The `minSdkVersion` attribute declares the minimum version for the app, and the `targetSdkVersion` attribute declares the highest (newest) version which has been optimized within the app. Each successive version of Android provides compatibility for apps that were built using the APIs from previous versions, so the app should *always* be compatible with future versions of Android while using the documented Android APIs.

The targetSdkVersion attribute is significant because it tells the system whether the app should inherit behavior changes in newer versions. It does not prevent an app from being installed on Android versions that are higher (newer) than the specified value. The system assumes that your app requires some backward-compatibility behaviors when running on the most recent version if you don't update targetSdkVersion

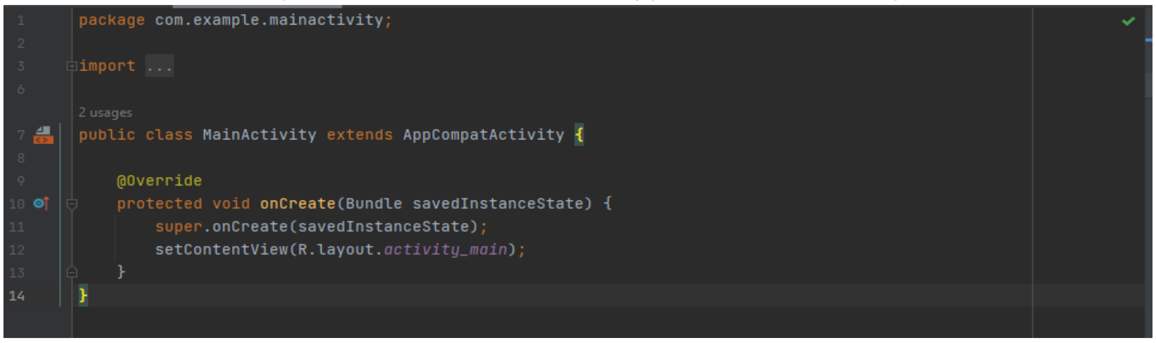
to the most recent version. For instance, alarms created with the AlarmManager APIs are now by default approximate so that the system can batch app alarms and conserve system power. However, the system will maintain the previous API behavior for an app if your target API level is lower than "19" .

Viewing and editing Java code

In the Project: Android view, components are organized into module folders in the java folder and written in Java. Each module's name includes both the app name and the domain name, which might be something like com.example.android.

An illustration of an activity component is provided below:

1. Click the module folder to expand it and show the MainActivity file for the activity written in Java (the MainActivity class).
2.
3. Double-click **MainActivity** to see the source file in the editing pane, as shown in the figure below.



```
package com.example.mainactivity;

import ...

2 usages
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

At the very top of the MainActivity.java file is a package statement that defines the app package. This is followed by an

import block condensed in the above figure, with " ...". Click the dots to expand the block to view it. The import statements import libraries needed for the app, such as the following, which imports the AppCompatActivity library:

```
import android.support.v7.app.AppCompatActivity;
```

Each activity in an app is implemented as a Java class. The following class declaration extends the AppCompatActivity class to implement features in a way that is backward-compatible with previous versions of Android:

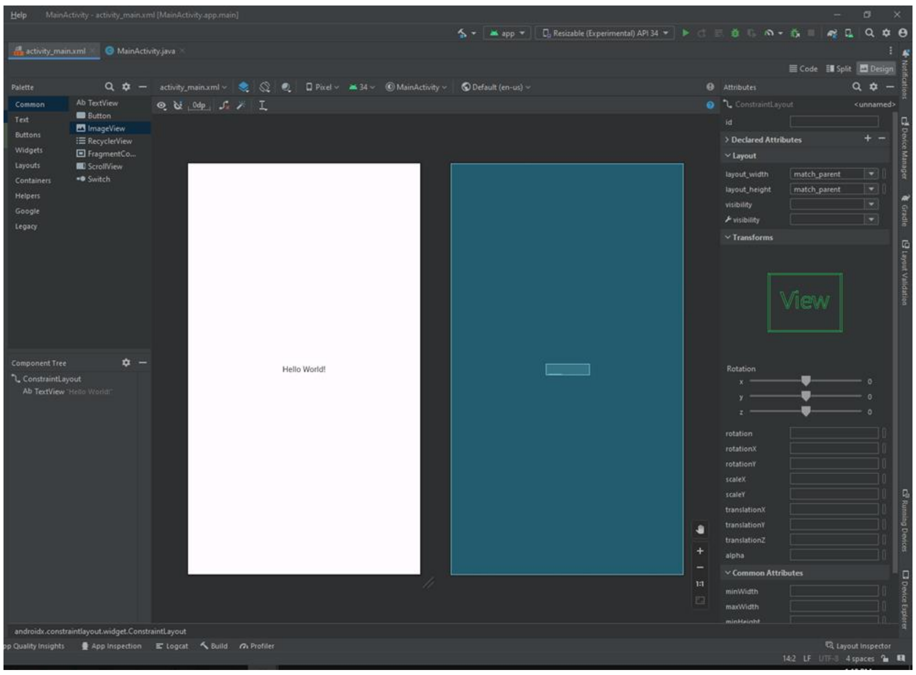
```
public class MainActivity extends AppCompatActivity {
    ...
}
```

As you already know, the Android system must read the app's AndroidManifest.xml file in order to recognize the existence of an activity before starting an app component like one. The AndroidManifest.xml file must therefore contain a list of all activities.

Viewing and editing layouts

Layout resources are written in XML and listed within the **layout** folder in the **res** folder in the Project: Android view. Click **res > layout** and then double-click **activity_main.xml** to see the layout file in the editing pane.

Android Studio shows the Design view of the layout, as shown in the figure below. This view provides a Palette pane of user interface elements, and a grid showing the screen layout.

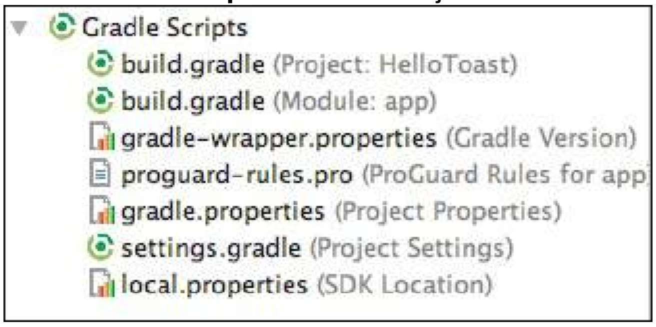


Understanding the build process

The package file format for distributing and installing Android mobile apps is known as the Android application package (APK). Tools and procedures used during the build process automatically convert each project into an APK.
The build system in Android Studio is based on Gradle, with the Android Plugin for Gradle providing additional Android-specific functionality. This build system is accessible from the Android Studio menu as an integrated tool.

Understanding build.gradle files

When you create a project, Android Studio automatically generates the necessary build files in the **Gradle Scripts** folder in Project: Android.



Gradle Scripts
- build.gradle (Project: HelloToast)
- build.gradle (Module: app)
- gradle-wrapper.properties (Gradle Version)
- proguard-rules.pro (ProGuard Rules for app)
- gradle.properties (Project Properties)
- settings.gradle (Project Settings)
- local.properties (SDK Location)

Each project has the following:

build.gradle (Project: *apptitle*)

This file defines build configurations that apply to all modules in your project and is the top-level build file for the entire project. It can be found in the root project directory. It is not recommended to edit the Android Studio-generated file to add app dependencies.

build.gradle (Module: app)

Android Studio creates separate `build.gradle (Module: app)` files for each module. You can edit the build settings to provide custom packaging options for each module, such as additional build types and product flavors, and to override settings in the manifest or top-level build.gradle file. This file is most often the file to edit when changing app-level configurations, such as declaring dependencies in the `dependencies` section. The following shows the contents of a project's `build.gradle (Module: app)` file:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
    buildToolsVersion "24.0.1"
    defaultConfig {
        applicationId "com.example.android.helloworld2"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
}
```

The build.gradle files use Gradle syntax. Gradle is a Domain Specific Language (DSL) for describing and manipulating the build logic using Groovy, which is a dynamic language for the Java Virtual Machine (JVM). You don't need to learn Groovy to make changes, because the Android Plugin for Gradle introduces most of the DSL elements you need.

Tip: To learn more about the Android plugin DSL, read the DSL reference documentation.

Plugin and Android blocks

In the above `build.gradle (Module: app)` file, the first statement applies the Android-specific Gradle plug-in build tasks:

```
apply plugin: 'com.android.application'

android {
   ...
}
```

The `android { }` block specifies the following for the build:

- The target SDK version for compiling the code:

```
compileSdkVersion 24
```

- The version of the build tools to use for building the app:

```
buildToolsVersion "24.0.1"
```

The defaultConfig block

Core settings and entries for the app are specified in a `defaultConfig { }` block within the `android { } block:`

```
   ...
   defaultConfig {
      applicationId "com.example.hello.helloworld"
      minSdkVersion 15
      targetSdkVersion 23
      versionCode 1
      versionName "1.0"
            testInstrumentationRunner
               "android.support.test.runner.AndroidJUnitRunner"
   }
...
```

The `minSdkVersion` and `targetSdkVersion` settings override any AndroidManifest.xml settings for the minimum SDK version and the target SDK version. See "Declaring the Android version" previously in this chapter for background information on these settings.

The `testInstrumentationRunner` statement adds the instrumentation support for testing the user interface with Espresso and UIAutomator. These are described in a separate lesson.

Build types

Build types for the app are specified in a `buildTypes { }` block, which controls how the app is built and packaged.

```
...
buildTypes {
   release {
      minifyEnabled false
       proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                                'proguard-rules.pro'
   }
}
...
```

The build type specified is `release` for the app's release. Another common build type is `debug`. Configuring build types is described in a separate lesson.

Dependencies

Dependencies for the app are defined in the `dependencies { }` block, which is the part of the build.gradle file that is most likely to change as you start developing code that depends on other libraries. The block is part of the standard Gradle API and belongs *outside* the `android { }` block.

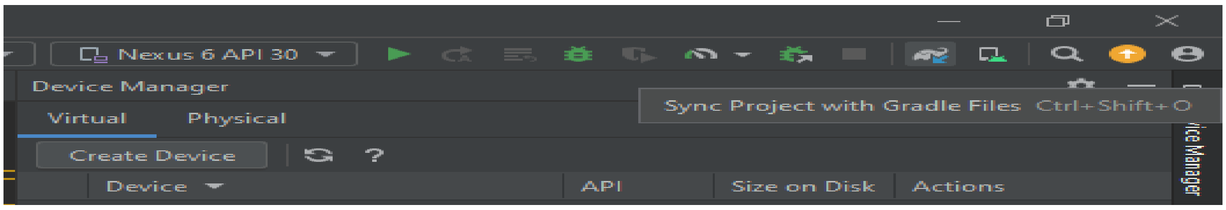
```
...
dependencies {
   compile fileTree(dir: 'libs', include: ['*.jar'])
   androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
exclude group: 'com.android.support', module: 'support-annotations'
   })
   compile 'com.android.support:appcompat-v7:24.2.0'
   testCompile 'junit:junit:4.12'
}
```

In the above snippet, the statement `compile fileTree(dir: 'libs', include: ['*.jar'])` adds a dependency of all ".jar" files inside the `libs` directory. The `compile` configuration compiles the main application — everything in it is added to the compilation classpath, and also packaged into the final APK.

Syncing your project

Android Studio requires that you sync the project files after making changes to the build configuration files so that it can import the changes and perform checks to ensure the configuration won't result in build errors.

To sync the project files, click **Sync Now** in the notification bar that appears when making a change, or click **Sync Project** from the menu bar. If Android Studio notices any errors with the configuration — for example, if the source code uses API features that are only available in an API level higher than the `compileSdkVersion` — the Messages window appears to describe the issue.



Running the app on an emulator or a device

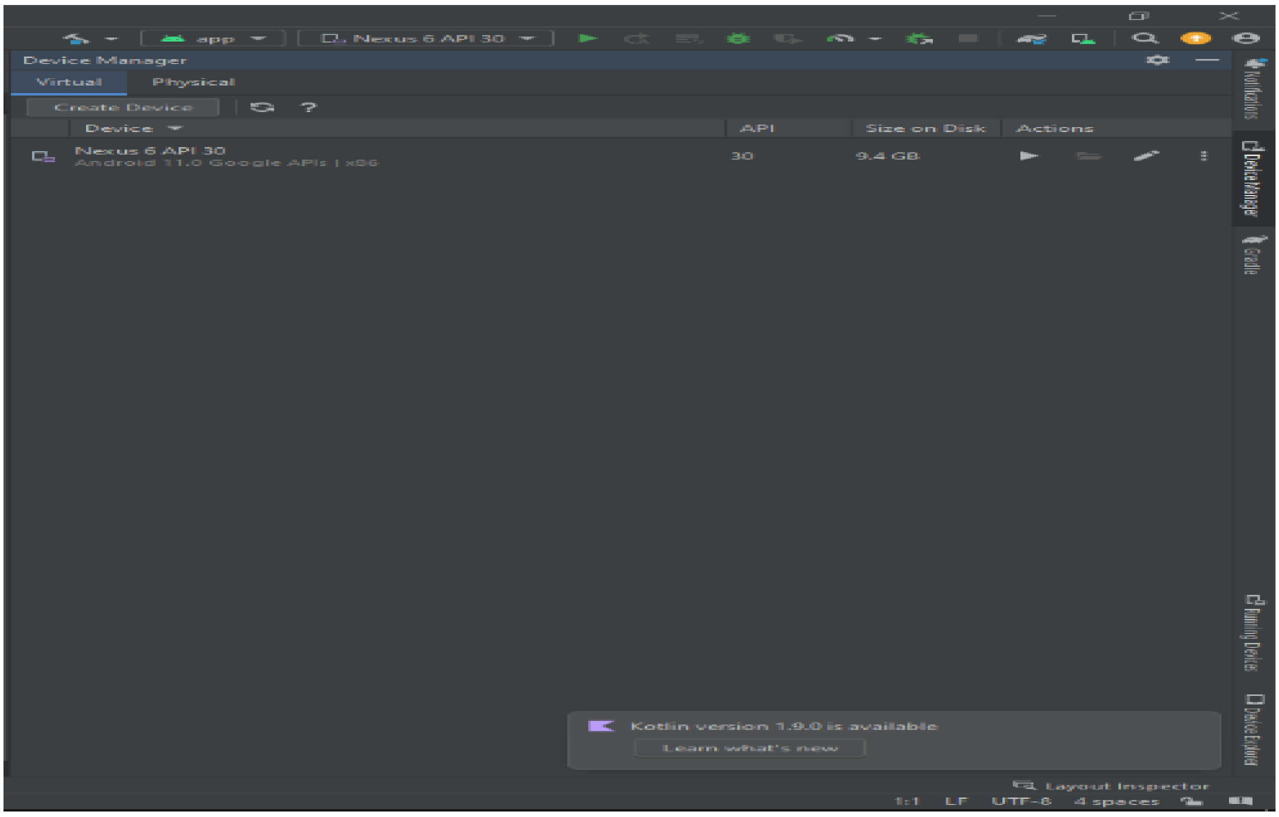
With virtual device emulators, you can test an app on various gadgets, including tablets and smartphones, with various API levels for various Android versions, to ensure that it functions and looks good for the majority of users. You don't have to rely on having a physical device on hand for app development, even though it's a good idea.
A virtual device or emulator that simulates the settings for a specific type of Android device is created by the Android Virtual Device (AVD) manager. Use the AVD Manager to specify a device's hardware specs and API level, then save the information as a virtual device configuration. When you launch the Android emulator, it reads a predetermined configuration to build an emulated device that functions exactly like the real thing on your computer.

Creating a virtual device

To run an emulator on your computer, use the AVD Manager to create a configuration that describes the virtual device.

Select **Tools > Android > AVD Manager**, or click the AVD Manager icon in the toolbar.

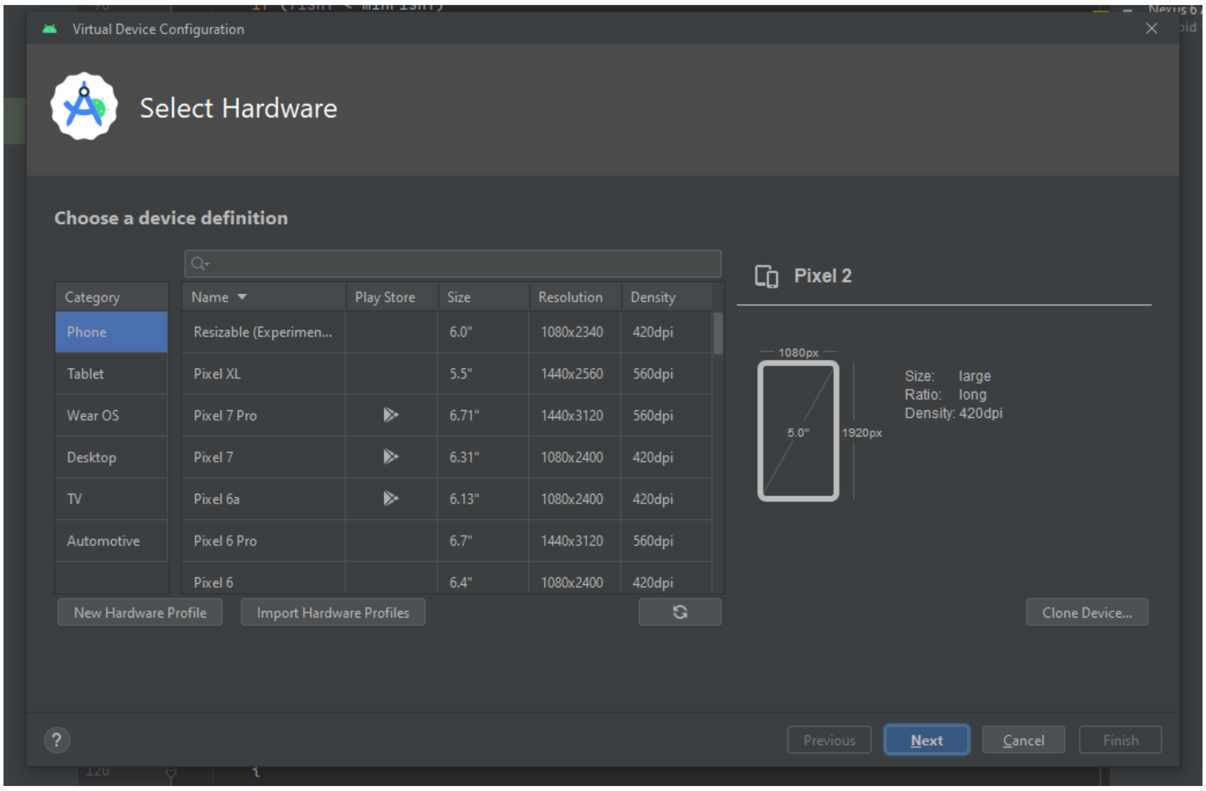
The "Your Virtual Devices" screen appears showing all of the virtual devices created previously. Click the **+Create Virtual Device** button to create a new virtual device.



Device	API	Size on Disk	Actions
Nexus 6 API 30 Android 11.0 Google APIs | x86	30	9.4 GB	

Kotlin version 1.9.0 is available
Learn what's new

You can select a device from a list of predefined hardware devices. For each device, the table shows its diagonal display size (Size), screen resolution in pixels (Resolution), and pixel density (Density). For example, the pixel density of the Nexus 5 device is `xxhdpi`, which means the app uses the icons in the **xxhdpi** folder of the **mipmap** folder. Likewise, the app will use layouts and drawables from folders defined for that density as well.



Category
Phone
Tablet
Wear OS
Desktop
TV
Automotive

Name	Play Store	Size	Resolution	Density
Resizable (Experimen...		6.0"	1080x2340	420dpi
Pixel XL		5.5"	1440x2560	560dpi
Pixel 7 Pro	▶	6.71"	1440x3120	560dpi
Pixel 7	▶	6.31"	1080x2400	420dpi
Pixel 6a	▶	6.13"	1080x2400	420dpi
Pixel 6 Pro		6.7"	1440x3120	560dpi
Pixel 6		6.4"	1080x2400	420dpi

Pixel 2

1080px
5.0"
1920px

Size: large
Ratio: long
Density: 420dpi

You also choose the version of the Android system for the device. The **Recommended** tab shows the recommended systems for the device. More versions are available under the **x86 Images** and **Other Images** tabs.

Running the app on the virtual device

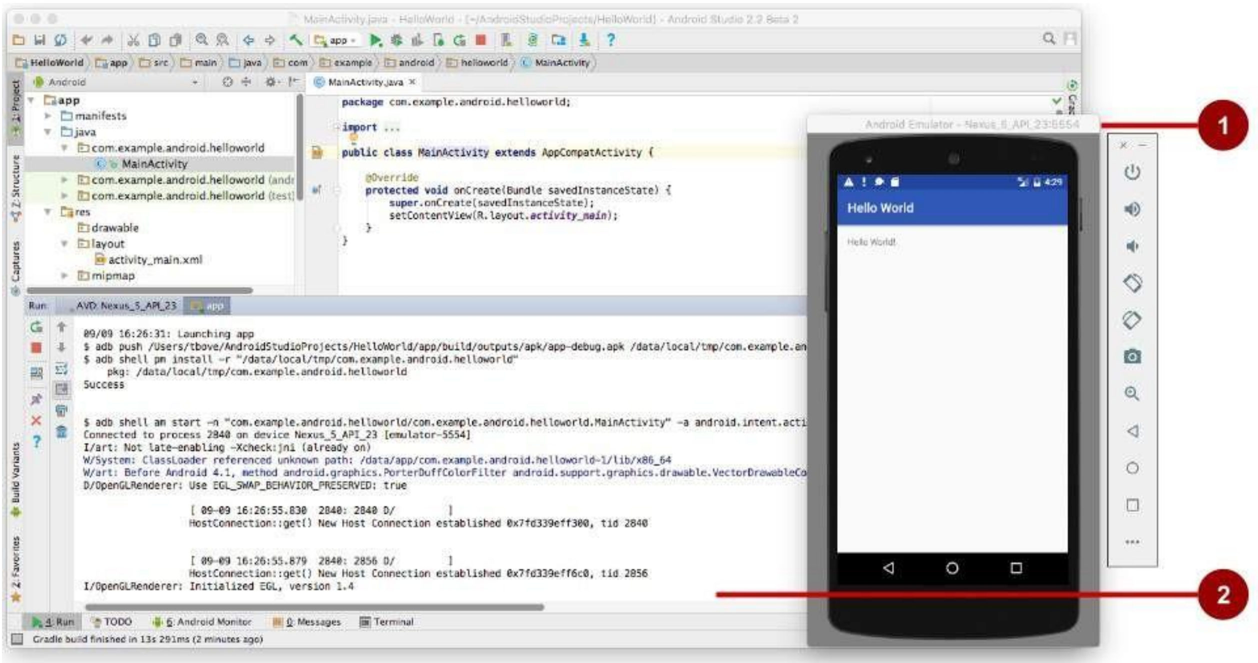
To run the app on the virtual device you created in the previous section, follow these steps:

1. In Android Studio, select **Run > Run app** or click the **Run icon** in the toolbar, Or else just press the F10 button.
2. In the Select Deployment Target window, under Available Emulators, select the virtual device you created, and click **OK**.

The emulator powers on and boots up the same as a real device. Depending on how quickly your computer is, this could take some time. As soon as the emulator is prepared, Android Studio builds the app and launches it there.

The following figure, which also depicts Android Studio's Run pane, should show the app you created using the Empty Activity template ("Hello World") and the actions taken to run the app on the emulator.

Note: When testing on an emulator, it is good practice to start it up once at the very beginning of your session, and not to close it until done so that it doesn't have to go through the boot process again.


MainActivity.java - HelloWorld - [~/AndroidStudioProjects/HelloWorld] - Android Studio 2.2 Beta 2
HelloWorld
app
src
main
java
com
example
android
helloworld
MainActivity
Android
MainActivity.java
manifests
java
com.example.android.helloworld
com.example.android.helloworld (androidTest)
com.example.android.helloworld (test)
res
drawable
layout
activity_main.xml
mipmap
Run
AVD: Nexus_5_API_23
Android Emulator - Nexus_5_API_23:5554
Hello World
Hello World!
4: Run
TODO
6: Android Monitor
0: Messages
Terminal
Gradle build finished in 13s 291ms (2 minutes ago)
1
2

```
package com.example.android.helloworld;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

```
09/09 16:26:31: Launching app
$ adb push /Users/tbove/AndroidStudioProjects/HelloWorld/app/build/outputs/apk/app-debug.apk /data/local/tmp/com.example.an
$ adb shell pm install -r "/data/local/tmp/com.example.android.helloworld"
    pkg: /data/local/tmp/com.example.android.helloworld
Success


$ adb shell am start -n "com.example.android.helloworld/com.example.android.helloworld.MainActivity" -a android.intent.acti
Connected to process 2840 on device Nexus_5_API_23 [emulator-5554]
I/art: Not late-enabling -Xcheck:jni (already on)
W/System: ClassLoader referenced unknown path: /data/app/com.example.android.helloworld-1/lib/x86_64
W/art: Before Android 4.1, method android.graphics.PorterDuffColorFilter android.support.graphics.drawable.VectorDrawableCo
D/OpenGLRenderer: Use EGL_SWAP_BEHAVIOR_PRESERVED: true

                  [ 09-09 16:26:55.830  2840: 2840 D/         ]
                 HostConnection::get() New Host Connection established 0x7fd339eff300, tid 2840


                  [ 09-09 16:26:55.879  2840: 2856 D/         ]
                 HostConnection::get() New Host Connection established 0x7fd339eff6c0, tid 2856
I/OpenGLRenderer: Initialized EGL, version 1.4
```

In the above figure:

1. The **Emulator** running the app.
2. The **Run Pane**. This shows the actions taken to install and run the app.

Running the app on a physical device

Always test your apps on physical device, because users will use the app on physical devices. Even though emulators are very useful, they can't depict every possible device state, including what would happen if an incoming call came in while the app was open. To utilize a physical device to run the app, you need the following:

- . An Android device such as a smartphone or tablet.
- A data cable to connect the Android device to your computer via the USB port.
- To run the app on a hardware device, you might need to take additional actions if you're using Linux or Windows. Check the Using Hardware Devices documentation. On Windows, you may need to install the appropriate USB driver for the device. See OEM USB Drivers.

To let Android Studio communicate with a device, turn on USB Debugging on the Android device. On Android version 4.2 and newer, the Developer options screen is hidden by default. Follow these steps to turn on USB Debugging:

1. On the physical device, open **Settings** and choose **About phone** at the bottom of the Settings screen.
2. Tap the **Build number** information seven times.

   You read that correctly: Tap it *seven times*.

3. Return to the previous screen (**Settings**). **Developer options** now appears at the bottom of the screen. Tap **Developer options**.
4. Choose **USB Debugging**.

Now, connect the device and run the app from Android Studio.

Troubleshooting the device connection

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

1. Disconnect the device from your computer, and then reconnect it.
2. Restart Android Studio.
3. If your computer still does not find the device or declares it "unauthorized":
   i. Disconnect the device from your computer.
   ii. On the device, choose **Settings > Developer Options**.
   iii. Tap **Revoke USB Debugging authorizations**.
   iv. Reconnect the device to your computer.
   v. When prompted, grant authorizations.
4. You may need to install the appropriate USB driver for the device. See Using Hardware Devices documentation.
5. Check the latest documentation, programming forums, or get help from your instructors.

Using the log

You can use the log as a strong debugging tool to examine values, execution paths, and exceptions. The logcat tab of the Android Monitor pane of Android Studio displays your log messages along with general log messages after you add logging statements to an app.

To see the Android Monitor pane, click the **Android Monitor** button at the bottom of the Android Studio main window. The Android Monitor offers two tabs:

- The **logcat** tab. The **logcat** tab displays log messages about the app as it is running. If you add logging statements to the app, your log messages from these statements appear with the other log messages under this tab.
- The **Monitors** tab. The **Monitors** tab monitors the performance of the app, which can be helpful for debugging and tuning your code.

Adding logging statements to your app

Logging statements add whatever messages you specify to the log. Adding logging statements at certain points in the code allows the developer to look at values, execution paths, and exceptions.

For example, the following logging statement adds `"MainActivity"` and `"Hello World"` to the log:

```
Log.d("MainActivity", "Hello World");
```

The following are the elements of this statement:

- `Log`: The Log class is the API for sending log messages.
- `d`: You assign a *log level* so that you can filter the log messages using the drop-down menu in the center of the **logcat** tab pane. The following are log levels you can assign:
  - `d`: Choose **Debug** or **Verbose** to see these messages.
  - `e`: Choose **Error** or **Verbose** to see these messages.
  - `w`: Choose **Warning** or **Verbose** to see these messages.
  - `i`: Choose **Info** or **Verbose** to see these messages.
- `"MainActivity"`: The first argument is a *log tag* which can be used to filter messages under the **logcat** tab. This is commonly

the name of the activity from which the message originates. However, you can make this anything that is useful to you for debugging the app. The best practice is to use a constant as a log tag, as follows:

1. Define the log tag as a constant before using it in logging statement:

```
private static final String LOG_TAG =
     MainActivity.class.getSimpleName();
```

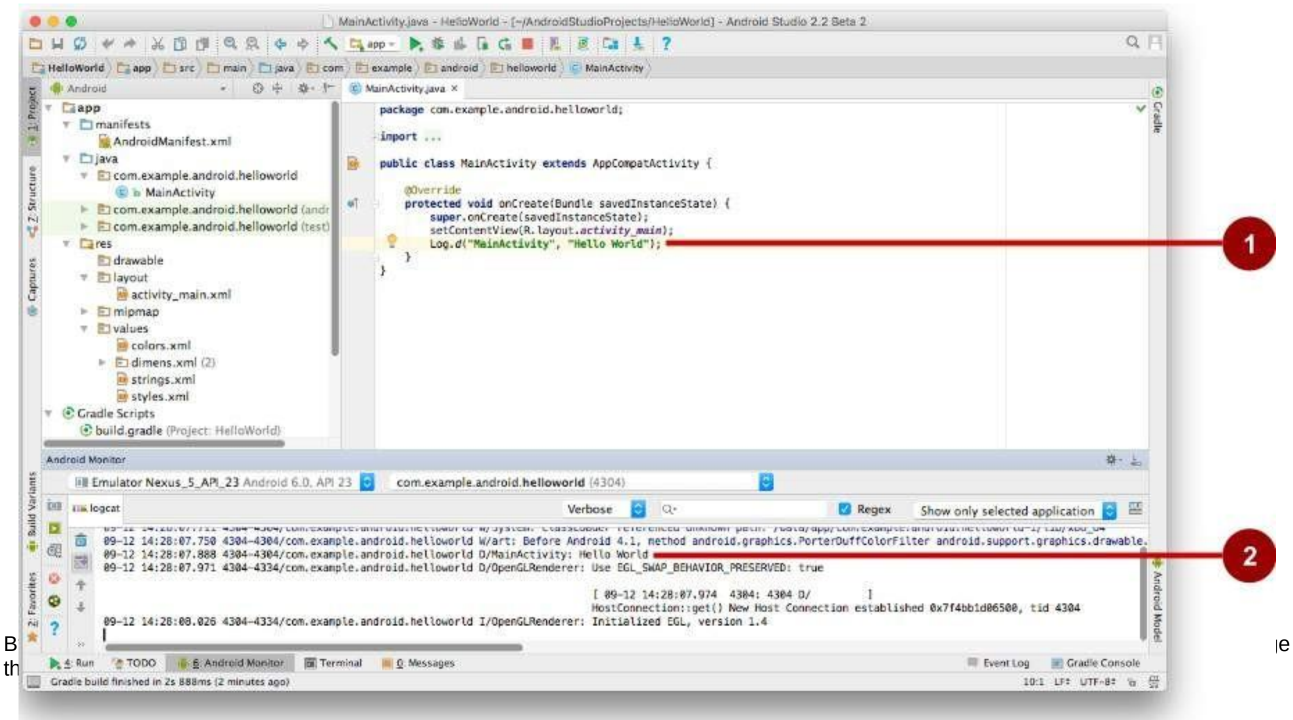
2. Use the constant in the logging statements:

```
Log.d(LOG_TAG, "Hello World");
```

3. `"Hello World"`: The second argument is the actual message that appears after the log level and log tag under the **logcat** tab.

Viewing your log messages

When you run the app on a device or an emulator, the Run pane takes the place of the Android Monitor pane. Once the app has begun to run, click the Android Monitor button at the bottom of the main window. If the logcat tab is not already selected, click it.



```
package com.example.android.helloworld;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d("MainActivity", "Hello World");
    }
}
```

```
09-12 14:28:07.750 4304-4304/com.example.android.helloworld W/art: Before Android 4.1, method android.graphics.PorterDuffColorFilter android.support.graphics.drawable.
09-12 14:28:07.888 4304-4304/com.example.android.helloworld D/MainActivity: Hello World
09-12 14:28:07.971 4304-4334/com.example.android.helloworld D/OpenGLRenderer: Use EGL_SWAP_BEHAVIOR_PRESERVED: true
                                                                              [ 09-12 14:28:07.974  4304: 4304 D/         ]
                                                                              HostConnection::get() New Host Connection established 0x7f4bb1d06500, tid 4304
09-12 14:28:08.026 4304-4334/com.example.android.helloworld I/OpenGLRenderer: Initialized EGL, version 1.4
```

MainActivity.java - HelloWorld - [~/AndroidStudioProjects/HelloWorld] - Android Studio 2.2 Beta 2
Android Monitor
Emulator Nexus_5_API_23 Android 6.0, API 23
com.example.android.helloworld (4304)
logcat
Verbose
Regex
Show only selected application
1
2

1.0: Layouts, Views and Resources

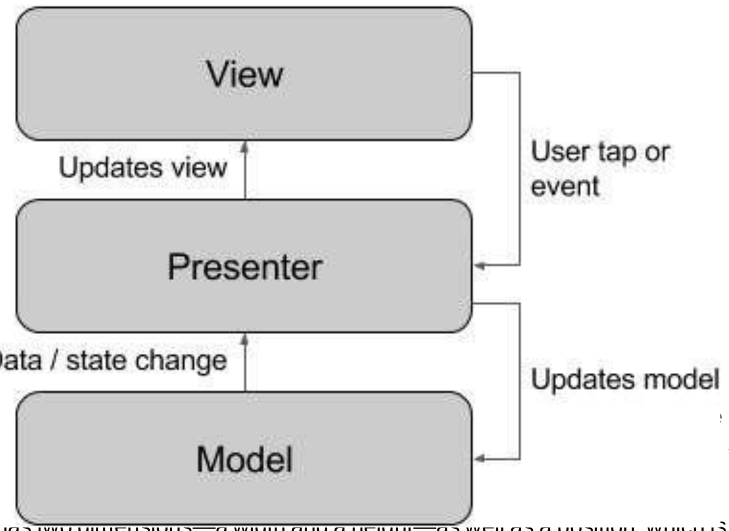
Contents:

This chapter goes over the code you would use to react to a user tapping a user interface element, as well as other resources you generate for your app and the screen's user interface layout.

The model-view-presenter pattern

Linking an activity to a layout resource is an example of part of the *model-view-presenter* (MVP) architecture pattern. The MVP pattern is a well-established way to group app functions:

- **Views.** Views are components of user interfaces that show data and react to human input. Each component of the screen represents a vision. The Android operating system offers a wide variety of viewpoints.
- **Presenters.** Presenters link the model and views of the application. They both give the model with user input from the view and the data that the model specifies for the views.
- **Model.** The model outlines the data's structure as well as the code needed to access and work with it. Some of the apps you develop during the lessons integrate models for data access. Although the Hello Toast app doesn't employ a data model, its logic, which involves displaying a message and increasing a tap counter, can be thought of as the model.



Vi

The screen is a *view*. The View class represents the b that provide interactive UI components such as b

A view has two dimensions—a width and a height—as well as a position, which is represented by a pair of left and top coordinates.
The device-independent pixel (dp) serves as the measurement unit for both position and size.
The Android operating system offers thousands of preconfigured views, some of which show:

- Text (TextView)
- Fields for entering and editing text (EditText)
- Buttons users can tap (Button) and other interactive components
- Scrollable text (ScrollView) and scrollable items (RecyclerView)
- Images (ImageView)

You can define a view to appear on the screen and respond to a user tap. A view can also be defined to accept text input, or to be invisible until needed.

You can specify the views in XML layout resource files. Layout resources are written in XML and listed within the **layout** folder in the **res** folder in the Project: Android view.

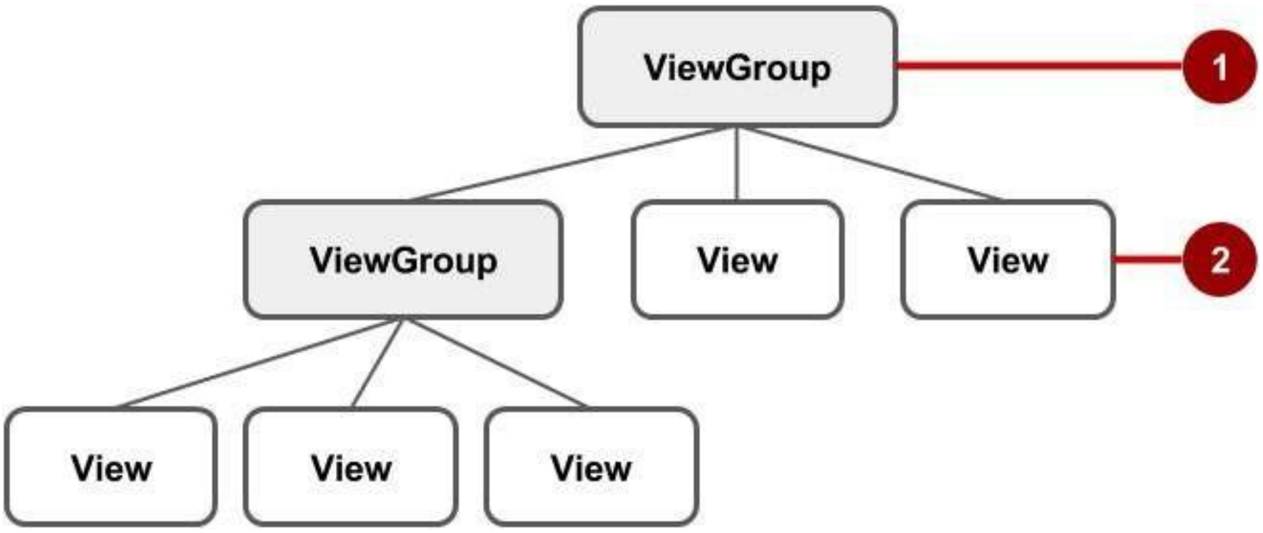
View groups

Views can be grouped together inside a *view group* (ViewGroup), which acts as a container of views. The relationship is parent-child, in which the *parent* is a view group, and the *child* is a view or view group within the group. The following are common view groups:

- ScrollView: a group that allows the child view to be scrolled and contains just one other child view.
- RecyclerView: a group that allows scrolling by adding and removing views dynamically from the screen and contains a list of other views or view groups.

Layout view groups

The views for a screen are organized in a hierarchy. At the *root* of this hierarchy is a ViewGroup that contains the layout of the entire screen. The view group's child screens can be other views or other view groups as shown in the following figure.

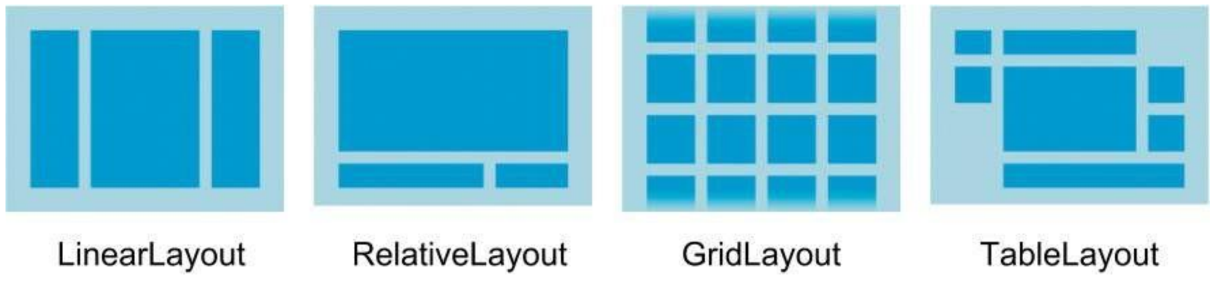


In the above figure:

1. The *root* view group.
2. The first set of child views and view groups whose parent is the root.
3. Because they arrange child views in a specified fashion and are frequently used as the root view group, some view groups are referred to as layouts. Examples of layouts include:

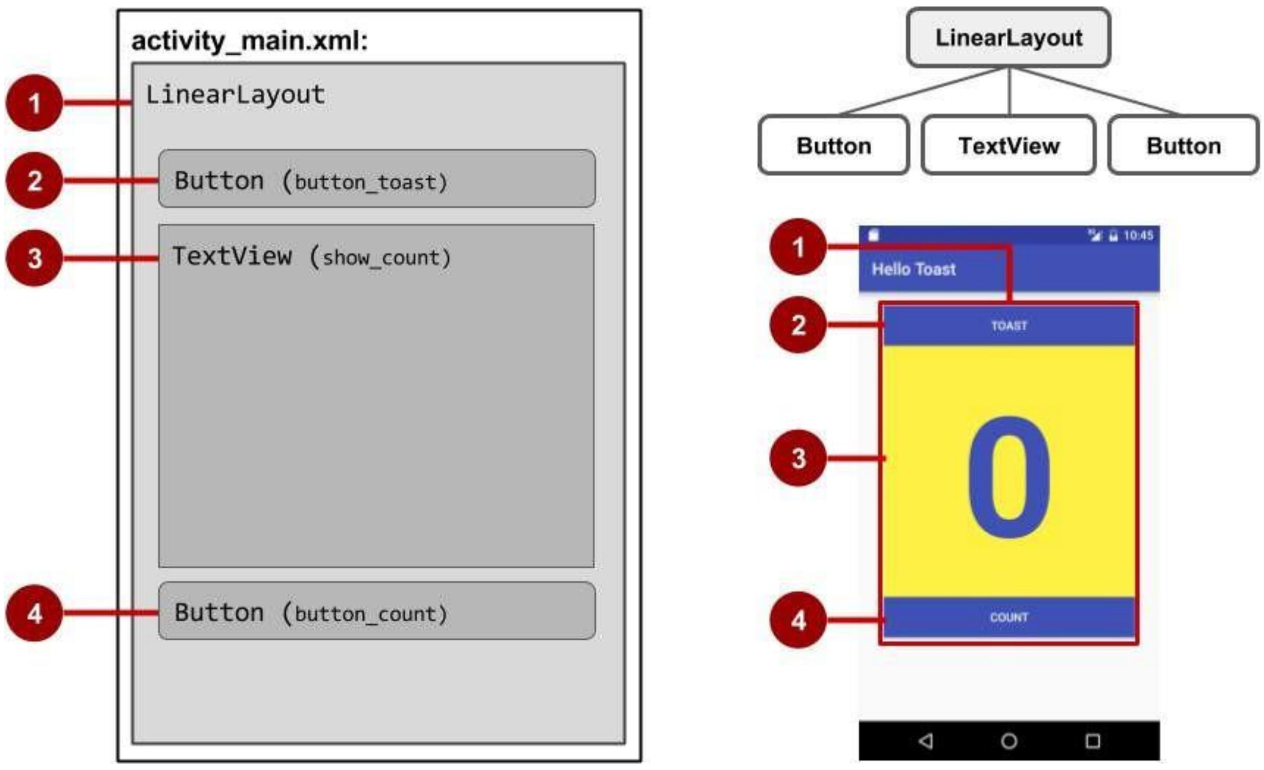
- LinearLayout: A group of child views positioned and aligned horizontally or vertically.
- RelativeLayout: A group of child views in which each view is positioned and aligned relative to other views within the view group. In other words, the positions of the child views can be described in relation to each other or to the parent view group.
- ConstraintLayout: A group of child views using anchor points, edges, and guidelines to control how views are positioned relative to other elements in the layout. ConstraintLayout was designed to make it easy to drag and drop views in the layout editor.

- TableLayout: A group of child views arranged into rows and columns.
- AbsoluteLayout: A group that lets you specify exact locations (x/y coordinates) of its child views. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.
- FrameLayout: A group of child views in a stack. FrameLayout is designed to block out an area on the screen to display one view. Child views are drawn in a stack, with the most recently added child on top. The size of the FrameLayout is the size of its largest child view.
- GridLayout: A group that places its child screens in a rectangular grid that can be scrolled.



Tip: Learn more about different layout types in Common Layout Objects.

A simple example of a layout with child views is the Hello Toast app in one of the early lessons. The view for the Hello Toast app appears in the figure below as a diagram of the layout file (activity_main.xml), along with a hierarchy diagram (top right) and a screenshot of the actual finished layout (bottom right).



In the figure above:

1. LinearLayout root layout, which contains all the child views, set to a vertical orientation.
2. Button (button_toast) child view. As the first child view, it appears at the top in the linear layout.
3. TextView (show_count) child view. As the second child view, it appears under the first child view in the linear layout.
4. Button (button_count) child view. As the third child view, it appears under the second child view in the linear layout.

The view hierarchy can grow to be complex for an app that shows many views on a screen. It's important to understand the view hierarchy, as it affects whether views are visible and efficiently they are drawn.

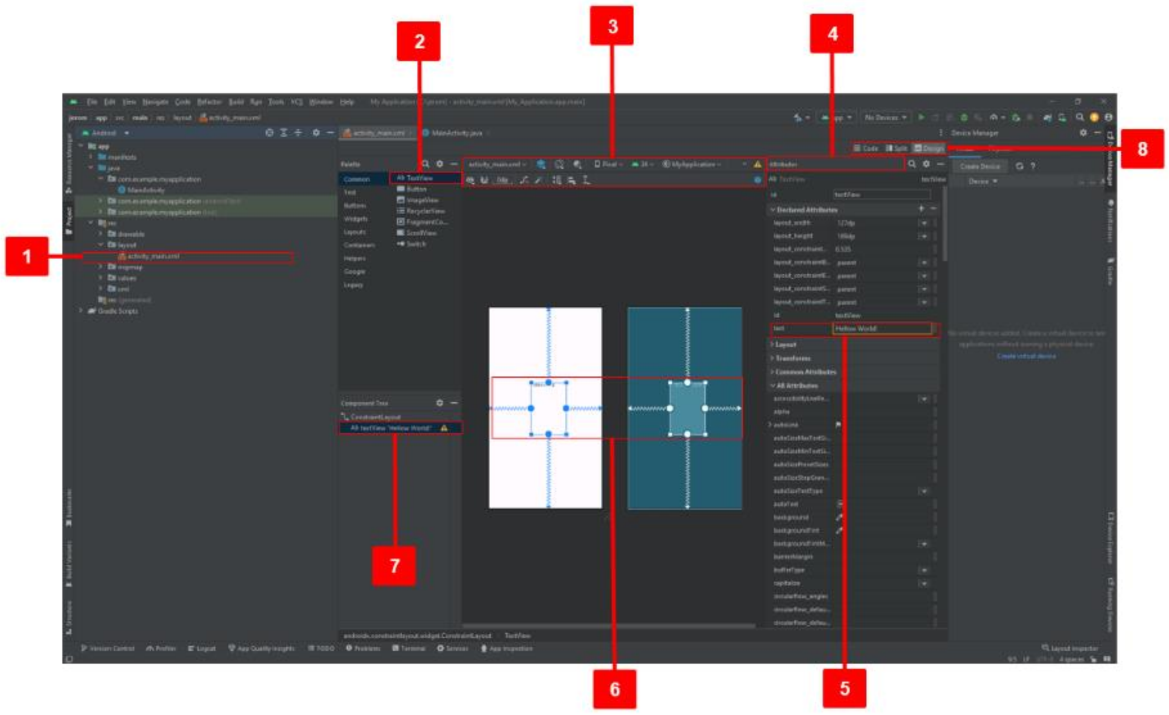
Tip: You can explore the view hierarchy of your app using Hierarchy Viewer. It shows a tree view of the hierarchy and lets you analyze the performance of views on an Android device. Performance issues are covered in a subsequent chapter.

You define views in the layout editor, or by entering XML code. The layout editor shows a visual representation of XML code.
e

Using the layout editor

Use the layout editor to edit the layout file. You can drag and drop view objects into a graphical pane, and arrange, resize, and specify properties for them. You immediately see the effect of changes you make.

To use the layout editor, open the XML layout file. The layout editor appears with the **Design** tab at the bottom highlighted. (If the **Text** tab is highlighted and you see XML code, click the **Design** tab.) For the Empty Activity template, the layout is as shown in the figure below.



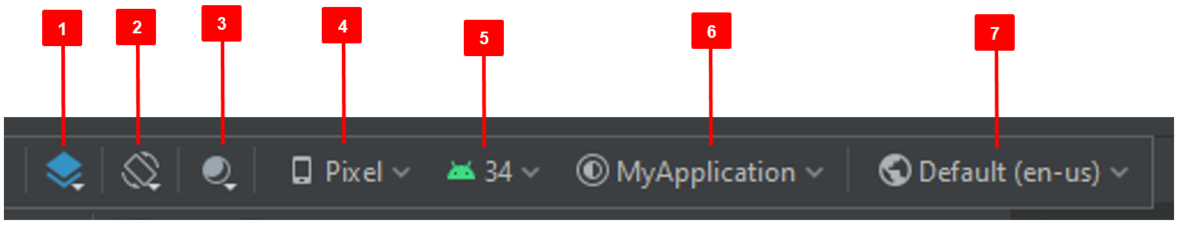
In the figure above:

1. **XML layout** file. The XML layout file, typically named **activiy_main.xml** file. Double-click it to open the layout editor.
2. **Palette of UI elements** (views). The Palette pane provides a list of UI elements and layouts. Add an element or layout to the UI by dragging it into the design pane.
3. **Design toolbar**. The design pane toolbar provides buttons to configure your layout appearance in the design pane and to edit the layout properties. See the figure below for details.

   Tip: Hover over each icon to view a tooltip that summarizes its function.

4. **Attributes pane**. The Attributes pane provides property controls for the selected view.
5. **Attribute control**. Attribute controls correspond to XML attributes. Shown in the figure is the `Text` property of the selected TextView, set to `Hello World!`.
6. **Design pane**. Drag views from the Palette pane to the design pane to position them in the layout.
7. **Component Tree**. The Component Tree pane shows the view hierarchy. Click a view or view group in this pane to select it. The figure shows the TextView selected.
8. **Design, Split,** and **Code** tabs. Click **Design** to see the layout editor, or **Code** to see XML code or click the **Split** to see both at the same time.

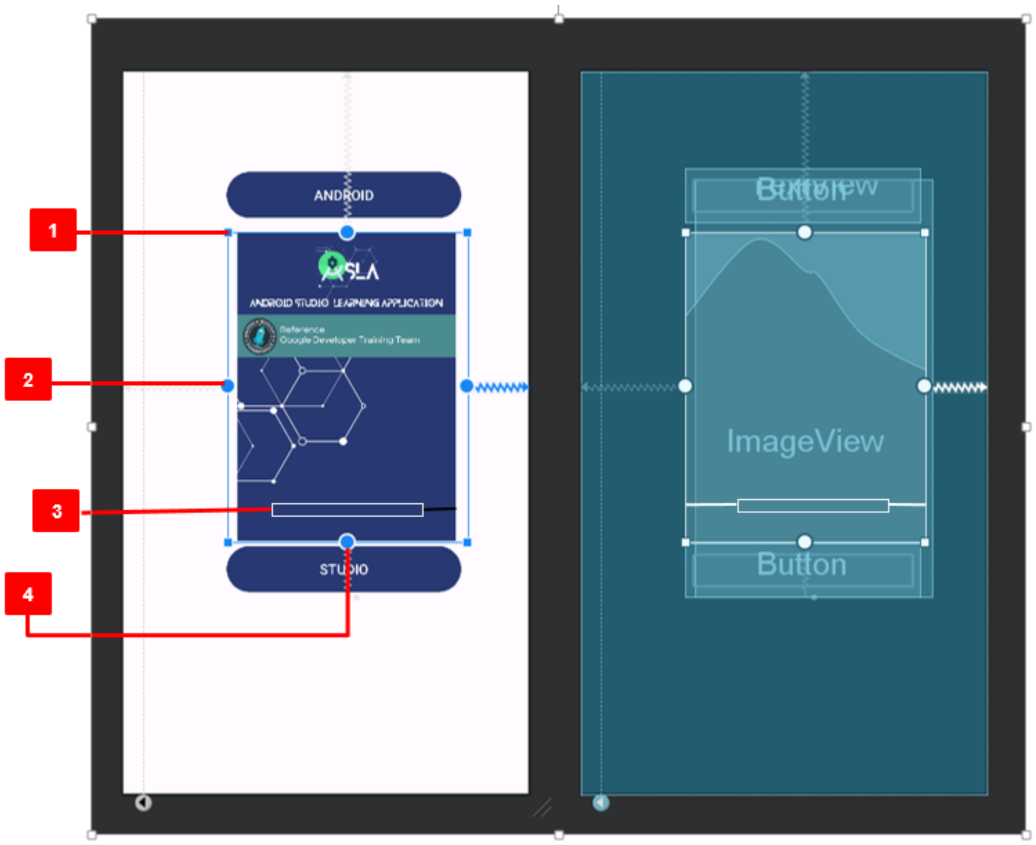
The layout editor's design toolbar offers a row of buttons that let you configure the appearance of the layout:



In the figure above:

1. **Design**, **Blueprint**, and **Both**: Click the **Design** icon (first icon) to display a color preview of your layout. Click the **Blueprint** icon (middle icon) to show only outlines for each view. You can see *both* views side by side by clicking the third icon.
2. **Screen orientation**: Click to rotate the device between landscape and portrait.
3. **App theme**: Select which UI theme to apply to the preview.
4. **Device type and size:** Select the device type (phone/tablet, Android TV, or Android Wear) and screen configuration (size and density).
5. **API version:** Select the version of Android on which to preview the layout.
6. **Layout Variants:** Switch to one of the alternative layouts for this file, or create a new one.
7. **Language**: Select the language to show for your UI strings. This list displays only the languages available in the string resources.

When you use a ConstraintLayout, the layout editor provides more options under the **Design** tab, such as handles for defining constraints. A connection or alignment to another view, the parent layout, or an invisible guideline is referred to as a constraint. Each restriction is represented by a line emanating from a handle that is circular. There is a circular constraint handle in the center of each side of each view. The view additionally displays resizing handles on each corner when selecting a view in the Component Tree pane or clicking on it in the layout.



In the above figure:

1. **Resizing handle.**
2. **Constraint line and handle**. In the figure, the constraint aligns the left side of the view to the left side of the button.
3. **Baseline handle**. The baseline handle aligns the text baseline of a view to the text baseline of another view.
4. **Constraint handle** without a constraint line.

Using XML

It is sometimes quicker and easier to edit the XML code directly, especially when copying and pasting the code for similar views.

To view and edit the XML code, open the XML layout file. The layout editor appears with the **Design** tab at the bottom highlighted. Click the **Text** tab to see the XML code. The following shows an XML code snippet of a LinearLayout with a Button and a TextView:

```
<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="match_parent"
    ... >

    <Button
        android:id="@+id/button_toast"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="wrap_content"
        ... />

    <TextView
        android:id="@+id/show_count"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="@dimen/counter_height"
        ... />
    ...
</LinearLayout>
```

XML attributes (view properties)

Views have *properties* that define where a view appears on the screen, its size, how the view relates to other views, and how it responds to user input. When defining views in XML, the properties are referred to as *attributes*.

For example, in the following XML description of a TextView, the `android:id`, `android:layout_width`, `android:layout_height`, `android:background`, are XML attributes that are translated automatically into the TextView's properties:

```
<TextView
       android:id="@+id/show_count"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:background="@color/myBackgroundColor"
       android:textStyle="bold"
       android:text="@string/count_initial_value"
/>
```

Attributes generally take this form:

```
android:attribute_name="value"
```

The *attribute_name* is the name of the attribute. The *value* is a string with the value for the attribute. For example:

```
android:textStyle="bold"
```

If the *value* is a resource, such as a color, the `@` symbol specifies what kind of resource. For example:

```
android:background="@color/myBackgroundColor"
```

The background attribute is set to the color resource identified as `myBackgroundColor`, which is declared to be `#FFF043`. Color resources are described in "Style-related attributes" in this chapter.

Every view and view group supports its own variety of XML attributes. Some attributes are specific to a view (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any views that may extend the TextView class. Some are common to all views, because they are inherited from the root View class (like the `android:id` attribute). For descriptions of specific attributes, see the overview section of the View class documentation.

Identifying a view

To uniquely identify a view and reference it from your code, you must give it an id. The `android:id` attribute lets you specify a unique `id` — a resource identifier for a view.

For example:

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

The `"@+id/button_count"` part of the above attribute creates a new `id` called `button_count` for the view. You use the plus (`+`) symbol to indicate that you are creating a new `id`.

Referencing a view

To refer to an existing resource identifier, omit the plus (`+`) symbol. For example, to refer to a view by its `id` in *another* attribute, such as `android:layout_toLeftOf` (described in the next section) to control the position of a view, you would use:

```
android:layout_toLeftOf="@id/show_count"
```

In the above attribute, `"@id/show_count"` refers to the view with the resource identifier `show_count`. The attribute positions the view to be "to the left of" the `show_count` view.

Positioning views

Some layout-related positioning attributes are required for a view, and automatically appear when you add the view to the XML layout, ready for you to add values.

LinearLayout positioning

For example, LinearLayout is required to have these attributes set:

- android:layout_width
- android:layout_height

  android:orientation

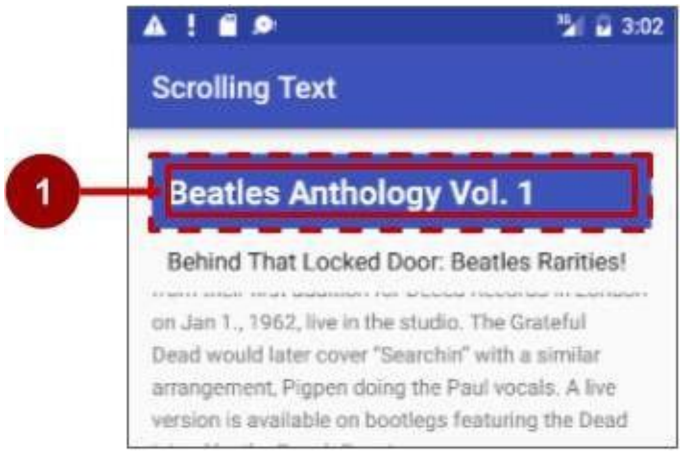
The `android:layout_width` and `android:layout_height` attributes can take one of three values:

  `match_parent` expands the view to fill its parent by width or height. When the LinearLayout is the root view, it expands to the size of the device screen. For a view within a root view group, it expands to the size of the parent view group.
  `wrap_content` shrinks the view dimensions just big enough to enclose its content. (If there is no content, the view becomes invisible.)
  Use a fixed number of `dp` (device-independent pixels) to specify a fixed size, adjusted for the screen size of the device. For example, `16dp` means 16 device-independent pixels. Device-independent pixels and other dimensions are described in "Dimensions" in this chapter.

The `android:orientation` can be:

horizontal: Views are arranged from left to right.
vertical: Views are arranged from top to bottom.

- Padding is the space, measured in device-independent pixels, between the edges of the view and the view's content, as shown in the figure below.


3:02
Scrolling Text
1
Beatles Anthology Vol. 1
Behind That Locked Door: Beatles Rarities!
on Jan 1., 1962, live in the studio. The Grateful
Dead would later cover "Searchin" with a similar
arrangement, Pigpen doing the Paul vocals. A live
version is available on bootlegs featuring the Dead

In the figure above:

1. Padding is the space between the edges of the view (dashed lines) and the view's content (solid line). Padding is not the same as margin, which is the space from the edge of the view to its parent.

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

- `Android:padding`: Sets the padding of all four edges.
- `android:paddingTop`: Sets the padding of the top edge.
- `android:paddingBottom`: Sets the padding of the bottom edge.
- `android:paddingLeft`: Sets the padding of the left edge.
- `android:paddingRight`: Sets the padding of the right edge.
- `android:paddingStart`: Sets the padding of the start of the view; used in place of the above, especially with views that are long and narrow.
- `android:paddingEnd`: Sets the padding, in pixels, of the end edge; used along with `android:paddingStart`.

Tip: To see all of the XML attributes for a LinearLayout, see the Summary section of the LinearLayout reference in the Developer Guide. Other root layouts, such as RelativeLayout and AbsoluteLayout, list their XML attributes in the Summary sections.

RelativeLayout Positioning

Another useful view group for layout is RelativeLayout, which you can use to position child views relative to each other or to the parent. The attributes you can use with RelativeLayout include the following:

android:layout_toLeftOf: Positions the right edge of this view to the left of another view (identified by its `ID`).
android:layout_toRightOf: Positions the left edge of this view to the right of another view (identified by its `ID`).
android:layout_centerHorizontal: Centers this view horizontally within its parent.
android:layout_centerVertical: Centers this view vertically within its parent.
android:layout_alignParentTop: Positions the top edge of this view to match the top edge of the parent.
android:layout_alignParentBottom: Positions the bottom edge of this view to match the bottom edge of the parent.

For a complete list of attributes for views in a RelativeLayout, see RelativeLayout.LayoutParams

Style-related attributes

You specify style attributes to customize the view's appearance. Views that *don't* have style attributes, such as `android:textColor`, `android:textSize`, and `android:background`, take on the styles defined in the app's theme.

The following are style-related attributes used in the XML layout example in the previous section:

- `Android:background`: Specifies a color or drawable resource to use as the background.
- `android:text`: Specifies text to display in the view.
- `android:textColor`: Specifies the text color.
- `android:textSize`: Specifies the text size.
- `android:textStyle`: Specifies the text style, such as `bold`.

Resource files

Resource files are a way of separating static values from code so that you don't have to change the code itself to change the values. You can store all the strings, layouts, dimensions, colors, styles, and menu text separately in resource files.

Resource files are stored in folders located in the **res** folder, including:

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

The syntax to reference a resource in an XML layout is as follows:

```
@package_name:resource_type/resource_name
```

- The *package_name* is the name of the package in which the resource is located. This is not required when referencing resources from the same package — that is, stored in the **res** folder of your project.
- *resource_type* is the `R` subclass for the resource type. See Resource Types for more information about each resource type and how to reference them.
- *resource_name* is either the resource filename without the extension, or the `android:name` attribute value in the XML element.

For example, the following XML layout statement sets the `android:text` attribute to a `string` resource:

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

- The *resource_type* is `string`.
- The resource_name is `button_label_toast.`
- There is no need for a *package_name* because the resource is stored in the project (in the strings.xml file).

Another example: this XML layout statement sets the `android:background` attribute to a `color` resource, and since the resource is defined in the project (in the colors.xml file), the *package_name* is not specified:

```
android:background="@color/colorPrimary"
```

In the following example, the XML layout statement sets the `android:textColor` attribute to a `color` resource. However, the resource is not defined in the project but supplied by Android, so the *package_name* `android` must also be specified, followed by a colon:

```
android:textColor="@android:color/white"
```

Tip: For more information about accessing resources from code, see Accessing Resources. For Android color constants, see the Android standard R.color resources.

Values resource files

Keeping values such as strings and colors in separate resource files makes it easier to manage them, especially if you use them more than once in your layouts.

For example, it is essential to keep strings in a separate resource file for translating and localizing your app, so that you can create a string resource file for each language without changing your code. Resource files for images, colors, dimensions, and other attributes are handy for developing an app for different device screen sizes and orientations.

Strings

String resources are located in the **strings.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly by opening it:

```
<resources>
    <string name="app_name">Hello Toast</string>
    <string name="button_label_count">Count</string>
    <string name="button_label_toast">Toast</string>
    <string name="count_initial_value">0</string>
</resources>
```

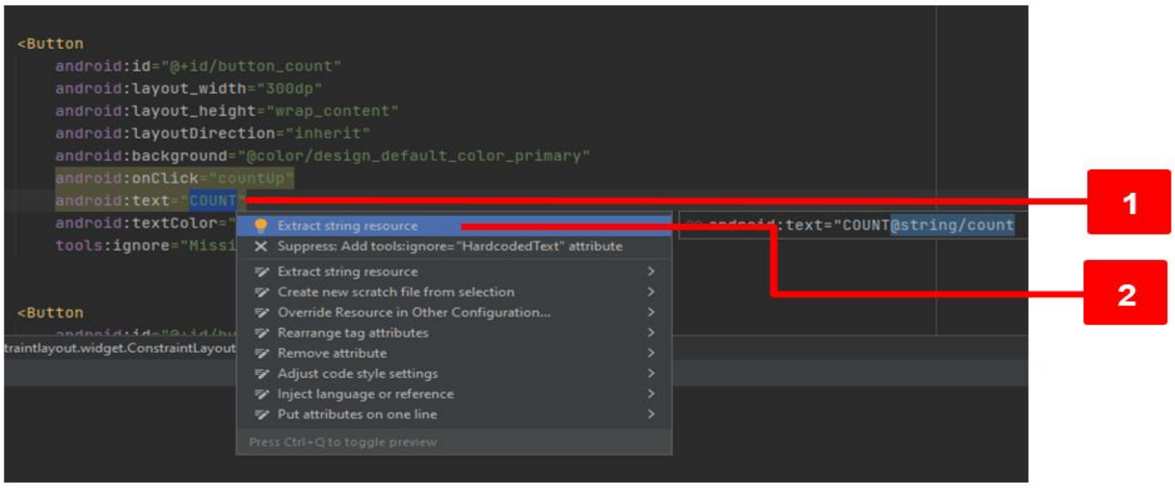
The `name` (for example, `button_label_count`) is the resource name you use in your XML code, as in the following attribute:

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

The string value of this `name` is the word (`Count`) enclosed within the `<string></string>` tags (you don't use quotation marks unless the quotation marks should be part of the string value.)

Extracting strings to resources

You should also *extract* hard-coded strings in an XML layout file to string resources. To extract a hard-coded string in an XML layout, follow these steps (refer to the figure):

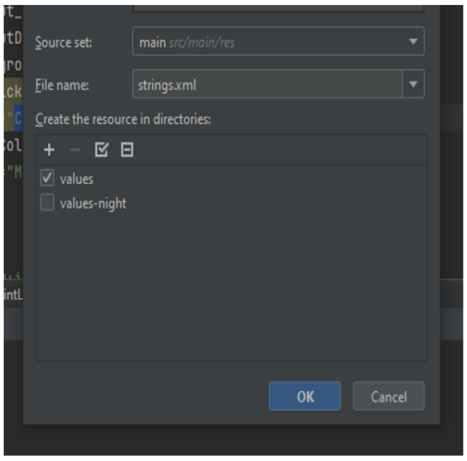


```
<Button
    android:id="@+id/button_count"
    android:layout_width="300dp"
    android:layout_height="wrap_content"
    android:layoutDirection="inherit"
    android:background="@color/design_default_color_primary"
    android:onClick="countUp"
    android:text="COUNT"
    android:textColor="
    tools:ignore="Missi

<Button
```

```
android:text="COUNT@string/count
```

1
2
Extract string resource
Suppress: Add tools:ignore="HardcodedText" attribute
Extract string resource
Create new scratch file from selection
Override Resource in Other Configuration...
Rearrange tag attributes
Remove attribute
Adjust code style settings
Inject language or reference
Put attributes on one line
Press Ctrl+Q to toggle preview
traintlayout.widget.ConstraintLayout



Source set: main src/main/res

File name: strings.xml

Create the resource in directories:

- [✓] values
- [ ] values-night

1. Click on the hard-coded string, and press Alt-Enter in Windows, or Option-Return on Mac OS X.
2. Select **Extract string resource**.
3. Edit the Resource name for the string value.

You can then use the resource name in your XML code. Use the expression `"@string/resource_name"` (including quotation marks) to refer to the string resource:

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

Colors

Color resources are located in the **colors.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly:

```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
    <color name="myBackgroundColor">#FFF043</color>
</resources>
```

The `name` (for example, `colorPrimary`) is the resource name you use in your XML code:

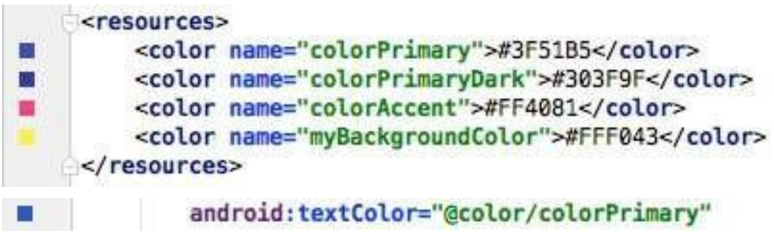
```
android:textColor="@color/colorPrimary"
```

The color value of this `name` is the hexadecimal color value (`#3F51B5`) enclosed within the `<color></color>` tags. The hexadecimal value specifies red, green, and blue (RGB) values. The value always begins with a pound (`#`) character, followed by the Alpha-Red-Green-Blue information. For example, the hexadecimal value for black is #000000, while the hexadecimal value for a variant of sky blue is #559fe3. Base color values are listed in the Color class documentation.

The `colorPrimary` color is one of the predefined base colors and is used for the app bar. In a production app, you could, for example, customize this to fit your brand. Using the base colors for other UI elements creates a uniform UI.

Tip: For the material design specification for Android colors, see Style and Using the Material Theme. For common color hexadecimal values, see Color Hex Color Codes. For Android color constants, see the Android standard R.color resources.

You can see a small block of the color choice in the left margin next to the color resource declaration in **colors.xml**, and also in the left margin next to the attribute that uses the resource name in the layout XML file.



```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
    <color name="myBackgroundColor">#FFF043</color>
</resources>

        android:textColor="@color/colorPrimary"
```

Tip: To see the color in a popup, turn on the Autopopup documentation feature. Choose **Android Studio > Preferences > Editor > General > Code Completion**, and check the "Autopopup documentation in (ms)" option. You can then hover your cursor over a color resource name to see the color.

Dimensions

Dimensions should be separated from the code to make them easier to manage, especially if you need to adjust your layout for different device resolutions. It also makes it easy to have consistent sizing for views, and to change the size of multiple views by changing one dimension resource.

Dimension resources are located in a **dimens.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. The **dimens.xml** shown in the view can be a folder holding more than one **dimens.xml** file for different device resolutions. For example, the app created from the Empty Activity template provides a second **dimens.xml** file for 820dp.

You can edit this file directly by opening it:

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
    <dimen name="my_view_width">300dp</dimen>
    <dimen name="count_text_size">200sp</dimen>
    <dimen name="counter_height">300dp</dimen>
</resources>
```

The `name` (for example, `activity_horizontal_margin`) is the resource name you use in the XML code:

```
android:paddingLeft="@dimen/activity_horizontal_margin"
```

The value of this `name` is the measurement (`16dp`) enclosed within the `<dimen></dimen>` tags. You

can extract dimensions in the same way as strings::

1. Click on the hard-coded dimension, and press Alt-Enter in Windows, or press Option-Return on Mac OS X.
2. Select **Extract dimension resource**.
3. Edit the Resource name for the dimension value.

Device-independent pixels (`dp`) are independent of screen resolution. For example, `10px` (10 fixed pixels) will look a lot smaller on a higher resolution screen, but Android will scale 1 `0dp` (10 device-independent pixels) to look right on different resolution screens. Text sizes can also be set to look right on different resolution screens using *scaled-pixel* (`sp`) sizes.

Tip: For more information about `dp` and `sp` units, see Supporting Different Densities.

Styles

A style is a resource that specifies common attributes such as height, padding, font color, font size, background color. Styles are meant for attributes that modify the look of the view.

Styles are defined in the **styles.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly. Styles are covered in a later chapter, along with the Material Design Specification.

Other resource files

Android Studio defines other resources that are covered in other chapters:

- **Images and icons**. The **drawable** folder provides icon and image resources. If your app does not have a drawable folder, you can manually create it inside the res folder. For more information about drawable resources, see Drawable Resources in the App Resources section of the Android Developer Guide.
- **Optimized icons**. The **mipmap** folder typically contains pre-calculated, optimized collections of app icons used by the Launcher. Expand the folder to see that versions of icons are stored as resources for different screen densities.
- **Menus**. You can use an XML resource file to define menu items and store them in your project in the **menu** folder. Menus are described in a later chapter.

Responding to view clicks

A *click event* occurs when the user taps or clicks a clickable view, such as a Button, ImageButton, ImageView (tapping or clicking the image), or FloatingActionButton.

The model-view-presenter pattern is useful for understanding how to respond to view clicks. When an event occurs with a view, the *presenter* code performs an action that affects the *model* code. In order to make this pattern work, you have to:

- Write a Java method that performs the specific action, which is determined by the logic of the *model* code — that is, the action depends on what you want the app to do when this event occurs. This is typically referred to as an *event handler*.
- Associate this event handler method to the *view*, so that the method executes when the event occurs.

The onClick attribute

Android Studio provides a shortcut for setting up a clickable view, and for associating an event handler with the view: use the `android:onClick` attribute with the clickable view's element in the XML layout.

For example, the following XML expression in the layout file for a Button sets `showToast()` as the event handler:

```
android:onClick="showToast"
```

When the `b`utton is tapped, its `android:onClick` attribute calls the `showToast()` method.

Write the event handler, such as `showToast()` referenced in the XML code above, to call other methods that implement the app's *model* logic:

```
public void showToast(View view) {
        // Do something in response to the button click.
}
```

In order to work with the `android:onClick` attribute, the `showToast()` method must be `public`, return `void`, and require a `view` parameter in order to know which view called the method.

Updating views

To update a view's contents, such as replacing the text in a TextView, your code must first instantiate an object from the view. Your code can then update the object, thereby updating the view.

To refer to the view in your code, use the findViewById() method of the View class, which looks for a view based on the resource `id`. For example, the following statement sets `mShowCount` to be the TextView with the resource id `show_count`:

```
mShowCount = (TextView) findViewById(R.id.show_count);
```

From this point on, your code can use `mShowCount` to represent the TextView, so that when you update `mShowCount`, the view is updated.

For example, when the following button with the `android:onClick` attribute is tapped, onClick calls the `countUp()` method:

```
android:onClick="countUp"
```

You can implement `countUp()` to increment the count, convert the count to a string, and set the string as the text for the `mShowCount` object:

```
public void countUp(View view) {
        mCount++;
        if (mShowCount != null)
            mShowCount.setText(Integer.toString(mCount));
}
```

Since you had already associated `mShowCount` with the TextView for displaying the count, the `mShowCount.setText()` method updates the text view on the screen.

1.2: Text and Scrolling Views

Contents:

This chapter describes one of the most often used views in apps: the TextView, which shows textual content on the screen. A TextView can be used to show a message, a response from a database, or even entire magazine-style articles that users can scroll. This chapter also shows how you can create a scrolling view of text and other elements.

TextView

One view you may use often is the TextView class, which is a subclass of the View class that displays text on the screen. You can use TextView for a view of any size, from a single character or word to a full screen of text. You can add a resource `id` to the TextView, and control how the text appears using attributes in the XML layout file.

You can refer to a TextView view in your Java code by using its resource `id`, and update the text from your code. If you want to allow users to edit the text, use EditText, a subclass of TextView that allows text input and editing. You learn all about EditText in another chapter.

TextView attributes

You can use XML attributes to control:

- Where the TextView is positioned in a layout (like any other view)
- How the view itself appears, such as with a background color

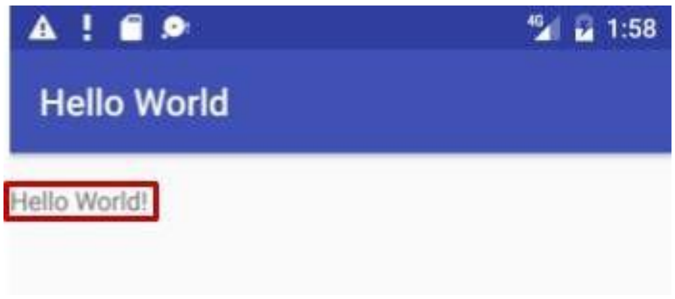
- What the text looks like within the view, such as the initial text and its style, size, and color

For example, to set the width, height, and position within a LinearLayout:

```
<TextView
   ...
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   … />
```

To set the initial text value of the view, use the `android:text` attribute:

```
android:text="Hello World!"
```



You can extract the text string into a string resource (perhaps called `hello_world`) that's easier to maintain for multiple-language versions of the app, or if you need to change the string in the future. After extracting the string, use the string resource name with `@string/` to specify the text:

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

The most often used attributes with TextView are the following:

- android:text: Set the text to display.
- android:textColor: Set the color of the text. You can set the attribute to a color value, a predefined resource, or a theme. Color resources and themes are described in other chapters.
- android:textAppearance: The appearance of the text, including its color, typeface, style, and size. You set this attribute to a predefined style resource or theme that already defines these values.
- android:textSize: Set the text size (if not already set by `android:textAppearance`). Use `sp` (scaled-pixel) sizes such as `20sp` or `14.5sp`, or set the attribute to a predefined resource or theme.
- android:textStyle: Set the text style (if not already set by `android:textAppearance`). Use `normal`, `bold`, `italic`, or `bold` | `italic`.
- Android:typeface: Set the text typeface (if not already set by `android:textAppearance`). Use `normal`, `sans`, `serif`, or `monospace`.
- android:lineSpacingExtra: Set extra spacing between lines of text. Use `sp` (scaled-pixel) or dp (device-independent pixel) sizes, or set the attribute to a predefined resource or theme.
- android:autoLink: Controls whether links such as URLs and email addresses are automatically found and converted to clickable (touchable) links. Use one of the following:
    - `none`: Match no patterns (default).
    - `web`: Match web URLs.
    - `email`: Match email addresses.
    - `phone`: Match phone numbers.
    - `map`: Match map addresses.
    - `all`: Match all patterns (equivalent to web|email|phone|map).

  For example, to set the attribute to match web URLs, use this:

  ```
  android:autoLink="web"
  ```

Using embedded tags in text

Pressing Return will enter hard line endings, or formatting the text first in a text editor with hard line endings, will allow you to see the text wrapped in strings.xml. The screen will not show the conclusions.
In either scenario, the text may have additional text formatting codes or embedded HTML tags. Text must be formatted in accordance with these guidelines to display correctly in a text view:

If you have an apostrophe (') in your text, you must *escape* it by preceding it with a backslash (**\'**). If you have a double-quote in your text, you must also escape it (**\"**). You must also escape any other non-ASCII characters. See the "Formatting and Styling" section of String Resources for more details.
The TextView ignores all HTML tags except the following:
Use the HTML and **</b>** tags around words that should be in bold.
Use the HTML and **</i>** tags around words that should be in italics. Note, however, that if you use curled apostrophes within an italic phrase, you should replace them with straight apostrophes.
You can combine bold and italics by combining the tags, as in ***...*** words...**</i></b>**.

To create a long string of text in the **strings.xml** file, enclose the entire text within `<string name="your_string_name">` `</string>` in the strings.xml file (*your_string_name* is the name you provide the string resource, such as `article_text`).

Text lines in the strings.xml file don't wrap around to the next line — they extend beyond the right margin. This is the correct behavior. Each new line of text starting at the left margin represents an entire paragraph.

Enter **\n** to represent the end of a line, and another **\n** to represent a blank line. If you don't add end-of-line characters, the paragraphs will run into each other when displayed on the screen.

Tip: If you want to see the text wrapped in strings.xml, you can press Return to enter hard line endings, or format the text first in a text editor with hard line endings. The endings will not be displayed on the screen.

Referring to a TextView in code

To refer to a TextView in your Java code, use its resource `id`. For example, to update a TextView with new text, you would:

1. Find the TextView (with the id `show_count`) and assign it to a variable. You use the `findViewById()` method of the View class, and refer to the view you want to find using this format:

```
R.id.view_id
```

In which `view_id` is the resource identifier for the view:

```
mShowCount = (TextView) findViewById(R.id.show_count);
```

2. After retrieving the view as a TextView member variable, you can then set the text of the text view to the new text using the setText() method of the TextView class:

```
mShowCount.setText(mCount_text);
```

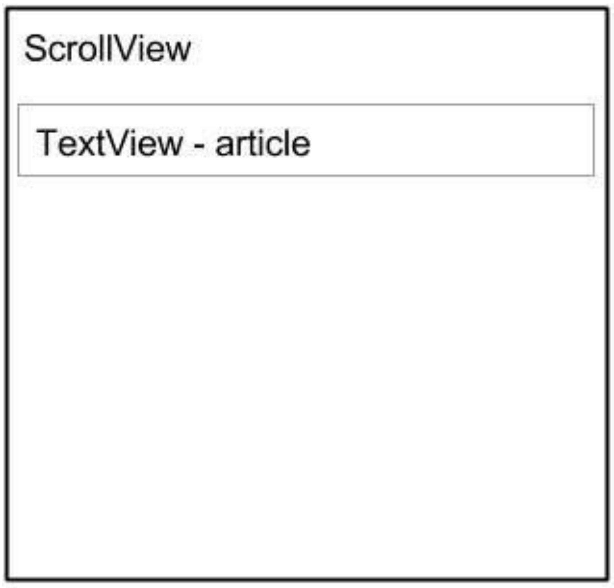
Scrolling views

The user can scroll horizontally or vertically by swiping right or left through a *scrolling view* that you can create if the information you want to display in your app is larger than the device's display.

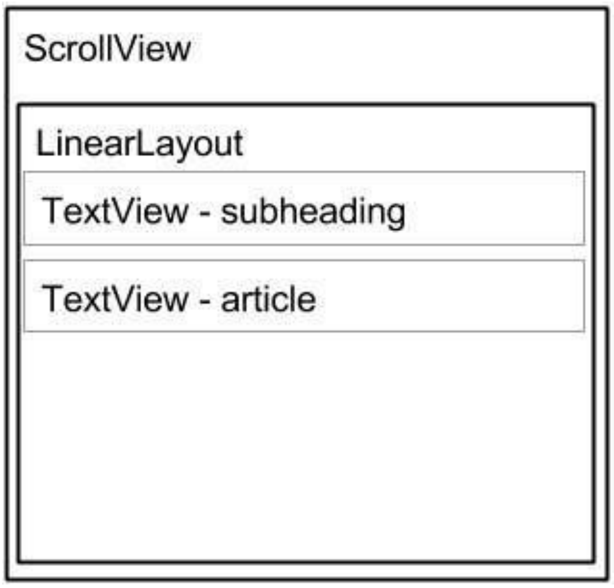
For news articles, books, or any other lengthy text that doesn't completely fit on the screen, you would typically use a scrolling view. A scrolling view can also be used to group views (like a TextView and a Button) together.

Creating a layout with a Scroll View

The ScrollView class provides the layout for a vertical scrolling view. (For horizontal scrolling, you would use HorizontalScrollView.) ScrollView is a subclass of FrameLayout, which means that you can place only *one* view as a child within it; that child contains the entire contents to scroll.



Despite the fact that a ScrollView can only contain one child view, the child view could be a view group that contains a hierarchy of child views, such as a LinearLayout. A vertically oriented LinearLayout is a good option for a view inside of a ScrollView..



ScrollView and performance

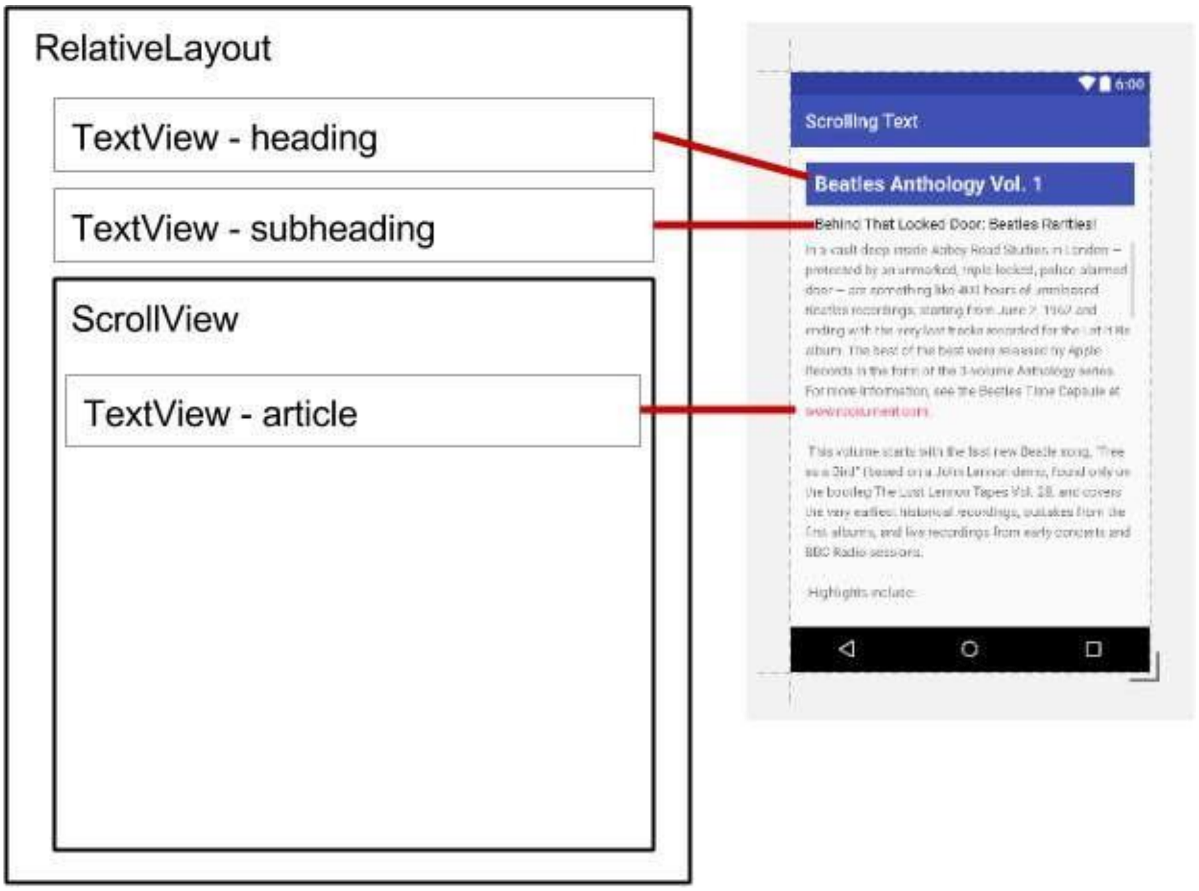
Even if your views aren't visible on the screen with a ScrollView, they are still all stored in memory and in the view hierarchy. Since the text is already in memory, ScrollView can be used to smoothly scroll through pages of free-form text. However, ScrollView can consume a lot of memory, which can hinder how well your app works overall.

Using nested instances of LinearLayout can also lead to an excessively deep view hierarchy, which can slow down performance. Nesting several instances of LinearLayout that use the `android:layout_weight` attribute can be especially expensive as each child view needs to be measured twice. Consider using flatter layouts such as RelativeLayout or GridLayout to improve performance.

Complex layouts with ScrollView may suffer performance issues, especially with child views such as images. We recommend that you *not* use images with a ScrollView. To display long lists of items, or images, consider using a RecyclerView. Also, using AsyncTask provides a simple way to perform work outside the main thread, such as loading images in a background thread, then applying them to the UI once finished. AsyncTask is covered in another chapter.

ScrollView with a TextView

Use a RelativeLayout for the screen that has a separate TextView for the article heading, another for the article subheading, and a third TextView for the scrolling article text (see figure below) if you want to display a scrollable magazine article on the screen. Only the ScrollView containing the article text would scroll on the screen.



Beatles Anthology Vol. 1

Behind That Locked Door: Beatles Rarities!

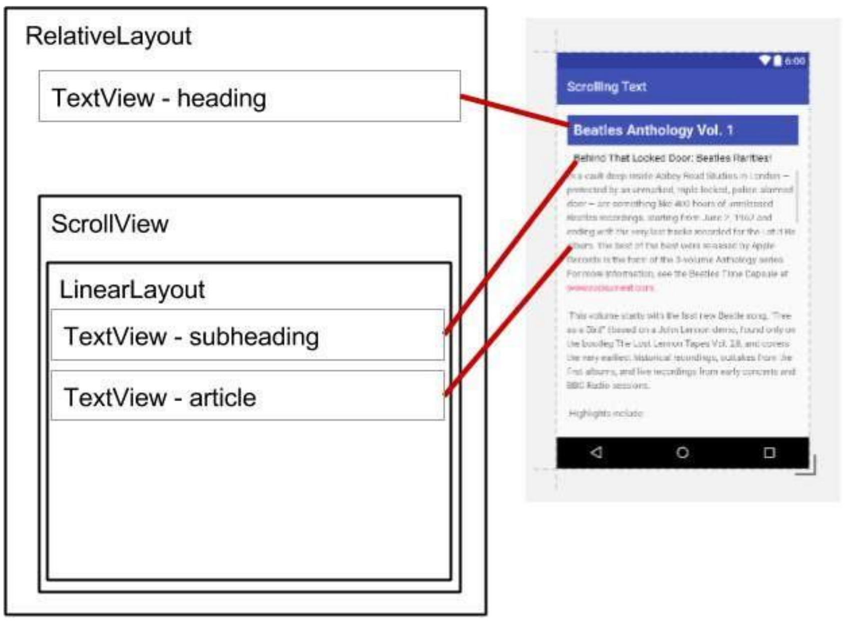
In a vault deep inside Abbey Road Studios in London – protected by an unmarked, triple locked, police alarmed door – are something like 400 hours of unreleased Beatles recordings, starting from June 7, 1962 and ending with the very last tracks recorded for the Let It Be album. The best of the best were released by Apple Records in the form of the 3-volume Anthology series. For more information, see the Beatles Time Capsule at www.rockument.com.

This volume starts with the last new Beatle song, "Free as a Bird" (based on a John Lennon demo, found only on the bootleg The Lost Lennon Tapes Vol. 28, and covers the very earliest historical recordings, outtakes from the first albums, and live recordings from early concerts and BBC Radio sessions.

Highlights include:

ScrollView with a LinearLayout

The ScrollView view group can contain only one view; however, that view can be a view group that contains views, such as LinearLayout. You can *nest* a view group such as LinearLayout *within* the ScrollView view group, thereby scrolling everything that is inside the LinearLayout.



When adding a LinearLayout inside a ScrollView, use `match_parent` for the LinearLayout's `android:layout_width` attribute to match the width of the parent view group (the ScrollView), and use `wrap_content` for the LinearLayout's `android:layout_height` attribute to make the view group only big enough to enclose its contents and padding.

Since ScrollView only supports vertical scrolling, you must set the LinearLayout orientation to vertical (by using the `android:orientation="vertical"` attribute), so that the entire LinearLayout will scroll vertically. For example, the following XML layout scrolls the `article` TextView along with the `article_subheading` TextView:

```
<ScrollView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/article_heading">

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

      <TextView
         android:id="@+id/article_subheading"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:padding="@dimen/padding_regular"
         android:text="@string/article_subtitle"
         android:textAppearance="@android:style/TextAppearance" />

      <TextView
         android:id="@+id/article"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:autoLink="web"
         android:lineSpacingExtra="@dimen/line_spacing"
         android:text="@string/article_text" />

   </LinearLayout>
</ScrollView>
```

```
<ScrollView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/article_heading">

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

      <TextView
         android:id="@+id/article_subheading"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:padding="@dimen/padding_regular"
         android:text="@string/article_subtitle"
         android:textAppearance="@android:style/TextAppearance" />

      <TextView
         android:id="@+id/article"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:autoLink="web"
         android:lineSpacingExtra="@dimen/line_spacing"
         android:text="@string/article_text" />

   </LinearLayout>
</ScrollView>
```

1.3: Resources to Help You Learn

Contents:

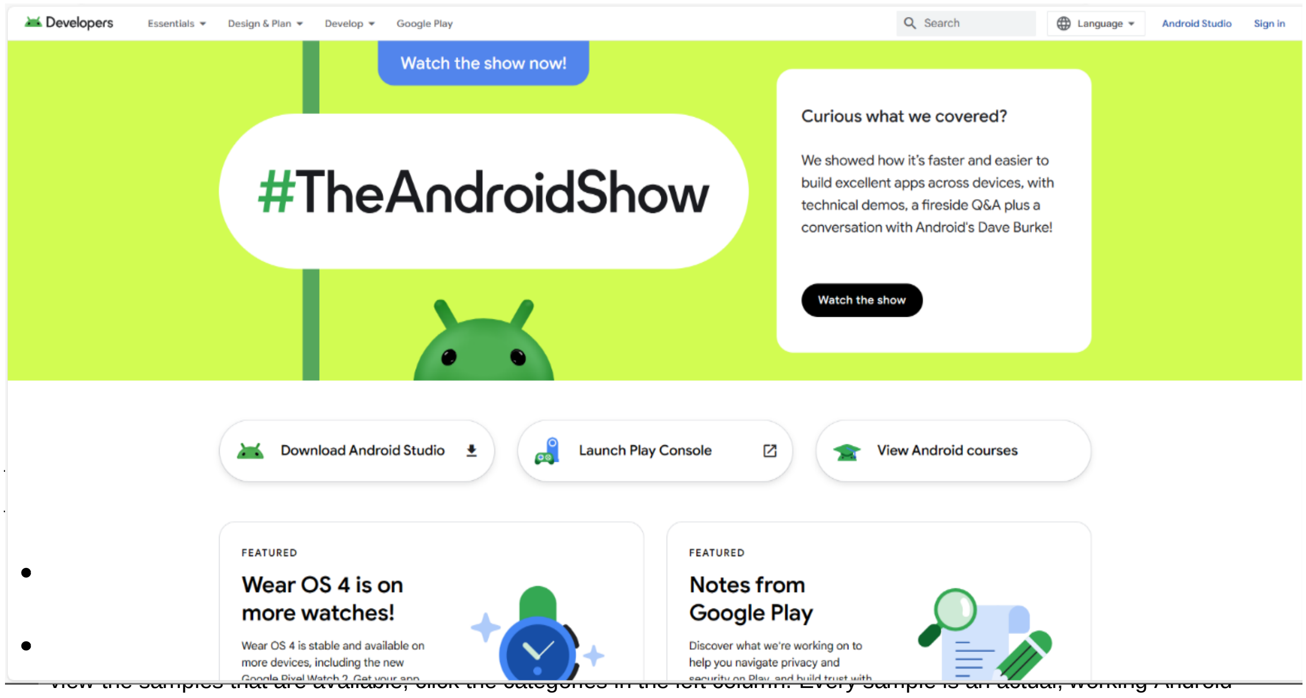
This chapter describes resources available for Android developers, and how to use them.

Exploring Android developer documentation

The best place to learn about Android development and to keep informed about the newest Android development tools is to browse the official Android developer documentation.

developer.android.com

Home page



Curious what we covered?

We showed how it's faster and easier to build excellent apps across devices, with technical demos, a fireside Q&A plus a conversation with Android's Dave Burke!

Wear OS 4 is on more watches!

Wear OS 4 is stable and available on more devices, including the new Google Pixel Watch 2. Get your app

Notes from Google Play

Discover what we're working on to help you navigate privacy and security on Play, and build trust with

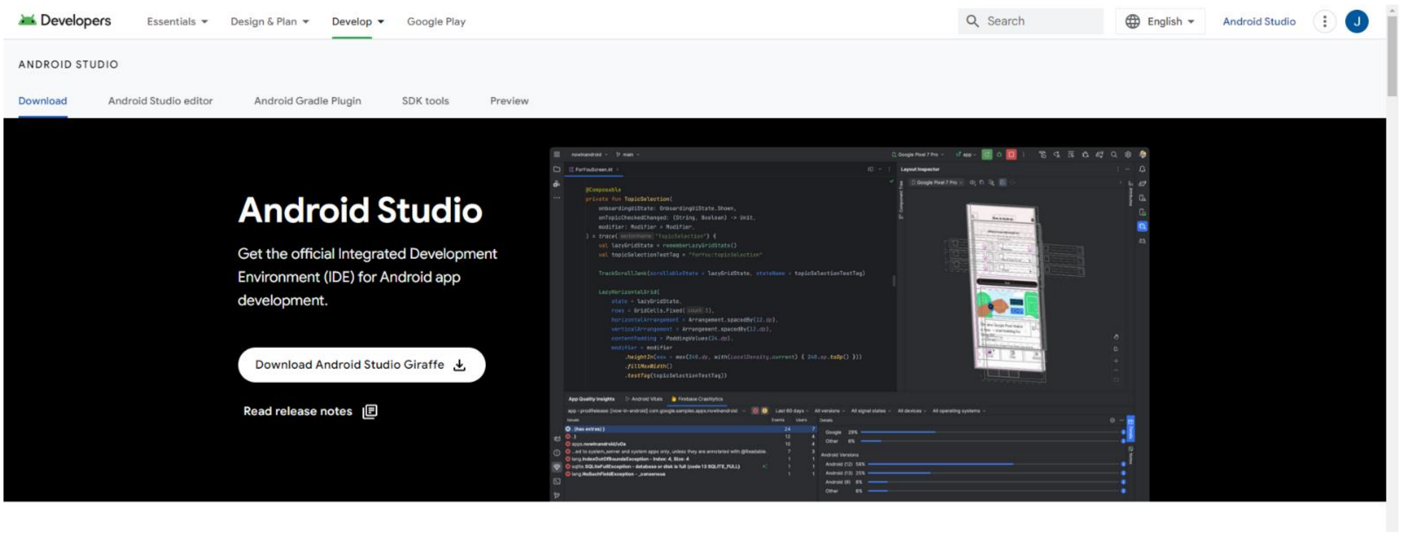
-
- view the samples that are available, click the categories in the left column. Every sample is an actual, working Android app. View the overall project structure and browse the sources and resources. You can copy and paste the necessary code and double-click a line to get the URL if you want to share a link to it. For more sample code, see "Exploring code samples in the Android SDK" in this chapter.
- **Watch stories**: Discover the apps created by other Android developers, as well as their experiences using Google Play and Android. The page provides videos and articles with the most recent news about Android development, including how developers enhanced user experiences and methods for boosting app user engagement.

  Links on the home page allow Android app developers to sign up for the Google Play developer program and preview their apps for the most recent Android version:

- **Developer Console**: The Google Play store is Google's digital distribution system for apps developed with the Android SDK. On the Google Play Developer Console page you can accept the Developer Agreement, pay the registration fee, and complete your account details in order to join the Google Play developer program.
- **Preview**: Visit the Android preview page to check the compatibility of your apps and to benefit from new features like app shortcuts, image keyboard support, circular icons, and more.
- **Android**, **Wear**, **TV**, and **Auto**: Learn about the newest versions of Android for smartphones and tablets, wearable

devices, television, and automobiles.

Android Studio page



- **Features**: Learn about the features of the newest and latest version of Android Studio.
- **Latest**: You can check and read news about Android Studio.
- **Resources**: Read articles about using Android Studio, including a basic introduction.
- **Videos**: Watch video tutorials about using Android Studio.
- **Download Options**: Download a version of Android Studio for a different operating system than the one you are using.

Android Studio documentation

The following are links into the Android Studio documentation that are useful for this training:

- Meet Android Studio
- Developer Workflow Basics
- Projects Overview
- Create App Icons with Image Asset Studio
- Add Multi-Density Vector Graphics
- Create and Manage Virtual Devices
- Android Monitor page
- Debug Your App
- Configure Your Build
- Sign Your App

Design, Develop, Distribute, and Preview

The Android documentation is accessible through the following links from the home page:

- **Design**: This section covers Material Design, which is a conceptual design philosophy that outlines how apps should look and work on mobile devices. Use the following links to learn more:
  - **Introducing material design**: An introduction to the material design philosophy.
  - **Downloads for designers**: Download color palettes for compatibility with the material design specification.
  - **Articles**: Check news and articles on Android design.
  - For links to tools like videos, templates, fonts, and color schemes, scroll down the Design page.
  - Links into the Design section that are helpful for this training include the ones below:
    - Material Design Guidelines
    - Style
    - Using the Material Theme

Components - Buttons
Dialogs design guide
Gestures design guide
Notification Design Guide
- Icons and other downloadable resources
- Design - Patterns - Navigation
- Drawable Resource Guide
- Styles and Themes Guide
- Settings
- Material Palette Generator

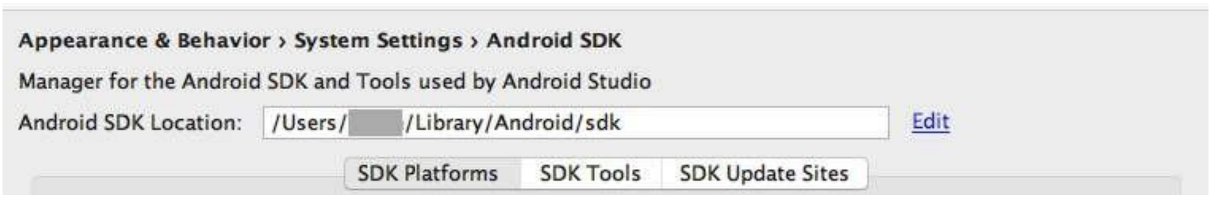
- **Develop**: In order to speed up your development, you can find application programming interface (API) details, reference materials, tutorials, tool guides, code samples, and information about Android's tools and libraries in this section. To find what you need, use the site navigation links in the left column or search. Popular links into the Develop section that are helpful for this training include the ones below:
  - Overview:
    - Introduction to Android
    - Vocabulary Glossary
    - Platform Architecture
    - Android Application Fundamentals
    - UI Overview
    - Platform Versions
    - Android Support Library
    - Working with System Permissions
  - Development practices:
    - Supporting Different Platform Versions
    - Supporting Multiple Screens
    - Supporting Different Densities
    - Best Practices for Interaction and Engagement
    - Best Practices for User Interface
    - Best Practices for Testing
    - Providing Resources
    - Optimizing Downloads for Efficient Network Access Guide
    - Best Practices for App Permissions
  - Articles and training guides:
    - Starting Another Activity
    - Specifying the Input Method Type
    - Handling Keyboard Input
    - Adding the App Bar
    - Using Touch Gestures
    - Creating Lists and Cards
    - Getting Started with Testing
    - Managing the Activity Lifecycle
    - Connecting to the Network
    - Managing Network Usage
    - Manipulating Broadcast Receivers On Demand
    - Scheduling Repeating Alarms
    - Transferring Data Without Draining the Battery
    - Saving Files
    - Saving Key-Value Sets
    - Saving Data in SQL Databases
    - Configuring Auto Backup for Apps
    - Working with System Permissions
  - General topics
    - Styles and Themes
    -

        Layouts
        - Menus
        - Intents and Intent Filters
        - Processes and threads
        - Loaders
        - Services
        - Notifications
        - Storage Options
        - Localizing with Resources
        - Content Providers
        - Cursors
        - Backing up App Data to the Cloud
        - Settings
        - System Permissions
    - Reference information:
        - Support Library
        - Android standard R.color resources
        - App Resources
- **Distribute**: This section provides information about everything that happens after you've written your app: putting it on the Play Store, growing your user base, and earning money.
    - Google Play
    - Essentials for a Successful App
    - Launch Checklist

Installing offline documentation

To access to documentation even when you are not connected to the internet, install the Software Development Kit (SDK) documentation using the SDK Manager. Follow these steps:

1. Choose **Tools > Android > SDK Manager**.
2. In the left column, click **Android SDK**.
3. To find the documentation on your computer, choose and copy the path for the Android SDK Location at the top of the screen:



Appearance & Behavior › System Settings › Android SDK

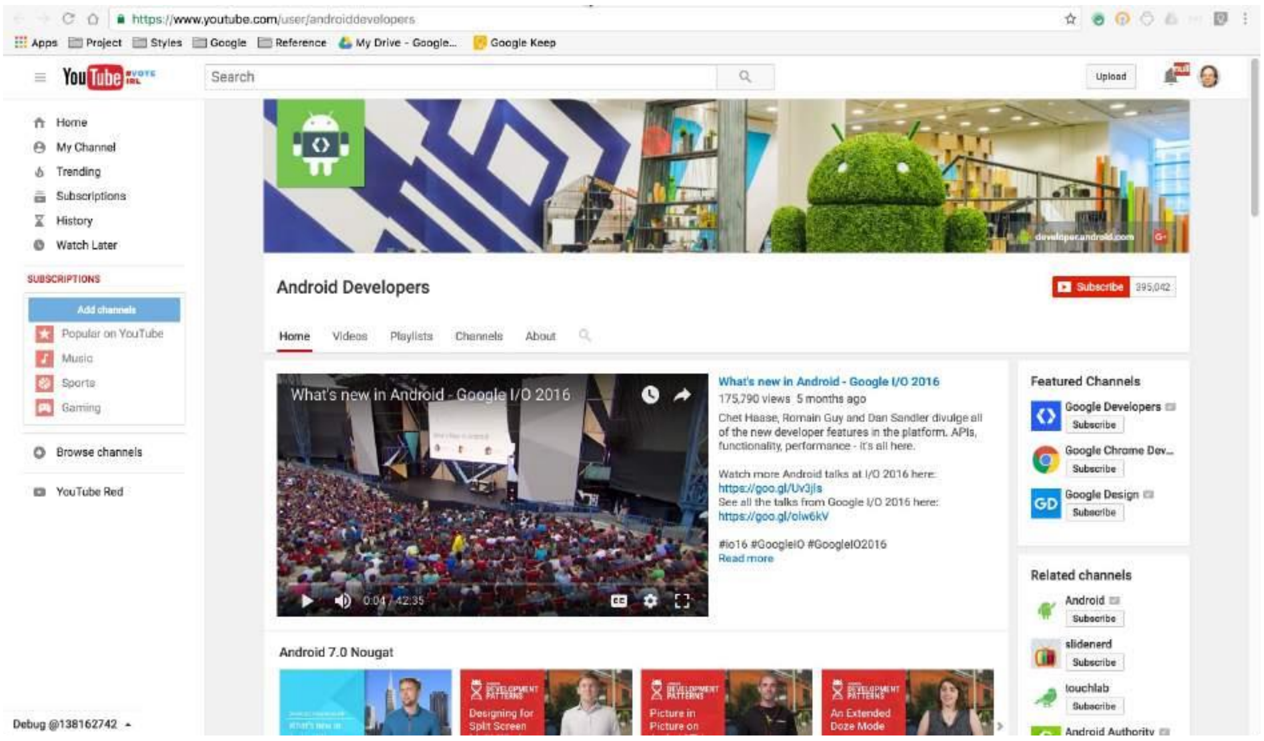
Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: /Users/[redacted]/Library/Android/sdk

4. Click the **SDK Tools** tab. You can install additional SDK Tools that are not installed by default, as well as an offline version of the Android developer documentation.
5. Click the checkbox for "Documentation for Android SDK" if it is not already installed, and click **Apply**.
6. When the installation finishes, click **Finish**.
7. Navigate to the **sdk** directory you copied above, and open the **docs** directory.
8. Find **index.html** and open it.

Watching developer videos

In addition to the Android documentation, the Android Developer YouTube channel is a great source of tutorials and tips. For email notifications of new videos, you can subscribe to the channel. Click the red Subscribe button in the top right corner as displayed below to become a subscriber.



The following are popular videos referred to in this training:

- Debugging and Testing in Android Studio
- Android Testing Support - Android Testing Patterns #1
- Android Testing Support - Android Testing Patterns #2
- Android Testing Support - Android Testing Patterns #3
- Threading Performance 101
- Good AsyncTask Hunting
- Scheduling Alarms Presentation
- RecyclerView Animations and Behind the Scenes (Android Dev Summit 2015)
- Android Application Architecture: The Next Billion Users
- Android Performance Patterns Playlist

In addition, Udacity offers online Android development course