Android Architecture & first Android app

A *project* in Android Studio contains everything that defines your workspace for an app, from source code and assets, to test code and build configurations. When you start a new project, Android Studio creates the necessary structure for all your files and makes them visible in the **Project** window on the left side of the IDE (click **View > Tool Windows > Project**). This page provides an overview of the key components inside your project.

Modules

A *module* is a collection of source files and build settings that allow you to divide your project into discrete units of functionality. Your project can have one or many modules and one module may use another module as a dependency. Each module can be independently built, tested, and debugged.

Additional modules are often useful when creating code libraries within your own project or when you want to create different sets of code and resources for different device types, such as phones and wearables, but keep all the files scoped within the same project and share some code.

You can add a new module to your project by clicking **File > New > New Module**.

Android Studio offers a few distinct types of module:

Android app module

Provides a container for your app's source code, resource files, and app level settings such as the module-level build file and Android Manifest file. When you create a new project, the default module name is "app".

In the **Create New Module** window, Android Studio offers the following app modules:

- Phone & Tablet Module
- Wear OS Module
- Android TV Module
- Glass Module

They each provide essential files and some code templates that are appropriate for the corresponding app or device type.

For more information on adding a module, read Add a Module for a New Device.

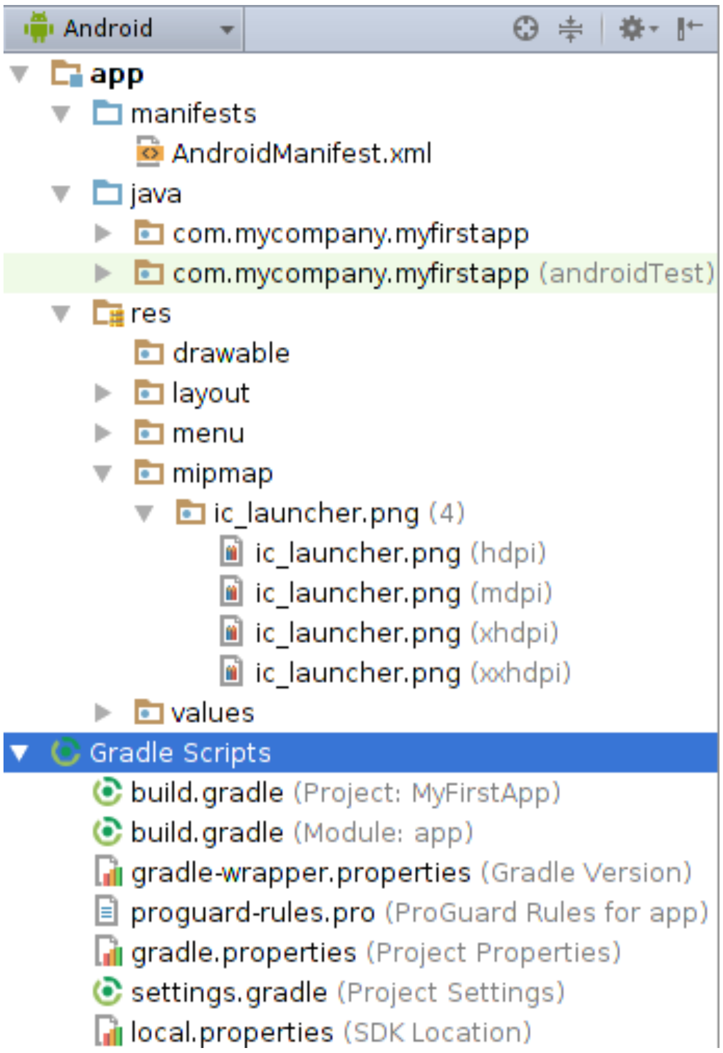
Library module

Provides a container for your reusable code, which you can use as a dependency in other app modules or import into other projects. Structurally, a library module is the same as an app module, but when built, it creates a code archive file instead of an APK, so it can't be installed on a device.

In the **Create New Module** window, Android Studio offers the following library modules:

- Android Library: This type of library can contain all file types supported in an Android project, including source code, resources, and manifest files. The build result is an Android Archive (AAR) file that you can add as a dependency for your Android app modules.
- Java Library: This type of library can contain only Java source files. The build result is an Java Archive (JAR) file that you can add as a dependency for your Android app modules or other Java projects.

Project files


```
Android
▼ app
    ▼ manifests
        AndroidManifest.xml
    ▼ java
        ▶ com.mycompany.myfirstapp
        ▶ com.mycompany.myfirstapp (androidTest)
    ▼ res
        drawable
        ▶ layout
        ▶ menu
        ▼ mipmap
            ▼ ic_launcher.png (4)
                ic_launcher.png (hdpi)
                ic_launcher.png (mdpi)
                ic_launcher.png (xhdpi)
                ic_launcher.png (xxhdpi)
        ▶ values
▼ Gradle Scripts
    build.gradle (Project: MyFirstApp)
    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)
```

By default, Android Studio displays your project files in the **Android** view. This view does not reflect the actual file hierarchy on disk, but is organized by modules and file types to simplify navigation between key source files of your project, hiding certain files or directories that are not commonly used. Some of the structural changes compared to the structure on disk include the following:

- Shows all the project's build-related configuration files in a top-level **Gradle Script** group.
- Shows all manifest files for each module in a module-level group (when you have different manifest files for different product flavors and build types).
- Shows all alternative resource files in a single group, instead of in separate folders per resource qualifier. For example, all density versions of your launcher icon are visible side-by-side.

Within each Android app module, files are shown in the following groups:

manifests

Contains the `AndroidManifest.xml` file.

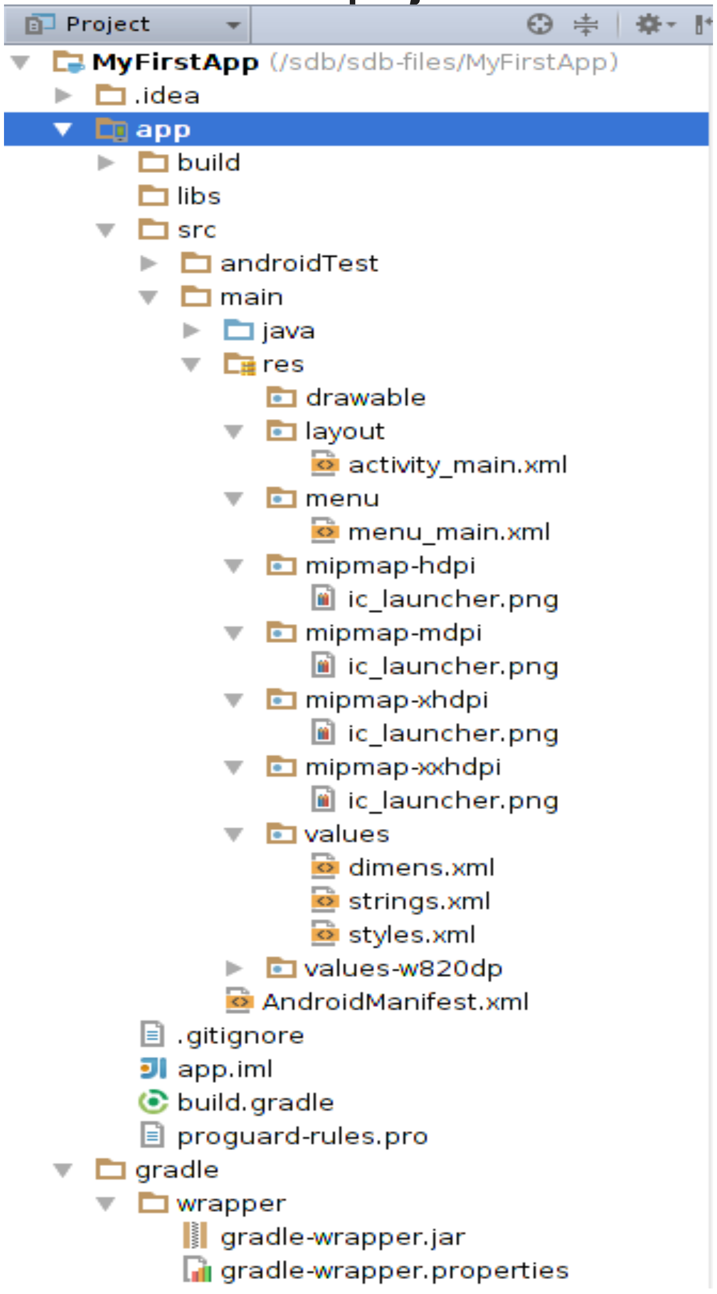
java

Contains the Java source code files, separated by package names, including JUnit test code.

res

Contains all non-code resources, such as XML layouts, UI strings, and bitmap images, divided into corresponding sub-directories. For more information about all possible resource types, see Providing Resources.

The Android project view



```
▼ MyFirstApp (/sdb/sdb-files/MyFirstApp)
  ► .idea
  ▼ app
    ► build
      libs
    ▼ src
      ► androidTest
      ▼ main
        ► java
        ▼ res
            drawable
          ▼ layout
              activity_main.xml
          ▼ menu
              menu_main.xml
          ▼ mipmap-hdpi
              ic_launcher.png
          ▼ mipmap-mdpi
              ic_launcher.png
          ▼ mipmap-xhdpi
              ic_launcher.png
          ▼ mipmap-xxhdpi
              ic_launcher.png
          ▼ values
              dimens.xml
              strings.xml
              styles.xml
          ► values-w820dp
          AndroidManifest.xml
      .gitignore
      app.iml
      build.gradle
      proguard-rules.pro
  ▼ gradle
    ▼ wrapper
        gradle-wrapper.jar
        gradle-wrapper.properties
```

To see the actual file structure of the project including all files hidden from the Android view, select **Project** from the dropdown at the top of the **Project** window.

When you select **Project** view, you can see a lot more files and directories. The most important of which are the following:

`module-name``/`

`build/`

Contains build outputs.

`libs/`

Contains private libraries.

`src/`

Contains all code and resource files for the module in the following subdirectories:

`androidTest/`

Contains code for instrumentation tests that run on an Android device. For more information, see the Android Test documentation.

`main/`

Contains the "main" sourceset files: the Android code and resources shared by all build variants (files for other build variants reside in sibling directories, such as `src/debug/` for the debug build type).

`AndroidManifest.xml`

Describes the nature of the application and each of its components. For more information, see theAndroidManifest.xml documentation.

`java/`

Contains Java code sources.

`jni/`

Contains native code using the Java Native Interface (JNI). For more information, see the Android NDK documentation.

`gen/`

Contains the Java files generated by Android Studio, such as your `R.java` file and interfaces created from AIDL files.

`res/`

Contains application resources, such as drawable files, layout files, and UI string. See Application Resources for more information.

`assets/`

Contains file that should be compiled into an `.apk` file as-is. You can navigate this directory in the same way as a typical file system using URIs and read files as a stream of bytes using the `AssetManager` . For example, this is a good location for textures and game data.

`test/`

Contains code for local tests that run on your host JVM.

`build.gradle` **(module)**

This defines the module-specific build configurations.

`build.gradle` **(project)**

This defines your build configuration that apply to all modules. This file is integral to the project, so you should maintain them in revision control with all other source code.

For information about other build files, see Configure Your Build.

Project structure settings

To change various settings for your Android Studio project, open the **Project Structure** dialog by clicking **File > Project Structure**. It contains the following sections:

- **SDK Location:** Sets the location of the JDK, Android SDK, and Android NDK that your project uses.
- **Project:** Sets the version for Gradle and the Android plugin for Gradle, and the repository location name.
- **Developer Services:** Contains settings for Android Studio add-in components from Google or other third parties. See Developer Services, below.
- **Modules:** Allows you to edit module-specific build configurations, including the target and minimum SDK, the app signature, and library dependencies. See Modules, below.

Developer services

The *Developer Services* section of the *Project Structure* dialog box contains configuration pages for several services that you can use with your app. This section contains the following pages:

- **AdMob:** Allows you to turn on Google's AdMob component, which helps you understand your users and show them tailored advertisements.
- **Analytics:** Allows you to turn on Google Analytics, which helps you measure user interactions with your app across various devices and environments.
- **Authentication:** Allows users to use Google Sign-In to sign in to your app with their Google accounts.
- **Cloud:** Allows you to turn on Firebase cloud-based services for your app.
- **Notifications:** Allows you to use Google Cloud Messaging to communicate between your app and your server.

Turning on any of these services may cause Android Studio to add necessary dependencies and permissions to your app. Each configuration page lists these and other actions that Android Studio takes if you enable the associated service.

Modules

The *Modules* settings section lets you change configuration options for each of your project's modules. Each module's settings page is divided into the following tabs:

- **Properties:** Specifies the versions of the SDK and build tools to use to compile the module.
- **Signing:** Specifies the certificate to use to sign your APK.
- **Flavors:** Lets you create multiple build *flavors*, where each flavor specifies a set of configuration settings, such as the module's minimum and target SDK version, and the version code and version name. For example, you might define one flavor that has a minimum SDK of 15 and a target SDK of 21, and another flavor that has a minimum SDK of 19 and a target SDK of 23.
- **Build Types:** Lets you create and modify build configurations, as described in Configuring Gradle Builds. By default, every module has *debug* and *release*build types, but you can define more as needed.
- **Dependencies:** Lists the library, file, and module dependencies for this module. You can add, modify, and delete dependencies from this pane. For more information about module dependencies, see Configuring Gradle Builds.

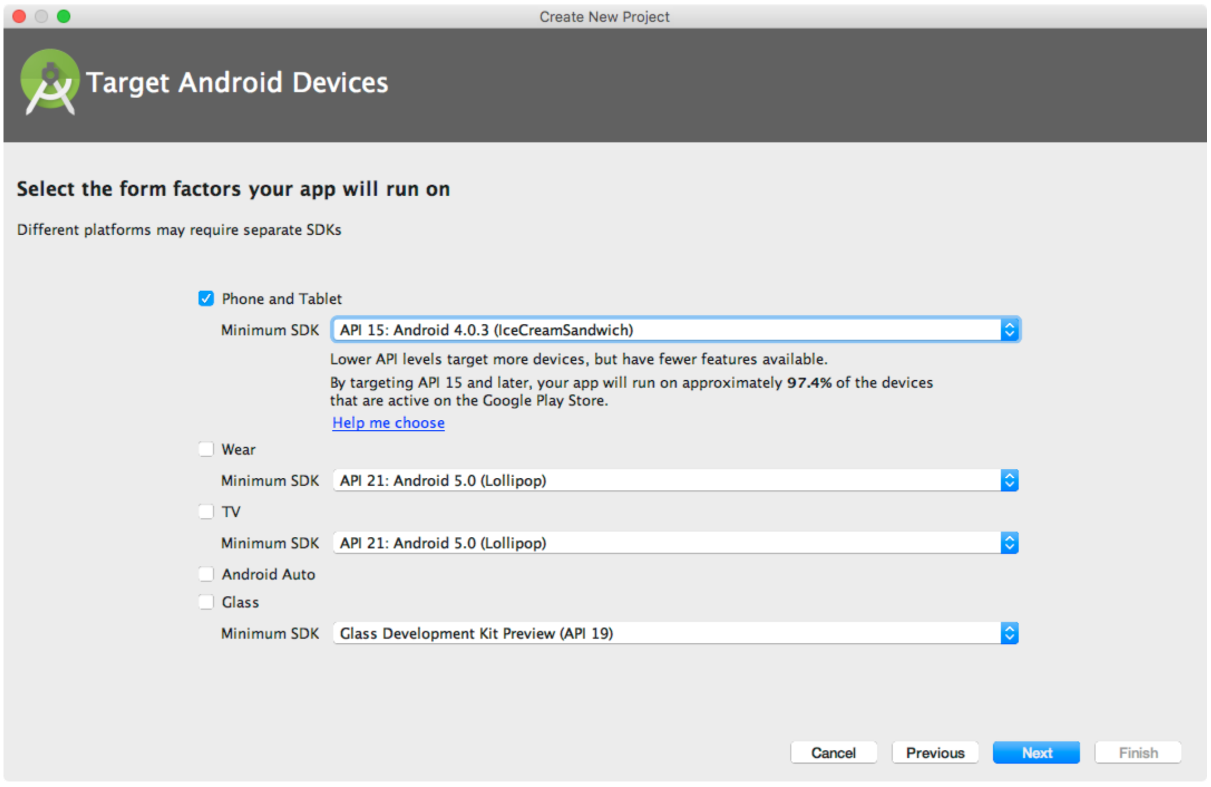
Starting an Android Studio project

After you have successfully installed the Android Studio IDE, double-click the Android Studio application icon to start it. Choose **Start a new Android Studio project** in the Welcome window, and name the project the same name that you want to use for the app.

When choosing a unique Company Domain, keep in mind that apps published to the Google Play must have a unique package name. Since domains are unique, prepending the app's name with your name, or your company's domain name, should provide an adequately unique package name. If you are not planning to publish the app, you can accept the default example domain. Be aware that changing the package name later is extra work.

Choosing target devices and the minimum SDK

When choosing Target Android Devices, Phone and Tablet are selected by default, as shown in the figure below. The choice shown in the figure for the Minimum SDK — **API 15: Android 4.0.3 (IceCreamSandwich)** — makes your app compatible with 97% of Android devices active on the Google Play Store.


Create New Project

Target Android Devices

Select the form factors your app will run on

Different platforms may require separate SDKs

☑ Phone and Tablet

Minimum SDK: API 15: Android 4.0.3 (IceCreamSandwich)

Lower API levels target more devices, but have fewer features available.

By targeting API 15 and later, your app will run on approximately **97.4%** of the devices that are active on the Google Play Store.

Help me choose

☐ Wear

Minimum SDK: API 21: Android 5.0 (Lollipop)

☐ TV

Minimum SDK: API 21: Android 5.0 (Lollipop)

☐ Android Auto

☐ Glass

Minimum SDK: Glass Development Kit Preview (API 19)

Different devices run different versions of the Android system, such as Android 4.0.3 or Android 4.4. Each successive version often adds new APIs not available in the previous version. To indicate which set of APIs are available, each version specifies an API level. For instance, Android 1.0 is API level 1 and Android 4.0.3 is API level 15.

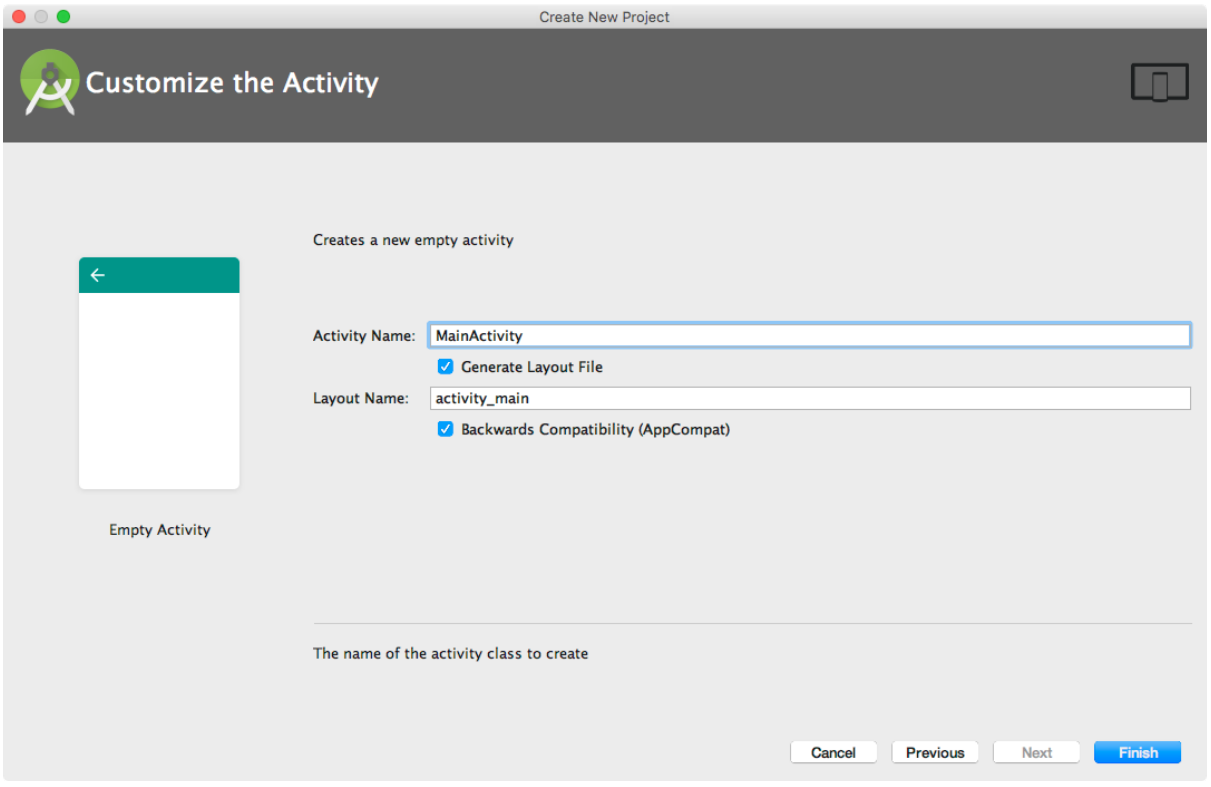
The Minimum SDK declares the minimum Android version for your app. Each successive version of Android provides compatibility for apps that were built using the APIs from previous versions, so your app should *always* be compatible with future versions of Android while using the documented Android APIs.

Choosing a template

Android Studio pre-populates your project with minimal code for an activity and a screen layout based on a *template*. A variety of templates are available, ranging from a virtually blank template (Add No Activity) to various types of activities.

You can customize the activity after choosing your template. For example, the Empty Activity template provides a single activity accompanied by a single layout resource for the screen. You can choose to accept the commonly used name for the activity (such as **MainActivity**) or change the name on the Customize the Activity screen. Also, if you use the Empty Activity template, be sure to check the following if they are not already checked:

- Generate Layout file: Leave this checked to create the layout resource connected to this activity, which is usually named **activity_main.xml**. The layout defines the user interface for the activity.
- Backwards Compatibility (AppCompat)**:** Leave this checked to include the AppCompat library so that the app is compatible with previous versions of Android even if it uses features found only in newer versions.



Creates a new empty activity

Activity Name: MainActivity

☑ Generate Layout File

Layout Name: activity_main

☑ Backwards Compatibility (AppCompat)

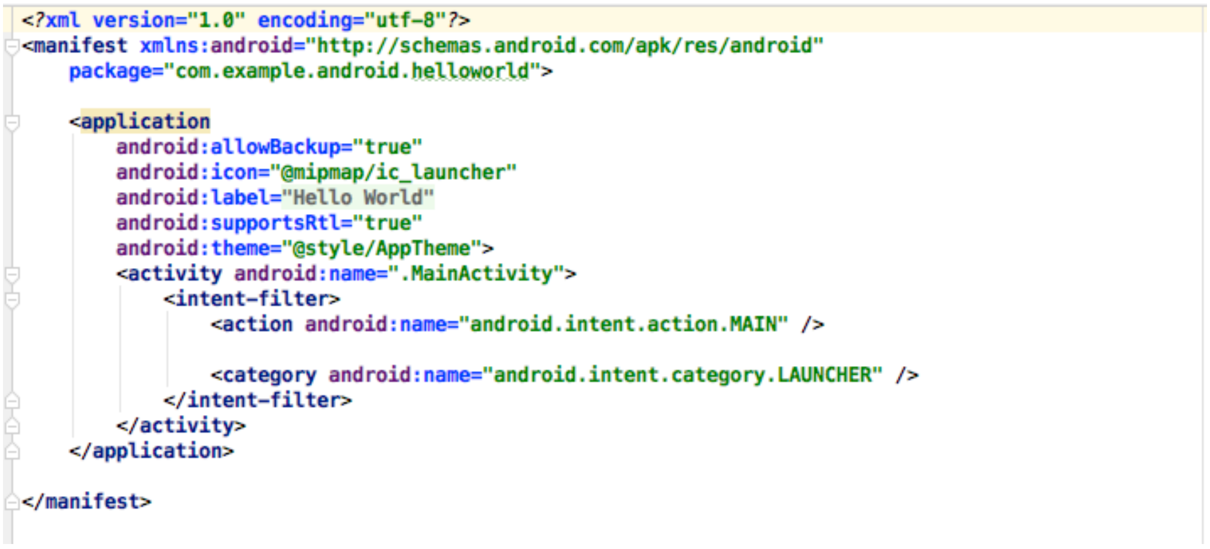
The name of the activity class to create

Android Studio creates a folder for the newly created project in the AndroidStudioProjects folder on your computer.

Viewing the Android Manifest

Before the Android system can start an app component, the system must know that the component exists by reading the app's AndroidManifest.xml file. The app must declare all its components in this file, which must be at the root of the app project directory.

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Hello World"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

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"
   ...
```

Setting the `android:allowBackup` attribute to `true` enables the app to be backed up automatically and restored as needed. Users invest time and effort to configure apps. Switching to a new device can cancel out all that careful configuration. The system performs this automatic backup for nearly all app data by default, and does so without the developer having to write any additional app code.
For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically create backups of app data to the cloud because the `android:allowBackup` attribute defaults to `true` if omitted. For apps < API level 22 you have to explicitly add the `android:allowBackup`attribute and set it to `true`.
Tip: To learn more about the automatic backup for apps, see [Configuring Auto Backup for Apps]().

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`. You can change the app name by changing the `Hello World`string to something else. String resources are described in a separate 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 devices may run different versions of the Android system, such as Android 4.0 or Android 4.4. Each successive version can add new APIs not available in the previous version. To indicate which set of APIs are available, each version specifies an API level. For instance, Android 1.0 is API level 1 and Android 4.4 is 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. For example, the Calendar Provider APIs were added in Android 4.0 (API level 14). If your app can't function without these APIs, declare API level 14 as the app's minimum supported version like this:

```
<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 does *not* prevent an app from being installed on Android versions that are higher (newer) than the specified value, but it is important because it indicates to the system whether the app should inherit behavior changes in newer versions. If you don't update the `targetSdkVersion` to the latest version, the system assumes that your app requires some backward-compatibility behaviors when running on the latest version. For example, among the behavior changes in Android 4.4, alarms created with the AlarmManager APIs are now inexact by default so that the system can batch app alarms and preserve system power, but the system will retain the previous API behavior for an app if your target API level is lower than `"19"`.

Using the log

The log is a powerful debugging tool you can use to look at values, execution paths, and exceptions. After you add logging statements to an app, your log messages appear along with general log messages in the logcat tab of the Android Monitor pane of Android Studio.

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();
```

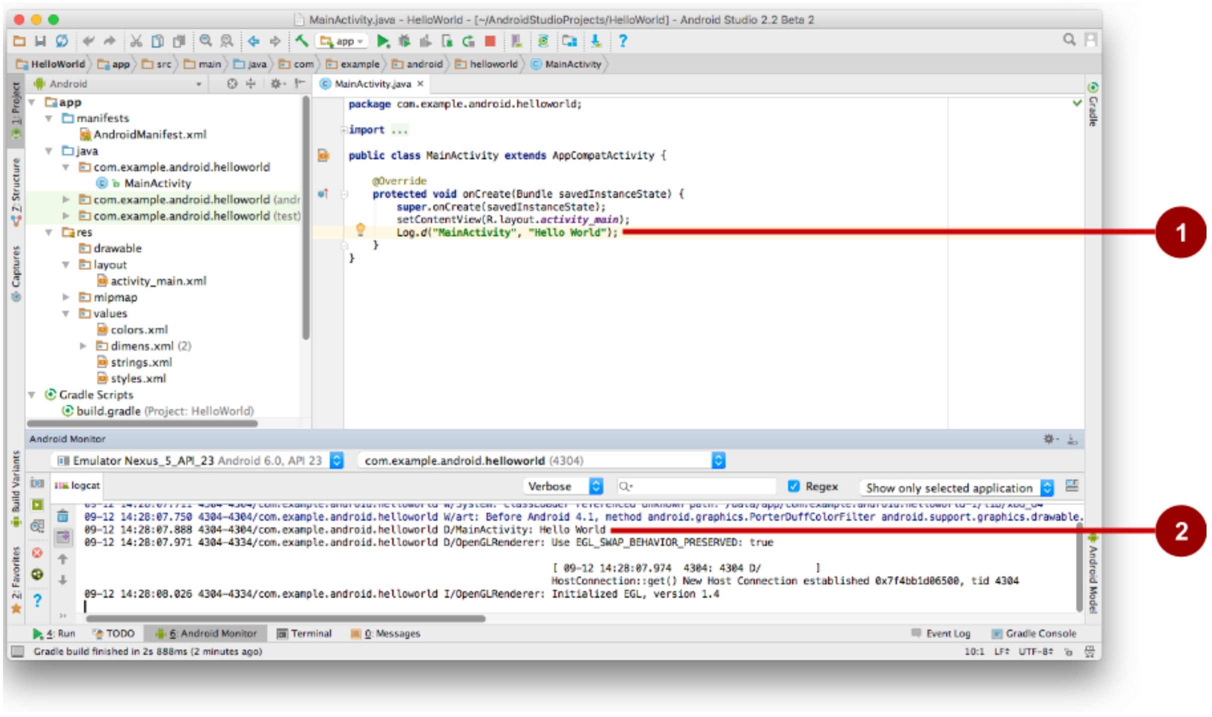
2. Use the constant in the logging statements:

```
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

The Run pane appears in place of the Android Monitor pane when you run the app on an emulator or a device. After starting to run the app, click the **Android Monitor**button at the bottom of the main window, and then click the **logcat** tab in the Android Monitor pane if it is not already selected.



```
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
HelloWorld › app › src › main › java › com › example › android › helloworld › MainActivity
Android
app
manifests
AndroidManifest.xml
java
com.example.android.helloworld
MainActivity
com.example.android.helloworld (androidTest)
com.example.android.helloworld (test)
res
drawable
layout
activity_main.xml
mipmap
values
colors.xml
dimens.xml (2)
strings.xml
styles.xml
Gradle Scripts
build.gradle (Project: HelloWorld)
Android Monitor
Emulator Nexus_5_API_23 Android 6.0, API 23
com.example.android.helloworld (4304)
logcat
Verbose
Regex
Show only selected application
4: Run
TODO
6: Android Monitor
Terminal
0: Messages
Event Log
Gradle Console
Gradle build finished in 2s 888ms (2 minutes ago)
1
2

In the above figure:

1. The logging statement in the `onCreate()` method of `MainActivity`.
2. Android Monitor pane showing `logcat` log messages, including the message from the logging statement.

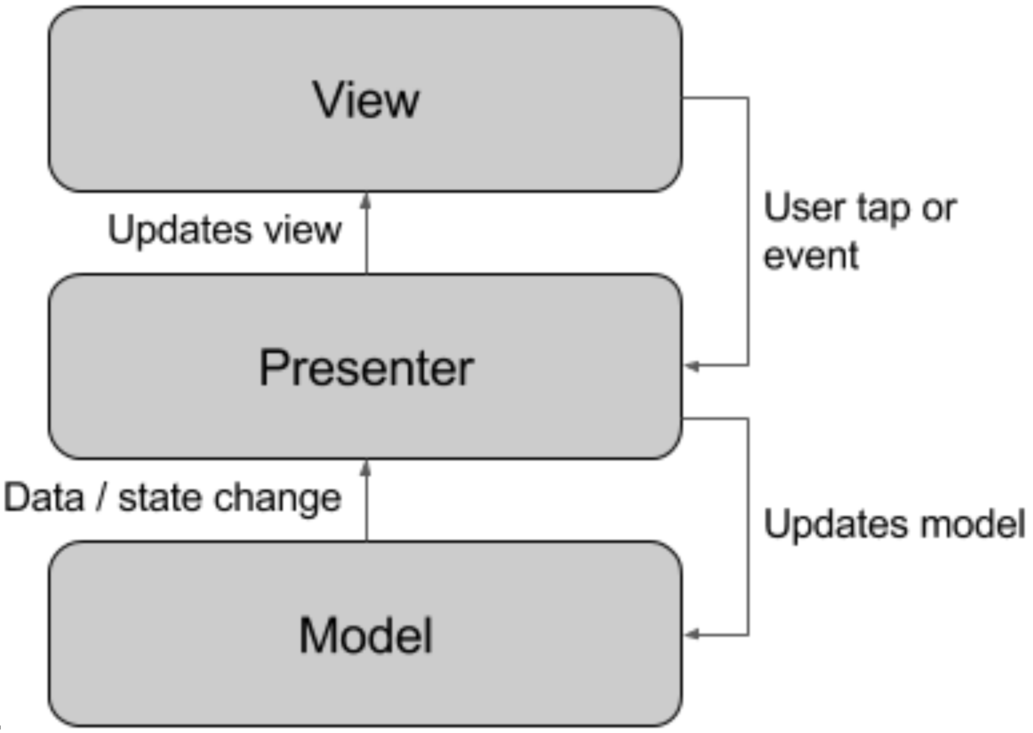
   By default, the log display is set to **Verbose** in the drop-down menu at the top of the **logcat** display to show all messages. You can change this to **Debug** to see messages that start with `Log.d`, or change it to **Error** to see messages that start with `Log.e`, and so on.

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 user interface elements that display data and respond to user actions. Every element of the screen is a view. The Android system provides many different kinds of views.
- **Presenters.** Presenters connect the application's views to the model. They supply the views with data as specified by the model, and also provide the model with user input from the view.
- **Model.** The model specifies the structure of the app's data and the code to access and manipulate the data. Some of the apps you create in the lessons

work with models for accessing data. The Hello Toast app does not use a data model, but you can think of its logic — display a message, and increase a tap counter — as the



model.

Views

The UI consists of a hierarchy of objects called *views* — every element of the screen is a *view*. The View class represents the basic building block for all UI components, and the base class for classes that provide interactive UI components such as buttons, checkboxes, and text entry fields.

A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the device-independent pixel (dp).

The Android system provides hundreds of predefined views, including those that display:

- 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.

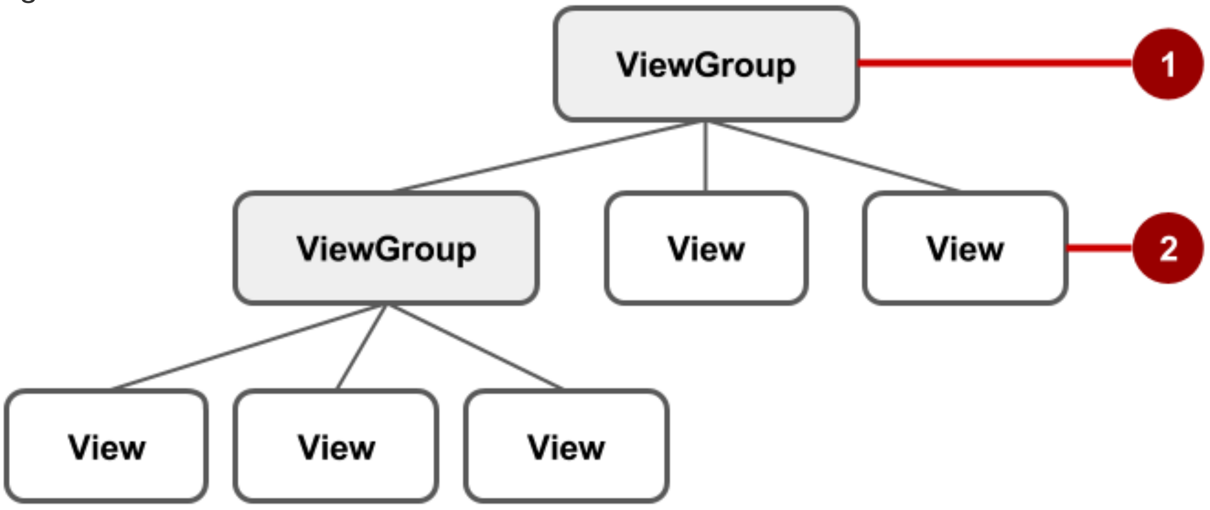
View groups

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 contains one other child view and enables scrolling the child view.
- RecyclerView: A group that contains a list of other views or view groups and enables scrolling them by adding and removing views dynamically from the screen.

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.



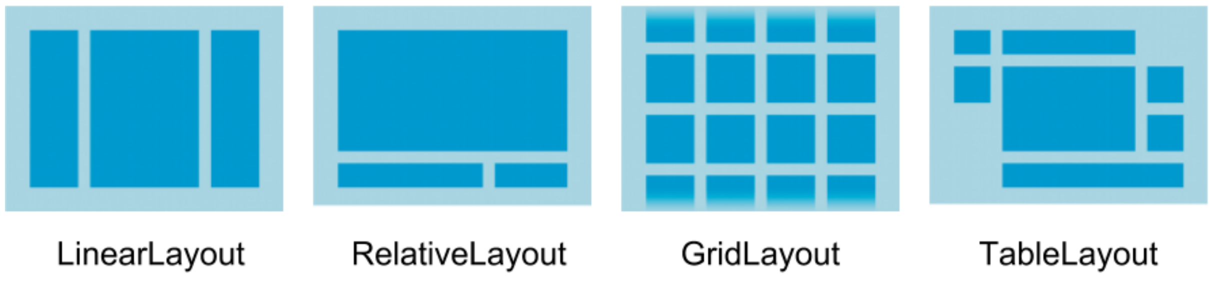
In the above figure:

1. The *root* view group.
2. The first set of child views and view groups whose parent is the root.

   Some view groups are designated as *layouts* because they organize child views in a specific way and are typically used as the root view group. Some examples of layouts are:

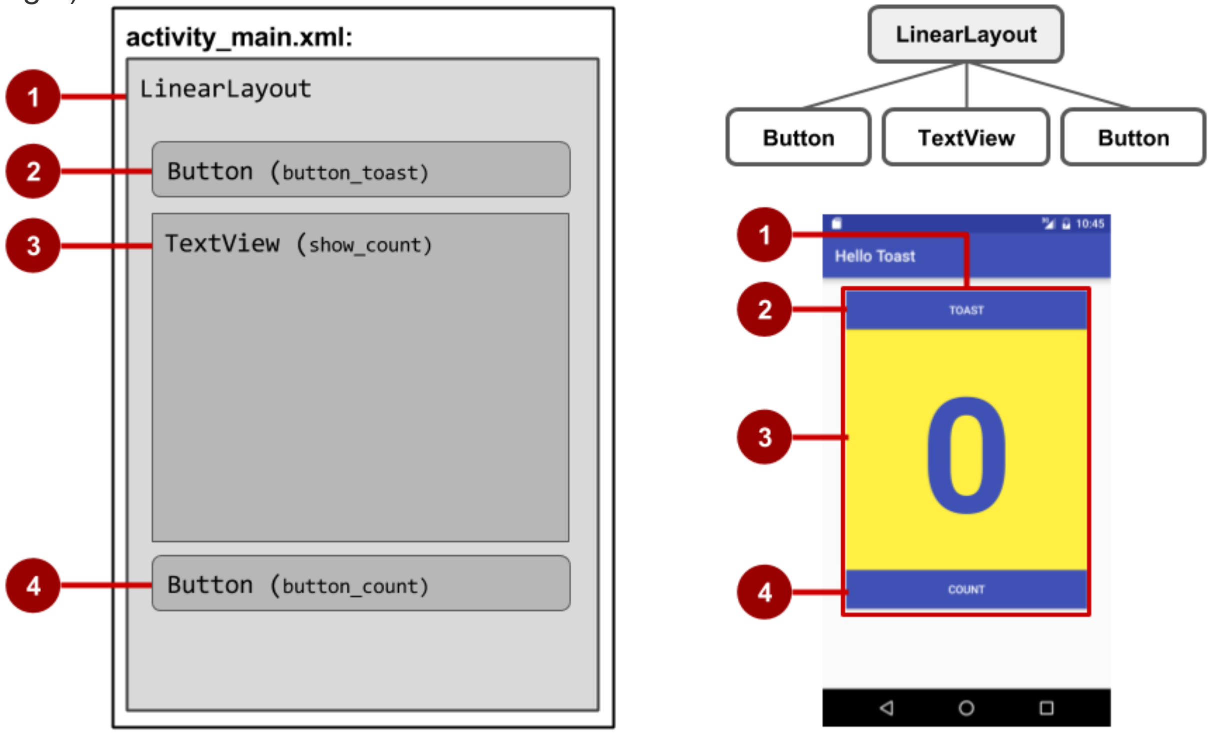
- LinearLayout: A group of child views positioned and aligned horizontally or vertically.

- 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.

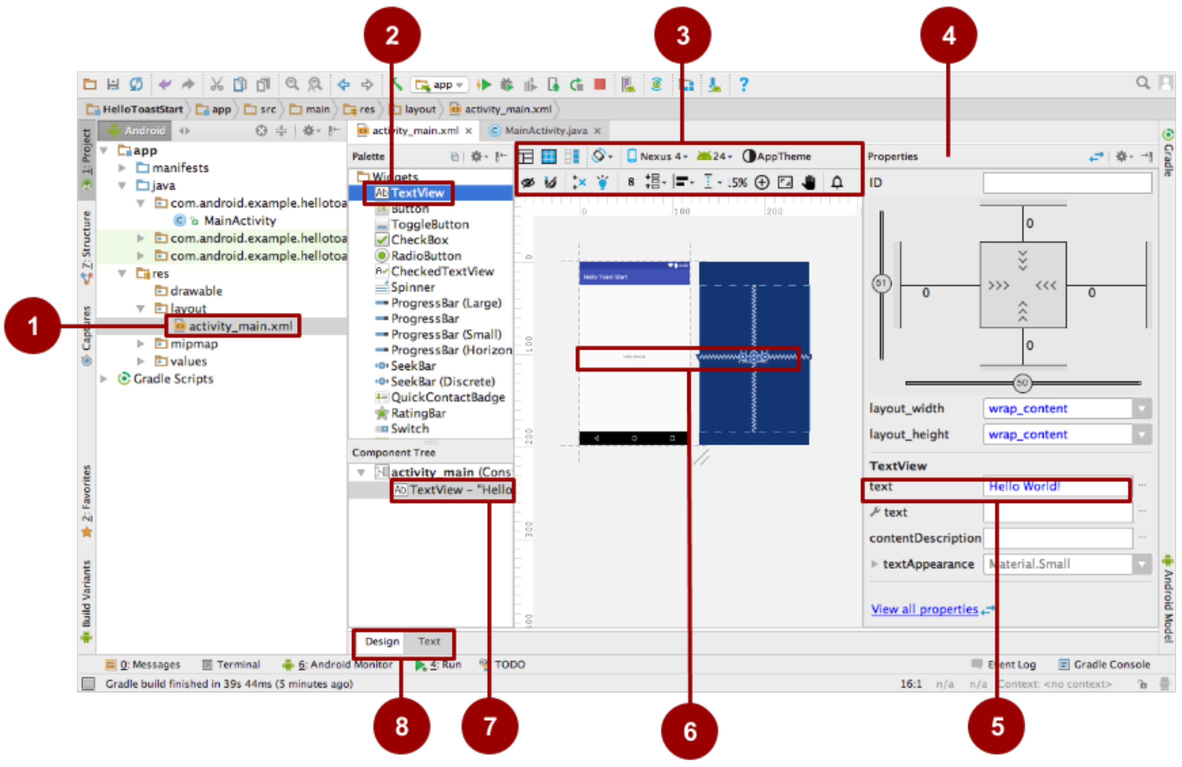
Tip: You can explore the view hierarchy of your app using Hierarchy Viewer. It shows a tree view of the hierarchy and lets you analyze the performance of views on an Android device. Performance issues are covered in a subsequent chapter.

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

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:

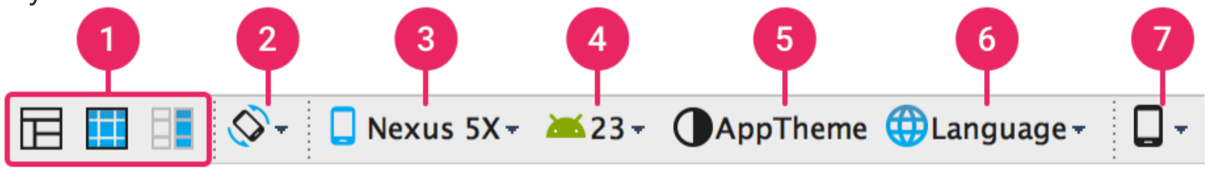
1. **XML layout** file. The XML layout file, typically named **activiy_main.xml** file. Double-click it to open the layout editor.
2. **Palette of UI elements** (views). The Palette pane provides a list of UI elements and layouts. Add an element or layout to the UI by dragging it into the design pane.
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. **Properties pane.** The Properties pane provides property controls for the selected view.
5. **Property control.** Property 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** and **Text** tabs. Click **Design** to see the layout editor, or **Text** to see XML code.

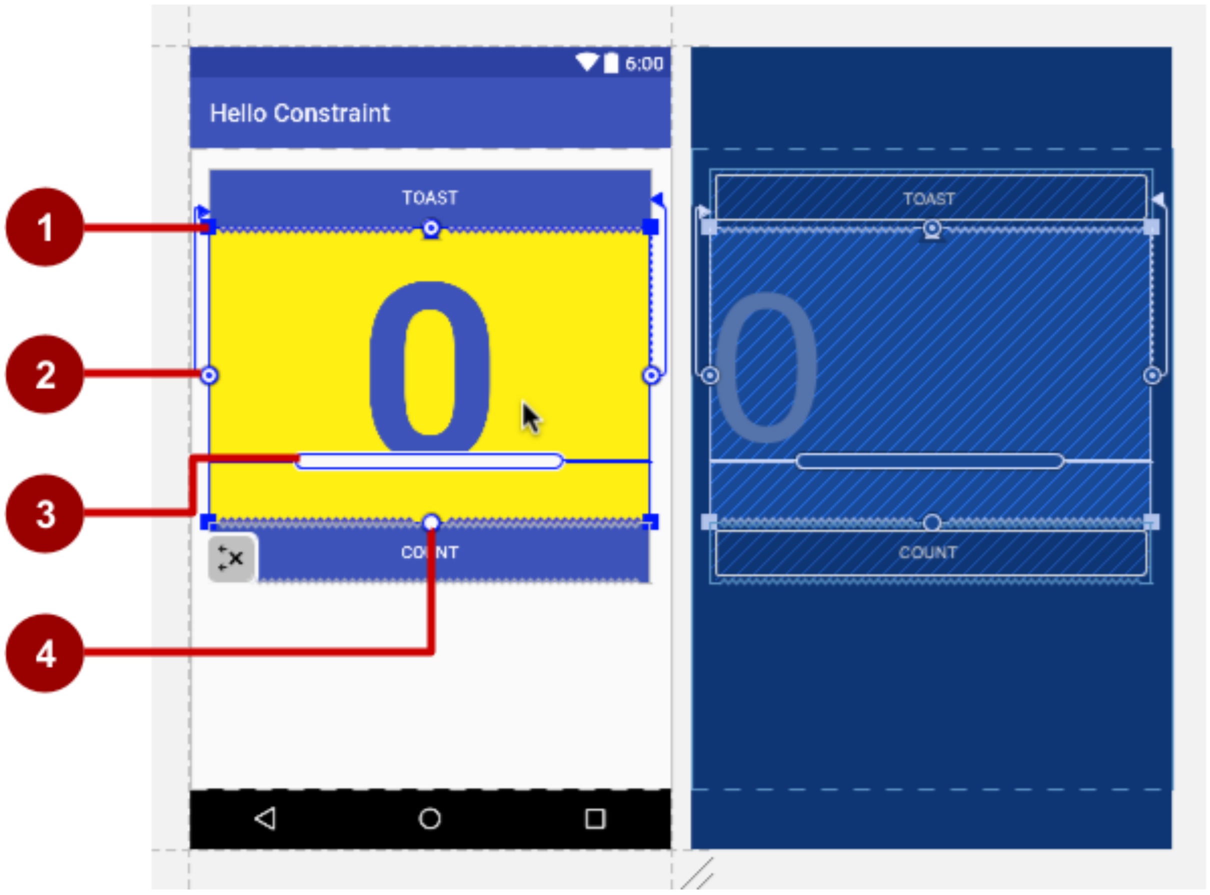
   The layout editor's design toolbar offers a row of buttons that let you configure the appearance of the
   layout:



   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. **Device type and size**: Select the device type (phone/tablet, Android TV, or Android Wear) and screen configuration (size and density).
4. **API version**: Select the version of Android on which to preview the layout.
5. **App theme**: Select which UI theme to apply to the preview.
6. **Language**: Select the language to show for your UI strings. This list displays only the languages available in the string resources.
7. **Layout Variants**: Switch to one of the alternative layouts for this file, or create a new one.

   The layout editor offers more features in the **Design** tab when you use a ConstraintLayout, including handles for defining constraints. A *constraint* is a connection or alignment to another view, to the parent layout, or to an invisible guideline. Each constraint appears as a line extending from a circular handle. Each view has a circular constraint handle in the middle of each side. After selecting a view in the Component Tree pane or clicking on it in the layout, the view also shows resizing handles on each corner.



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:

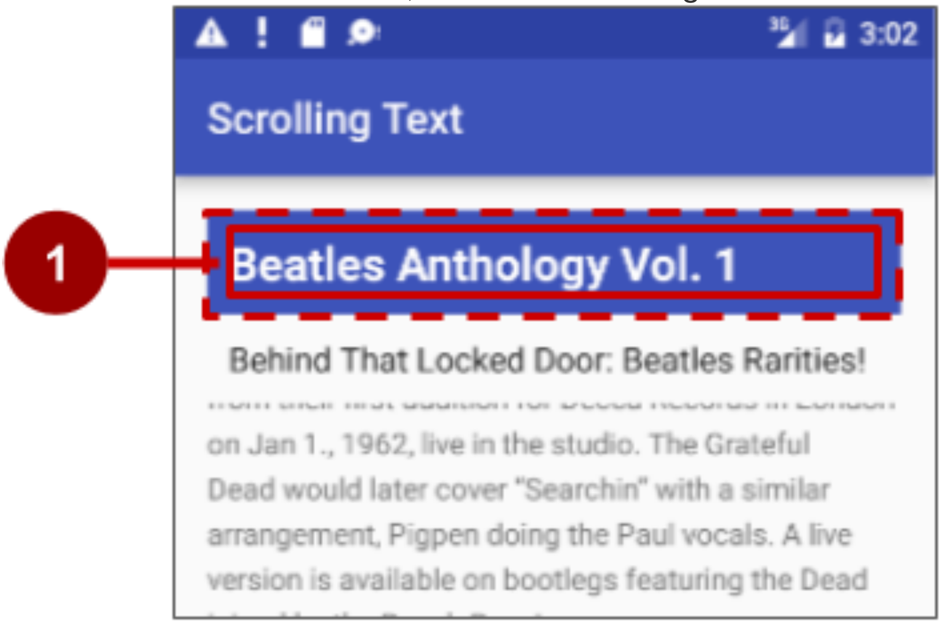
- `match_parent` expands the view to fill its parent by width or height. When the LinearLayout is the root view, it expands to the size of the device screen. For a view within a root view group, it expands to the size of the parent view group.
- `wrap_content` shrinks the view dimensions just big enough 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.

Other layout-related attributes include:

- `Android:layout_gravity`: This attribute is used with a view to control where the view is arranged within its parent view group. For example, the following attribute centers the view horizontally on the screen:
- `android:layout_gravity="center_horizontal"`
- 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



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

below.

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'*thave 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>
```

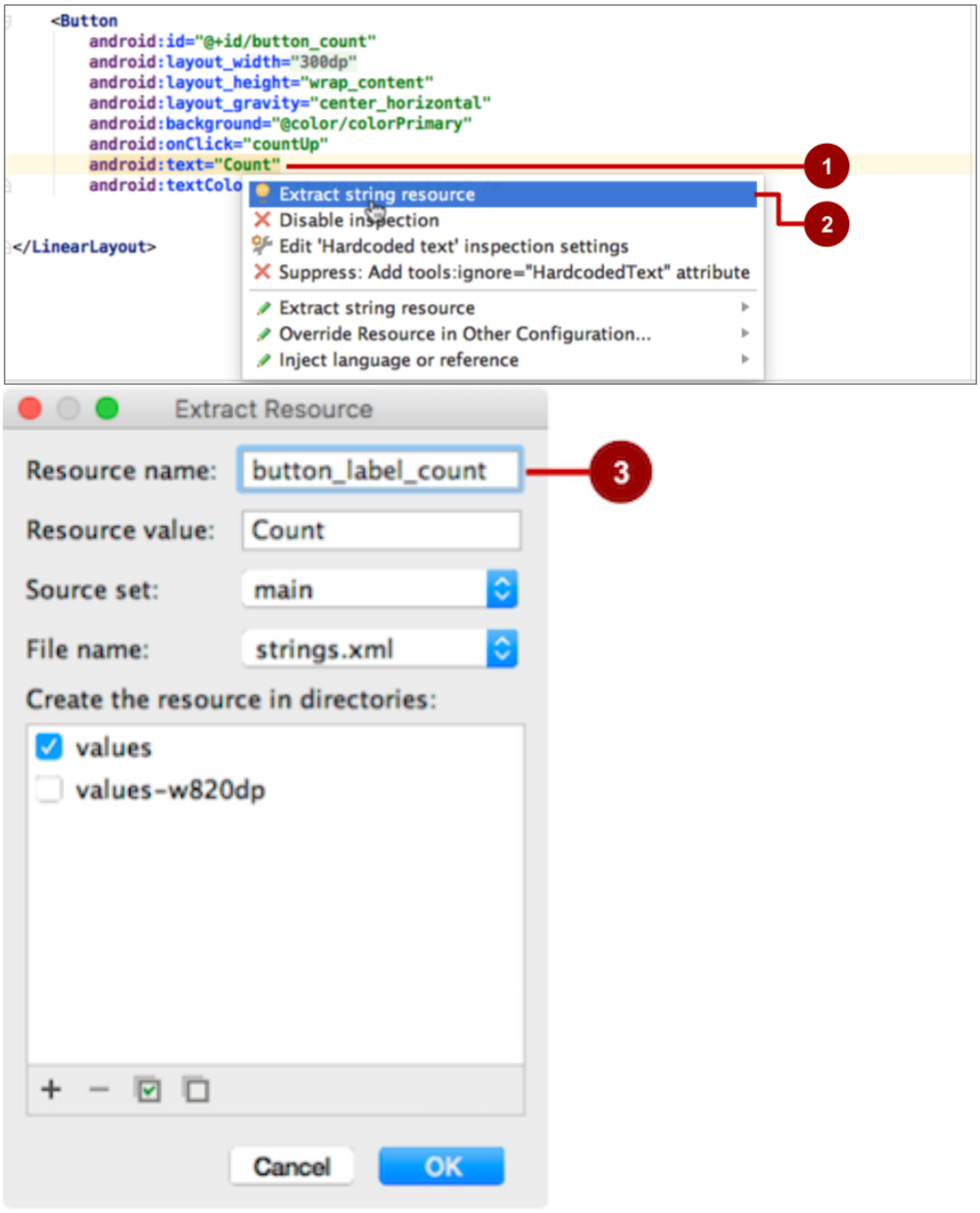
The `name` (for example, `button_label_count`) is the resource name you use in your XML code, as in the following attribute:

```
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:layout_gravity="center_horizontal"
        android:background="@color/colorPrimary"
        android:onClick="countUp"
        android:text="Count"
        android:textColo

</LinearLayout>
```

Resource name: button_label_count

Resource value: Count

Source set: main

File name: strings.xml

Create the resource in directories:

- [x] values
- [ ] values-w820dp

1
2
3
Extract string resource
Disable inspection
Edit 'Hardcoded text' inspection settings
Suppress: Add tools:ignore="HardcodedText" attribute
Extract string resource
Override Resource in Other Configuration...
Inject language or reference
Extract Resource
Cancel
OK

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:

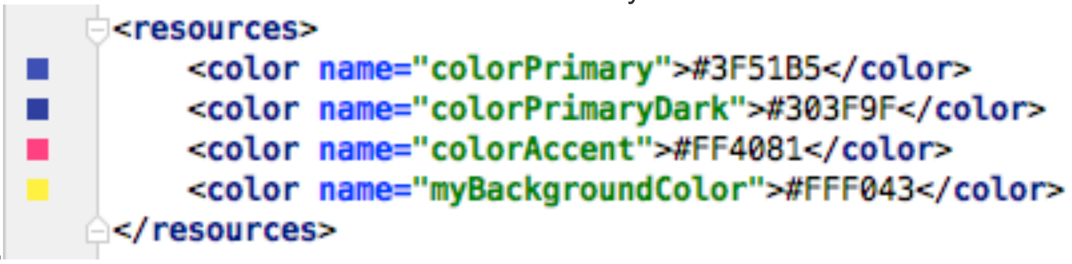
```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
    <color name="myBackgroundColor">#FFF043</color>
</resources>
```

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

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

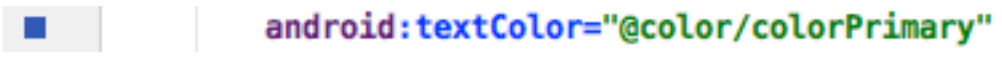
The color value of this `name` is the hexadecimal color value (`#3F51B5`) enclosed within the `<color></color>` tags. The hexadecimal value specifies red, green, and blue (RGB) values. The value always begins with a pound (`#`) character, followed by the Alpha-Red-Green-Blue information. For example, 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



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

file.



```
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.

About activities

An activity represents a single screen in your app with an interface the user can interact with. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading individual messages. Your app is a collection of activities that you either create yourself, or that you reuse from other apps.

Although the activities in your app work together to form a cohesive user experience in your app, each one is independent of the others. This enables your app to start activities in other apps, and other apps can start your activities (if your app allows it). For example, a messaging app you write could start an activity in a camera app to take a picture, and then start the activity in an email app to let the user share that picture in email.

Typically, one activity in an app is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start other activities in order to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes.

When an activity is stopped because a new activity starts, the first activity is notified of that change with the activity's lifecycle callback methods. The Activity lifecycle is the set of states an activity can be in, from when it is first created, to each time it is stopped or resumed, to when the system destroys it. You'll learn more about the activity lifecycle in the next chapter.

Creating activities

To implement an activity in your app, do the following:

- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.

When you create a new project for your app, or add a new activity to your app, in Android Studio (with File > New > Activity), template code for each of these tasks is provided for you.

Create the activity class

Activities are subclasses of the Activity class, or one of its subclasses. When you create a new project in Android Studio, your activities are, by default, subclasses of the AppCompatActivity class. The AppCompatActivity class is a subclass of Activity that lets you to use up-to-date Android app features such as the action bar and material design, while still enabling your app to be compatible with devices running older versions of Android.

Here is a skeleton subclass of AppCompatActivity:

```
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
   }
}
```

The first task for you in your activity subclass is to implement the standard activity lifecycle callback methods (such as OnCreate()) to handle the state changes for your activity. These state changes include things such as when the activity is created, stopped, resumed, or destroyed. You'll learn more about the activity lifecycle and lifecycle callbacks in the next chapter.

The one required callback your app must implement is the onCreate() method. The system calls this method when it creates your activity, and all the essential components of your activity should be initialized here. Most importantly, the OnCreate() method calls setContentView() to create the primary layout for the activity.

You typically define the user interface for your activity in one or more XML layout files. When the setContentView() method is called with the path to a layout file, the system creates all the initial views from the specified layout and adds them to your activity. This is often referred to as *inflating* the layout.

You may often also want to implement the onPause() method in your activity class. The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back). You'll learn more about onPause() and all the other lifecycle callbacks in the next chapter.

In addition to lifecycle callbacks, you may also implement methods in your activity to handle other behavior such as user input or button clicks.

Implement a user interface

the user interface for an activity is provided by a hierarchy of views, which controls a particular space within the activity's window and can respond to user interaction.

The most common way to define a user interface using views is with an XML layout file stored as part of your app's resources. Defining your layout in XML enables you to maintain the design of your user interface separately from the source code that defines the activity's behavior.

You can also create new views directly in your activity code by inserting new view objects into a ViewGroup, and then passing the root ViewGroup to setContentView(). After your layout has been inflated -- regardless of its source -- you can add more views in Java anywhere in the view hierarchy.

Declare the activity in the manifest

Each activity in your app must be declared in the Android app manifest with the `<activity>` element, inside `<application>`. When you create a new project or add a new activity to your project in Android Studio, your manifest is created or updated to include skeleton activity declarations for each activity. Here's the declaration for the main activity.

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

The `<activity>` element includes a number of attributes to define properties of the activity such as its label, icon, or theme. The only required attribute is android:name, which specifies the class name for the activity (such as "MainActivity"). See the `<activity>` element reference for more information on activity declarations.
The `<activity>` element can also include declarations for intent filters. The intent filters specify the kind of intents your activity will accept.

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

Intent filters must include at least one `<action>` element, and can also include a `<category>` and optional `<data>`. The main activity for your app needs an intent filter that defines the "main" action and the "launcher" category so that the system can launch your app. Android Studio creates this intent filter for the main activity in your project:

The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).
Other activities in your app can also declare intent filters, but only your main activity should include the "main" action.. You'll learn more about implicit intents and intent filters in a later section.

Add more activities to your project

The main activity for your app and its associated layout file comes with your project when you create it. You can add new activities to your project in Android Studio with the **File > New > Activity** menu. Choose the activity template you

want to use, or open the Gallery to see all the available templates.

When you choose an activity template, you'll see the same set of screens for creating the new activity that you did when you initially created the project. Android Studio provides these three things for each new activity in your app:

- A Java file for the new activity with a skeleton class definition and onCreate() method. The new activity, like the main activity, is a subclass of AppCompatActivity.
- An XML file containing the layout for the new activity. Note that the setContentView() method in the activity class inflates this new layout.
- An additional `<activity>` element in the Android manifest that specifies the new activity. The second activity definition does not include any intent filters. If you intend to use this activity only within your app (and not enable that activity to be started by any other app), you do not need to add filters.

About intents

All Android activities are started or activated with an *intent*. Intents are message objects that make a request to the Android runtime to start an activity or other app component in your app or in some other app. You don't start those activities yourself;

When your app is first started from the device home screen, the Android runtime sends an intent to your app to start your app's main activity (the one defined with the MAIN action and the LAUNCHER category in the Android Manifest). To start other activities in your app, or request that actions be performed by some other activity available on the device, you build your own intents with the Intent class and call the startActivity() method to send that intent.

In addition to starting activities, intents are also used to pass data between activities. When you create an intent to start a new activity, you can include information about the data you want that new activity to operate on. So, for example, an email activity that displays a list of messages can send an intent to the activity that displays that message. The display activity needs data about the message to display, and you can include that data in the intent.

In this chapter you'll learn about using intents with activities, but intents are also used to start services and broadcast receivers. You'll learn about both those app components later on in the book.

Intent types

There are two types of intents in Android:

- *Explicit intents* specify the receiving activity (or other component) by that activity's fully-qualified class name. Use an explicit intent to start a component in your own app (for example, to move between screens in the user interface), because you already know the package and class name of that component.
- *Implicit intents* do not specify a specific activity or other component to receive the intent. Instead you declare a general action to perform in the intent. The Android system matches your request to an activity or other component that can handle your requested action. You'll learn more about implicit intents in a later chapter.

Intent objects and fields

An Intent object is an instance of the Intent class. For explicit intents, the key fields of an intent include the following:

- The activity *class* (for explicit intents). This is the class name of the activity or other component that should receive the intent, for example, com.example.SampleActivity.class. Use the intent constructor or the intent's setComponent(), setComponentName() or setClassName() methods to specify the class.
- The intent *data*. The intent data field contains a reference to the data you want the receiving activity to operate on, as a Uri object.
- Intent *extras*. These are key-value pairs that carry information the receiving activity requires to accomplish the requested action.
- Intent *flags*. These are additional bits of metadata, defined by the Intent class. The flags may instruct the Android system how to launch an activity or how to treat it after it's launched.

For implicit intents, you may need to also define the intent action and category. You'll learn more about intent actions and categories in section 2.3.

Starting an activity with an explicit intent

To start a specific activity from another activity, use an explicit intent and the startActivity() method. Explicit intents include the fully-qualified class name for the activity or other component in the Intent object. All the other intent fields are optional, and null by default.

For example, if you wanted to start the ShowMessageActivity to show a specific message in an email app, use code like this.

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
startActivity(messageIntent);
```

The Intent constructor takes two arguments for an explicit intent.

- An application context. In this example, the activity class provides the content (here, `this`).
- The specific component to start (`ShowMessageActivity.class`).

Use the startActivity() method with the new intent object as the only argument. The startActivity() method sends the intent to the Android system, which launches the ShowMessageActivity class on behalf of your app. The new activity appears on the screen, and the originating activity is paused.

The started activity remains on the screen until the user taps the back button on the device, at which time that activity closes and is reclaimed by the system, and the originating activity is resumed. You can also manually close the started activity in response to a user action (such as a button click) with the finish() method:

```
public void closeActivity (View view) {
    finish();
}
```

Passing data between activities with intents

In addition to simply starting one activity from another, you also use intents to pass information between activities. The intent object you use to start an activity can include intent *data* (the URI of an object to act on), or intent *extras*, which are bits of additional data the activity might need.

In the first (sending) activity, you:

1. Create the Intent object.
2. Put data or extras into that intent.
3. Start the new activity with startActivity().

In the second (receiving) activity, you:

1. Get the intent object the activity was started with.
2. Retrieve the data or extras from the Intent object.

When to use intent data or intent extras

You can use either intent data and intent extras to pass data between the activities. There are several key differences between data and extras that determine which you should use.

The intent *data* can hold only one piece of information. A URI representing the location of the data you want to operate on. That URI could be a web page URL (http://), a telephone number (tel://), a goegraphic location (geo://) or any other custom URI you define.

Use the intent data field:

- When you only have one piece of information you need to send to the started activity.
- When that information is a data location that can be represented by a URI.

Intent *extras* are for any other arbitrary data you want to pass to the started activity. Intent extras are stored in a Bundle object as key and value pairs. Bundles are a map, optimized for Android, where the keys are strings, and the values can be any primitive or object type (objects must implement the Parcelable interface). To put data into the intent extras you can use any of the Intent class's putExtra() methods, or create your own bundle and put it into the intent with putExtras().

Use the intent extras:

- If you want to pass more than one piece of information to the started activity.
- If any of the information you want to pass is not expressible by a URI.

Intent data and extras are not exclusive; you can use data for a URI and extras for any additional information the started activity needs to process the data in that URI.

Add data to the intent

To add data to an explicit intent from the originating activity, create the intent object as you did before:

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use the setData() method with a Uri object to add that URI to the intent. Some examples of using setData() with URIs:

```
// A web page URL
messageIntent.setData(Uri.parse("http://www.google.com"));
// a Sample file URI
messageIntent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
// A sample content: URI for your app's data model
messageIntent.setData(Uri.parse("content://mysample.provider/data"));
// Custom URI
messageIntent.setData(Uri.parse("custom:" + dataID + buttonId));
```

Keep in mind that the data field can only contain a single URI; if you call setData() multiple times only the last value is used. Use intent extras to include additional information (including URIs.)

After you've added the data, you can start the activity with the intent as usual.

```
startActivity(messageIntent);
```

Add extras to the intent

To add intent extras to an explicit intent from the originating activity:

1. Determine the keys to use for the information you want to put into the extras, or define your own. Each piece of information needs its own unique key.
2. Use the putExtra() methods to add your key/value pairs to the intent extras. Optionally you can create a Bundle object, add your data to the bundle, and then add the bundle to the intent.

The Intent class includes several intent extra keys you can use, defined as constants that begin with the word EXTRA_. For example, you could use Intent.EXTRA_EMAIL to indicate an array of email addresses (as strings), or Intent.EXTRA_REFERRER to specify information about the originating activity that sent the intent.

You can also define your own intent extra keys. Conventionally you define intent extra keys as static variables with names that begin with EXTRA_. To guarantee that the key is unique, the string value for the key itself should be prefixed with your app's fully qualified class name. For example:

```
public final static String EXTRA_MESSAGE = "com.example.mysampleapp.MESSAGE";
public final static String EXTRA_POSITION_X = "com.example.mysampleapp.X";
public final static String EXTRA_POSITION_Y = "com.example.mysampleapp.Y";
```

Create an intent object (if one does not already exist):

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use a putExtra() method with a key to put data into the intent extras. The Intent class defines many putExtra() methods for different kinds of data:

```
messageIntent.putExtra(EXTRA_MESSAGE, "this is my message");
messageIntent.putExtra(EXTRA_POSITION_X, 100);
messageIntent.putExtra(EXTRA_POSITION_Y, 500);
```

Alternately, you can create a new bundle and populate that bundle with your intent extras. Bundle defines many "put" methods for different kinds of primitive data as well as objects that implement Android's Parcelable interface or Java's Serializable.

```
Bundle extras = new Bundle();
extras.putString(EXTRA_MESSAGE, "this is my message");
extras.putInt(EXTRA_POSITION_X, 100);
extras.putInt(EXTRA_POSITION_Y, 500);
```

After you've populated the bundle, add it to the intent with the putExtras() method (note the "s" in Extras):

```
messageIntent.putExtras(extras);
```

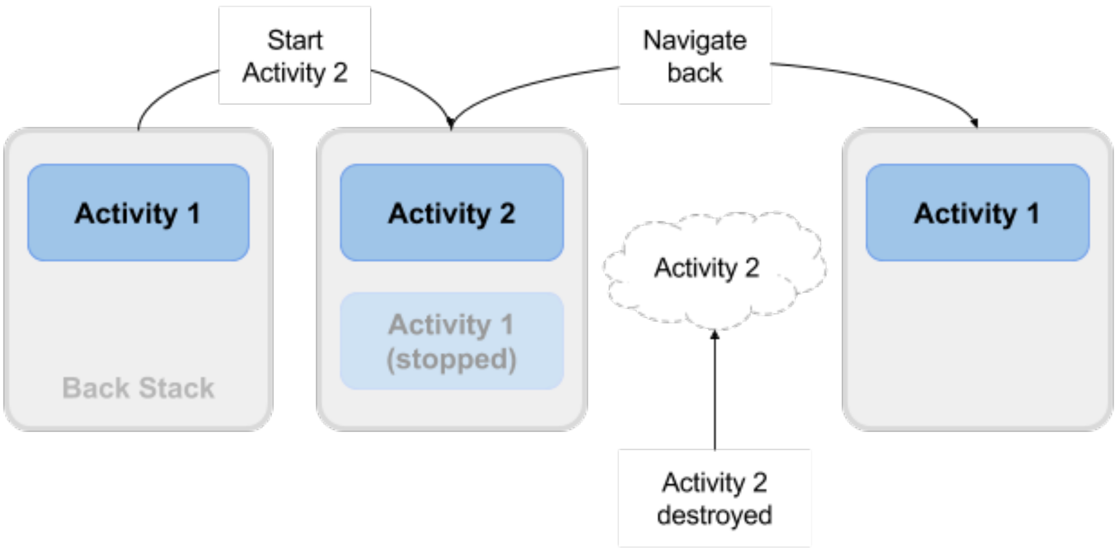
Start the activity with the intent as usual:

```
startActivity(messageIntent);
```

About the activity lifecycle

The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it is initially created to when it is destroyed and the system reclaims that activity's resources. As the user interacts with your app and other apps on the device, the different activities move into different states.

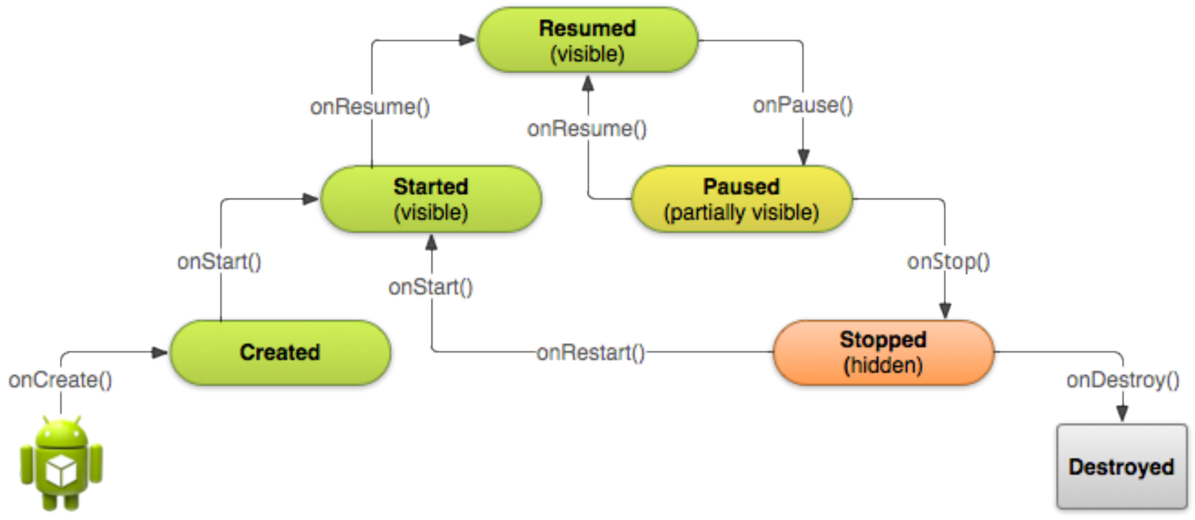
For example, when you start an app, the app's main activity (Activity 1) is started, comes to the foreground, and receives the user focus. When you start a second activity (Activity 2), that new activity is also created and started, and the main activity is stopped. When you're done with the second activity and navigate back, the first activity resumes. The second activity stops and is no longer needed; if the user does not resume the second activity, it is eventually destroyed by the system.



Activity states and lifecycle callback methods

When an activity transitions into and out of the different lifecycle states as it runs, the Android system calls several lifecycle callback methods at each stage. All of the callback methods are hooks that you can override in each of your Activity classes to define how that activity behaves when the user leaves and re-enters the activity. Keep in mind that the lifecycle states (and callbacks) are per activity, not per app, and you may implement different behavior at different points in the lifecycle for different activities in your app.

This figure shows each of the activity states and the callback methods that occur as the activity transitions between different
states:



Depending on the complexity of your activity, you probably don't need to implement all the lifecycle callback methods in your activities. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application.

Activity created (onCreate() method)

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // The activity is being created.
}
```

Your activity enters into the created state when it is started for the first time. When an activity is first created the system calls the onCreate() method to initialize that activity. For example, when the user taps your app icon from the Home screen to start that app, the system calls the onCreate() method for the activity in your app that you've declared to be the "launcher" or "main" activity. In this case the main activity's onCreate() method is analogous to the main() method in other programs.

Similarly, if your app starts another activity with an intent (either explicit or implicit), the system matches your intent request with an activity and calls onCreate() for that new activity.

The onCreate() method is the only required callback you must implement in your activity class. In your onCreate() method you perform basic application startup logic that should happen only once, such as setting up the user interface, assigning class-scope variables, or setting up background tasks.

Created is a transient state; the activity remains in the created state only as long as it takes to run onCreate(), and then the activity moves to the started state.

Activity started (onStart() method)

```
@Override
protected void onStart() {
    super.onStart();
    // The activity is about to become visible.
}
```

After your activity is initialized with onCreate(), the system calls the onStart() method, and the activity is in the started state. The onStart() method is also called if a stopped activity returns to the foreground, such as when the user clicks the back or up buttons to navigate to the previous screen. While onCreate() is called only once when the activity is created, the onStart() method may be called many times during the lifecycle of the activity as the user navigates around your app.

When an activity is in the started state and visible on the screen, the user cannot interact with it until onResume() is called, the activity is running, and the activity is in the foreground.

Typically you implement onStart() in your activity as a counterpart to the onStop() method. For example, if you release hardware resources (such as GPS or sensors) when the activity is stopped, you can re-register those resources in the onStart() method.

Started, like created, is a transient state. After starting the activity moves into the resumed (running) state.

Activity resumed/running (onResume() method)

```
@Override
protected void onResume() {
    super.onResume();
    // The activity has become visible (it is now "resumed").
}
```

Your activity is in the resumed state when it is initialized, visible on screen, and ready to use. The resumed state is often called the running state, because it is in this state that the user is actually interacting with your app.

The first time the activity is started the system calls the onResume() method just after onStart(). The onResume() method may also be called multiple times, each time the app comes back from the paused state.

As with the onStart() and onStop() methods, which are implemented in pairs, you typically only implement onResume() as a counterpart to onPause(). For example, if in the onPause() method you halt any onscreen animations, you would start those animations again in onResume().

The activity remains in the resumed state as long as the activity is in the foreground and the user is interacting with it. From the resumed state the activity can move into the paused state.

Activity paused (onPause() method)

```
@Override
protected void onPause() {
    super.onPause();
    // Another activity is taking focus
    // (this activity is about to be "paused").
}
```

The paused state can occur in several situations:

- The activity is going into the background, but has not yet been fully stopped. This is the first indication that the user is leaving your activity.
- The activity is only partially visible on the screen, because a dialog or other transparent activity is overlaid on top of it.
- In multi-window or split screen mode (API 24), the activity is displayed on the screen, but some other activity has the user focus.

The system calls the onPause() method when the activity moves into the paused state. Because the onPause() method is the first indication you get that the user may be leaving the activity, you can use onPause() to stop animation or video playback, release any hardware-intensive resources, or commit unsaved activity changes (such as a draft email).

The onPause() method should execute quickly. Don't use onPause() for for CPU-intensive operations such as writing persistent data to a database. The app may still be visible on screen as it passed through the paused state, and any delays in executing onPause() can slow the user's transition to the next activity. Implement any heavy-load operations when the app is in the stopped state instead.

Note that in multi-window mode (API 24), your paused activity may still fully visible on the screen. In this case you do not want to pause animations or video playback as you would for a partially visible activity. You can use the inMultiWindowMode() method in the Activity class to test whether your app is running in multiwindow mode.

Your activity can move from the paused state into the resumed state (if the user returns to the activity) or to the stopped state (if the user leaves the activity altogether).

Activity stopped (onStop() method)

```
@Override
protected void onStop() {
    super.onStop();
    // The activity is no longer visible (it is now "stopped")
}
```

An activity is in the stopped state when it is no longer visible on the screen at all. This is usually because the user has started another activity, or returned to the home screen. The system retains the activity instance in the back stack, and if the user returns to that activity it is restarted again. Stopped activities may be killed altogether by the Android system if resources are low.

The system calls the onStop() method when the activity stops. Implement the onStop() method to save any persistent data and release any remaining resources you did not already release in onPause(), including those operations that may have been too heavyweight for onPause().

Activity destroyed (onDestroy() method)

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // The activity is about to be destroyed.
}
```

When your activity is destroyed it is shut down completely, and the Activity instance is reclaimed by the system. This can happen in several cases:

- You call finish() in your activity to manually shut it down.
- The user navigates back to the previous activity.
- The device is in a low memory situation where the system reclaims stopped activities to free more resources.
- A device configuration change occurs. You'll learn more about configuration changes later in this chapter.

Use onDestroy() to fully clean up after your activity so that no component (such as a thread) is running after the activity is destroyed.

Note that there are situations where the system will simply kill the activity's hosting process without calling this method (or any others), so you should not rely on onDestroy() to save any required data or activity state. Use onPause() or onStop() instead.

Activity restarted (onRestart() method)

```
@Override
protected void onRestart() {
    super.onRestart();
    // The activity is about to be restarted.
}
```

The restarted state is a transient state that only occurs if a stopped activity is started again. In this case the onRestart() method is called in between onStop() and onStart(). If you have resources that need to be stopped or started you typically implement that behavior in onStop() or onStart() rather than onRestart().