Mobile Application Development

Development of an Android application, Activities and Intent

(Professional Android 4 Application Development chapter 3)

Outline

- Building blocks of Android application
- The application manifest file, Manifest editor, Creating resources, Using resources
- Android application Lifecycle
- Application's Priority and its process states
- Extending and using Android Application class
- Overriding the Application Lifecycle events
- Android activities: Creating Activities, Activity Lifecycle, Activity Stacks,
- Activity States, Monitoring State Changes, Understanding Activity Lifetimes, Android Activity classes
- Overview of an Intent, Starting Activities, Sub activities, and Services using implicit and explicit Intents

Building Blocks of Android Application

- Activities — Your application's presentation layer.
  - The UI of your application is built around one or more extensions of the Activity class.
  - Activities use Fragments and Views to layout and display information, and to respond to user actions.
  - Compared to desktop development, Activities are equivalent to Forms.

Building Blocks of Android Application

- Services — The invisible workers of your application.
  - Service components run without a UI, updating your data sources and Activities, triggering Notifications, and broadcasting Intents.
  - They're used to perform long running tasks, or those that require no user interaction (such as network lookups or tasks that need to continue even when your application's Activities aren't active or visible.)

Building Blocks of Android Application

- Content Providers — Shareable persistent data storage.
  - Content Providers manage and persist application data and typically interact with SQL databases. They're also the preferred means to share data across application boundaries.
  - You can configure your application's Content Providers to allow access from other applications, and you can access the Content Providers exposed by others.
  - Android devices include several native Content Providers that expose useful databases such as the media store and contacts.

Building Blocks of Android Application

- Intents — A powerful interapplication message-passing framework.
  - Intents are used extensively throughout Android.
  - You can use Intents to start and stop Activities and Services, to broadcast messages system-wide or to an explicit Activity, Service, or Broadcast Receiver, or to request an action be performed on a particular piece of data.
  - Types of intents are Explicit, implicit, and broadcast.

Building Blocks of Android Application

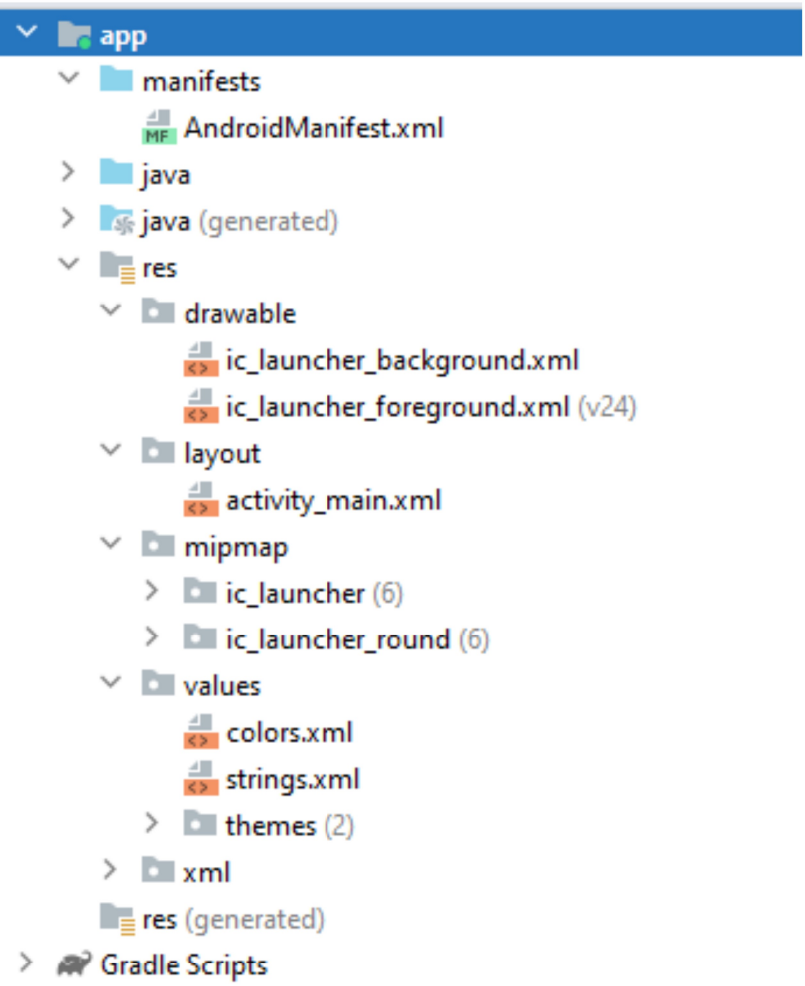
- Broadcast Receivers — Intent listeners.
  - Broadcast Receivers enable your application to listen for Intents that match the criteria you specify.
  - Broadcast Receivers start your application to react to any received Intent, making them perfect for creating event-driven applications.

Building Blocks of Android Application

- Widgets — Visual application components that are typically added to the device home screen.
  - A special variation of a Broadcast Receiver, widgets enable you to create dynamic, interactive application components for users to embed on their home screens.

Building Blocks of Android Application

- Notifications — Notifications enable you to alert users to application events without stealing focus or interrupting their current Activity.
  - They're the preferred technique for getting a user's attention when your application is not visible or active, particularly from within a Service or Broadcast Receiver.
  - For example, when a device receives a text message or an email, the messaging and Gmail applications use Notifications to alert you by flashing lights, playing sounds, displaying icons, and scrolling a text summary.



Project Structure

The Application Manifest File

- Each Android project includes a manifest file, AndroidManifest.xml, stored in the root of its project hierarchy.
- The manifest defines the structure and metadata of your application, its components, and its requirements.
- It includes nodes for each of the Activities, Services, Content Providers, and Broadcast Receivers that make up your application and, using Intent Filters and Permissions, determines how they interact with each other and with other applications.
- The manifest can also specify application metadata (such as its icon, version number, or theme), and additional top-level nodes can specify any required permissions, unit tests, and define hardware, screen, or platform requirements.

The Application Manifest File

- The manifest is made up of a root **manifest** tag with a **package** attribute set to the project's package.
- It should also include an **xmlns:android** attribute that supplies several system attributes used within the file.
- Use the **versionCode** attribute to define the current application version as an integer that increases with each version iteration, and use the **versionName** attribute to specify a public version that will be displayed to users.
- You can also specify whether to allow (or prefer) for your application be installed on external storage (usually an SD card) rather than internal storage using the **installLocation** attribute.
- To do this specify either **preferExternal** or **auto**, where the former installs to external storage whenever possible, and the latter asks the system to decide.

Application Manifest

- The following XML snippet shows a typical manifest node:

```
<manifest xmlns:android=”http://schemas.android.com/apk/res/android”
    package=“com.paad.myapp“
    android:versionCode=“1“
    android:versionName=“0.9 Beta“
    android:installLocation=“preferExternal“>
    [ ... manifest nodes ... ]
</manifest>
```


Application Manifest

- The manifest tag can include nodes that defi ne the application components, security settings, test classes, and requirements that make up your application.
- The following list gives a summary of the available manifest sub-node tags and provides an XML snippet demonstrating how each tag is used.

Application Manifest

- uses-sdk — This node enables you to defi ne a minimum and maximum SDK version that must be available on a device for your application to function properly, and target SDK for which it has been designed using a combination of minSDKVersion, maxSDKVersion, and targetSDKVersion attributes, respectively.
- <uses-sdk android:minSdkVersion=”6”
  android:targetSdkVersion=”15”/>

Application Manifest

- uses-configuration — The uses-configuration nodes specify each combination of input mechanisms are supported by your application.
- You shouldn’t normally need to include this node, though it can be useful for games that require particular input controls.
- <uses-configuration android:reqTouchScreen=”finger”
  android:reqNavigation=”trackball”
  android:reqHardKeyboard=”true”
  android:reqKeyboardType=”qwerty”/>

Application Manifest

- uses-feature — Android is available on a wide variety of hardware platforms. Use multiple uses-feature nodes to specify which hardware features your application requires.
- This prevents your application from being installed on a device that does not include a required piece of hardware, such as NFC hardware, as follows:
- <uses-feature android:name=”android.hardware.nfc” />

Application Manifest

- The supports-screen node enables you to specify the screen sizes your application has been designed and tested to.
- <supports-screens android:smallScreens=”false”
  android:normalScreens=”true”
  android:largeScreens=”true”
  android:xlargeScreens=”true”
  android:requiresSmallestWidthDp=”480”
  android:compatibleWidthLimitDp=”600”
  android:largestWidthLimitDp=”720”/>

Application Manifest

- supports-gl-texture — Declares that the application is capable of providing texture assets that are compressed using a particular GL texture compression format.
- You must use multiple supports-gl-texture elements if your application is capable of supporting multiple texture compression formats.
- <supports-gl-texture android:name=”GL_OES_compressed_ETC1_RGB8_texture” />

Application Manifest

- uses-permission — As part of the security model, uses-permission tags declare the user permissions your application requires.
- Each permission you specify will be presented to the user before the application is installed.
- <uses-permission android:name=”android.permission.ACCESS_FINE_LOCATION”/>

Application Manifest

- instrumentation — Instrumentation classes provide a test framework for your application components at run time.
- They provide hooks to monitor your application and its interaction with the system resources.
- Create a new node for each of the test classes you've created for your application.
- <instrumentation android:label="My Test"
  android:name=".MyTestClass"
  android:targetPackage="com.paad.apackage">
- </instrumentation>

Application Manifest

- application — A manifest can contain only one application node. It uses attributes to specify the metadata for your application (including its title, icon, and theme).
- During development you should include a debuggable attribute set to true to enable debugging, then be sure to disable it for your release builds.
- <application android:icon="@drawable/icon"
  android:logo="@drawable/logo"
  android:theme="@android:style/Theme.Light"
  android:name=".MyApplicationClass"
  android:debuggable="true">
  [... application nodes ...]
- </application>

Application Manifest

- activity — An activity tag is required for every Activity within your application.
- Use the android:name attribute to specify the Activity class name. You
- must include the main launch Activity and any other Activity that may be displayed.
- Trying to start an Activity that's not defi ned in the manifest will throw a runtime exception.
- Each Activity node supports intent-filter child tags that defi ne the Intents that can be used to start the Activity.

Application Manifest

- Note that a period is used as shorthand for the application's package name when specifying the Activity's class name.
- <activity android:name=".MyActivity" android:label="@string/app_name">
- <intent-filter>

  <action android:name="android.intent.action.MAIN" />

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

Application Manifest

- service — As with the activity tag, add a service tag for each Service class used in your application. Service tags also support intent-filter child tags to allow late runtime binding.
- <service android:name=".MyService">
- </service>

Application Manifest

- provider — Provider tags specify each of your application's Content Providers. Content Providers are used to manage database access and sharing.
- <provider android:name=".MyContentProvider"
- android:authorities="com.paad.myapp.MyContentProvider"/>

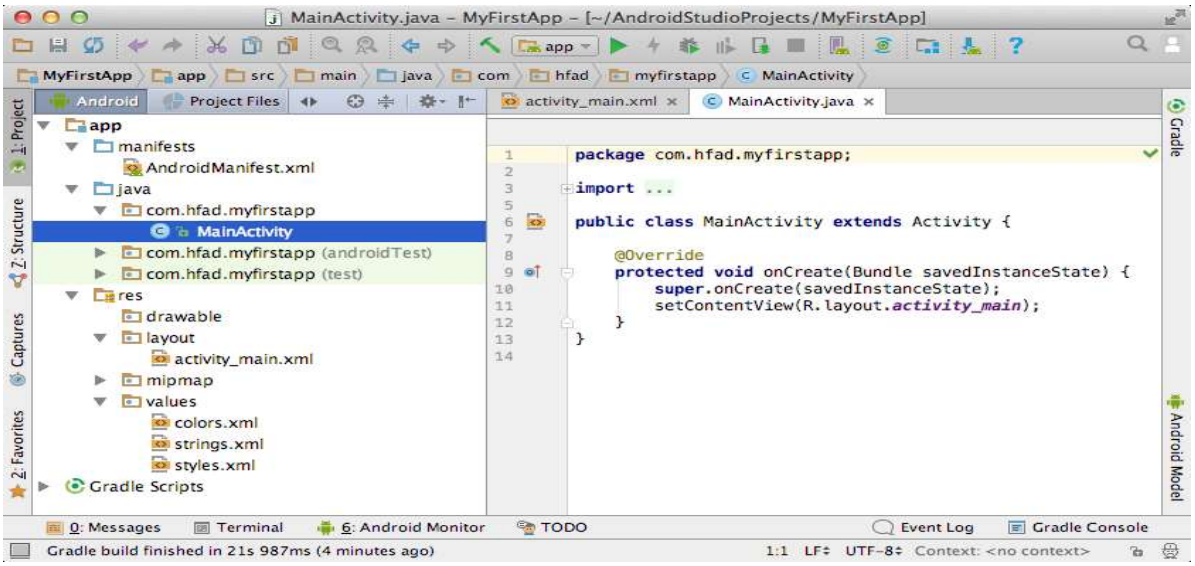
Application Manifest

- receiver — By adding a receiver tag, you can register a Broadcast Receiver without having to launch your application first.
- Broadcast Receivers are like global event listeners that, when registered, will execute whenever a matching Intent is broadcast by the system or an application.
- <receiver android:name=".MyIntentReceiver">
  <intent-filter>
    <action android:name="com.paad.mybroadcastaction" />
  </intent-filter>
- </receiver>

Application Manifest

- uses-library — Used to specify a shared library that this application requires.
- <uses-library android:name="com.google.android.maps"
  android:required="false"/>

Manifest Editor

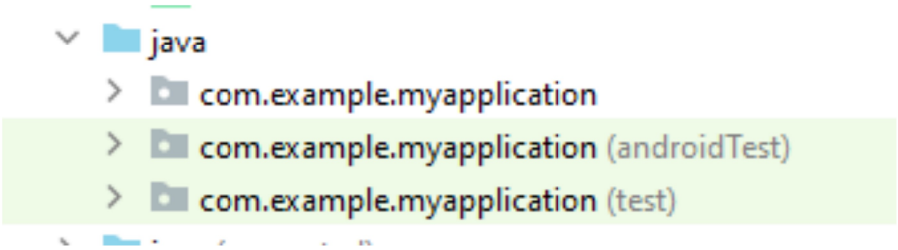



```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.myapplication">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="My Application"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyApplication"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

Java File

- The Java folder contains the Java source code files. These files are used as a controller for controlled UI (Layout file).

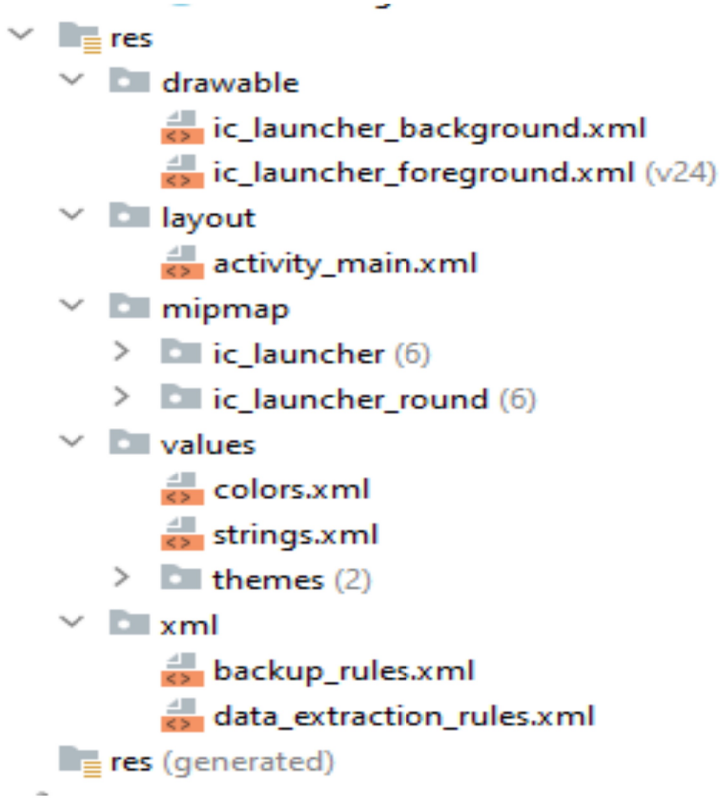
- It gets the data from the Layout file and after processing that data output will be shown in the UI layout. It works on the backend of an Android application.



Creating Resources

- Application resources are stored under the **res** folder in your project hierarchy.
- Each of the available resource types is stored in subfolders, grouped by resource type.
- If you start a project using the ADT Wizard, it creates a res folder that contains subfolders for the values, drawable-ldpi, drawable-mdpi, drawable-hdpi, and layout resources that contain the default string resource definitions, application icon, and layouts respectively, as shown in Figure

Creating Resources

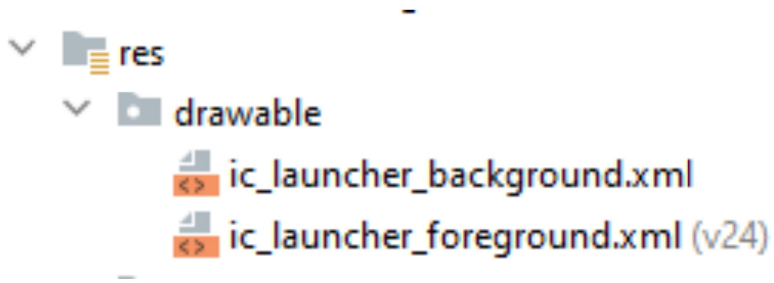


Creating Resources

- Each resource type is stored in a different folder: simple values, Drawables, colors, layouts, animations, styles, menus, XML files,and raw resources.
- When your application is built, these resources will be compiled and compressed as efficiently as possible and included in your application package.
- This process also generates an R class file that contains references to each of the resources you include in your project.

Drawables

- Drawable resources include bitmaps and NinePatches (stretchable PNG images). They also include complex composite Drawables, such as LevelListDrawables and StateListDrawables, that can be defined in XML.
- All Drawables are stored as individual files in the **res/drawable** folder.



Layouts

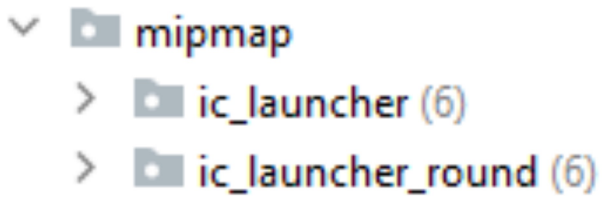
- Layout resources enable you to decouple your presentation layer from your business logic by designing UI layouts in XML rather than constructing them in code.
- You can use layouts to define the UI for any visual component, including Activities, Fragments, and Widgets.
- Within an Activity this is done using **setContentView** (usually within the onCreate method)
- Each layout definition is stored in a separate file, each containing a single layout, in the **res/layout** folder. The filename then becomes resource identifier.

Layouts of Hello World Example

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res
    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>
```


mipmap

- Mipmap folder contains the Image Asset file that can be used in Android Studio application.
- You can generate the following icon types like Launcher icons, Action bar and tab icons, and Notification icons.



Simple Values (string)

- The strings.xml file contains string resources of the Android application. The different string value is identified by a unique name that can be used in the Android application program.

```
<resources>
    <string name="app_name">My Application</string>
</resources>
```

- String resources are specified with the **string** tag, as shown in the above XML snippet:
- Android supports simple text styling, so you can use the HTML tags <b>, <i>, and <u> to apply bold, italics, or underlining, respectively, to parts of your text strings, as shown in the following example:

Simple Values

- Supported simple values include strings, colors, dimensions, styles, and string or integer arrays.
- All simple values are stored within XML files in the res/values folder.
- Within each XML file, you indicate the type of value being stored using tags, as shown in the sample XML file in Listing

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">To Do List</string>
  <plurals name="androidPlural">
    <item quantity="one">One android</item>
    <item quantity="other">%d androids</item>
  </plurals>
  <color name="app_background">#FF0000FF</color>
  <dimen name="default_border">5px</dimen>
  <string-array name="string_array">
    <item>Item 1</item>
    <item>Item 2</item>
    <item>Item 3</item>
  </string-array>
  <array name="integer_array">
    <item>3</item>
    <item>2</item>
    <item>1</item>
  </array>
</resources>
```

Simple Values (string)

- The strings.xml file contains string resources of the Android application. The different string value is identified by a unique name that can be used in the Android application program.

```
<resources>
    <string name="app_name">My Application</string>
</resources>
```

- String resources are specified with the **string** tag, as shown in the above XML snippet:
- Android supports simple text styling, so you can use the HTML tags <b>, <i>, and <u> to apply bold, italics, or underlining, respectively, to parts of your text strings, as shown in the following example:

Simple Values (color)

- Use the **color** tag to define a new color resource. Specify the color value using a # symbol followed by the (optional) alpha channel, and then the red, green, and blue values using one or two hexadecimal numbers with any of the following notations:
  - #RGB
  - #RRGGBB
  - #ARGB
  - #AARRGGBB
- The following example shows how to specify a fully opaque blue and a partially transparent green:
- <color name=”opaque_blue”>#00F</color>
- <color name=”transparent_green”>#7700FF00</color>

Simple Values (color)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```


Simple Values (dimension)

- Dimensions are most commonly referenced within style and layout resources. They're useful for creating layout constants, such as borders and font heights.
- To specify a dimension resource, use the **dimen** tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:
- px (screen pixels) - corresponds to actual pixels on the screen.
- in (physical inches) - based on the physical size of the screen.
- pt (physical points) - 1/72 of an inch based on the physical size of the screen.
- mm (physical millimeters) - based on the physical size of the screen.

Simple Values (dimension)

- dp (density-independent pixels) - Density Independent Pixel, it varies based on screen density . In 160 dpi screen, 1 dp = 1 pixel. Except for font size, use dp always.

- sp (scale-independent pixels) - Scale Independent Pixel, scaled based on user's font size preference. Fonts should use sp.

Simple Values (dimension)

- The following XML snippet shows how to specify dimension values for a large font size and a standard border:
- <dimen name="standard_border">5dp</dimen>
- <dimen name="large_font_size">16sp</dimen>

Styles and Themes

- Style resources let your applications maintain a consistent look and feel by enabling you to specify the attribute values used by Views.
- The most common use of themes and styles is to store the colors and fonts for an application.
- To create a style, use a **style** tag that includes a **name** attribute and contains one or more **item** tags.
- Each **item** tag should include a **name** attribute used to specify the attribute (such as font size or color) being defined.

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <style name=”base_text”>
    <item name=”android:textSize”>14sp</item>
    <item name=”android:textColor”>#111</item>
    </style>
</resources>
```


Styles and Themes

- Styles support inheritance using the parent attribute on the style tag, making it easy to create simple variations:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <style name=”small_text” parent=”base_text”>
    <item name=”android:textSize”>8sp</item>
    </style>
</resources>
```

Styles and Themes

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.MyApplication" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```


Gradle Scripts

- Gradle is a build system (open source) that is used to automate building, testing, deployment, etc. “build.gradle” are scripts where one can automate the tasks.
- Every Android project needs a Gradle for generating an apk from the .java and .xml files in the project.
- There are two types of build.gradle scripts

- Top-level build.gradle
- Module-level build.gradle

Gradle Scripts

- **Top-level build.gradle:**
- It is located in the root project directory and its main function is to define the build configurations that will be applied to all the modules in the project.

- **Module-level build.gradle:**
- Located in the project/module directory of the project this Gradle script is where all the dependencies are defined and where the SDK versions are declared.

Animations

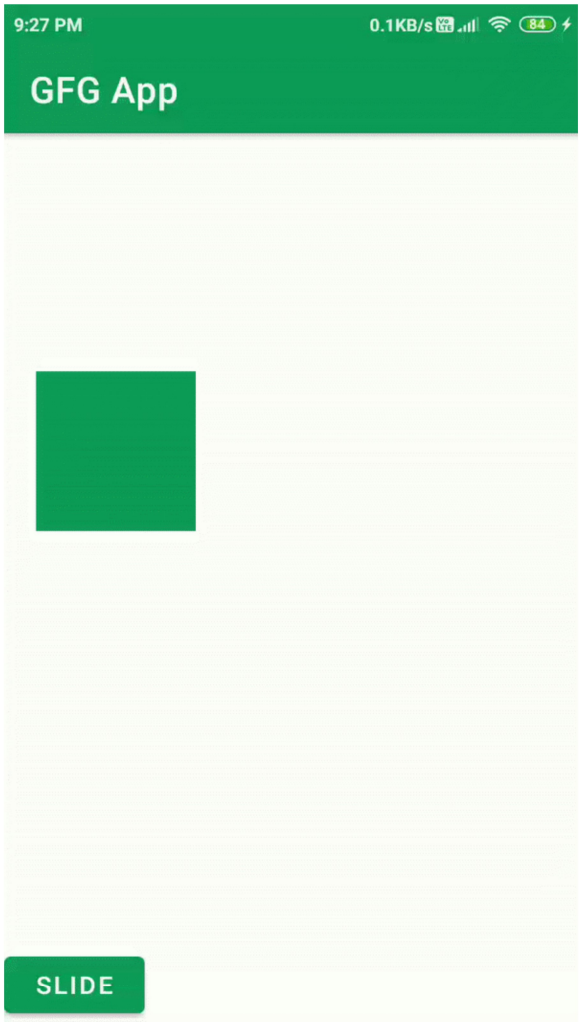
- Animation is the process of adding a motion effect to any view, image, or text.

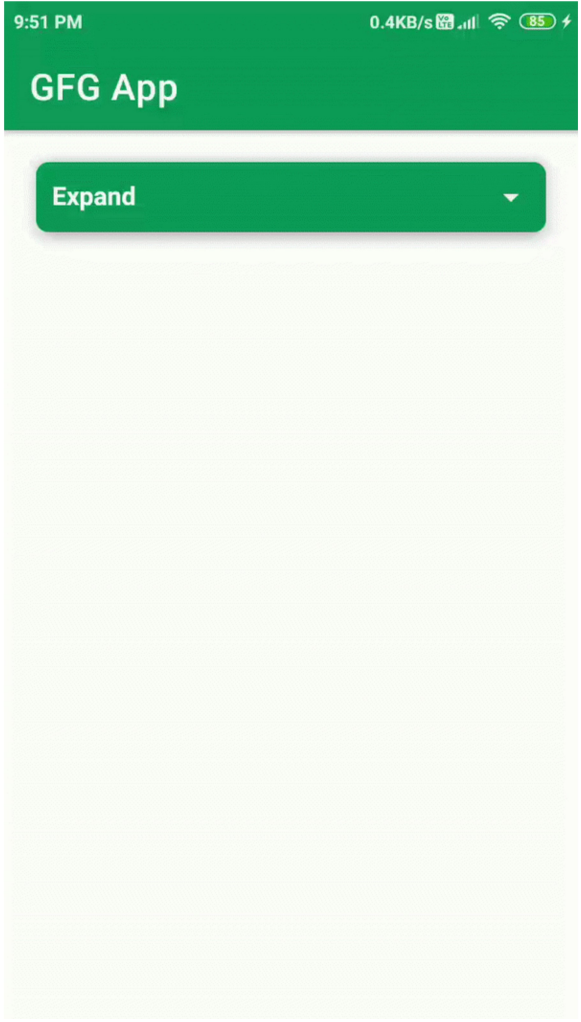
- With the help of an animation, you can add motion or can change the shape of a specific view.
-

  Animation in Android is generally used to give your UI a rich look and feel. The animations are basically of three types as follows:

Animations

- Android supports three types of animation:
  - Property animations — A tweened animation that can be used to potentially animate any property on the target object by applying incremental changes between two values. This can be used for anything from changing the color or opacity of a View to gradually fade it in or out, to changing a font size, or increasing a character's hit points.
  - View animations — Tweened animations that can be applied to rotate, move, and stretch a View.
  - Frame animations — Frame-by-frame "cell" animations used to display a sequence of Drawable images.





Animations

Android Application Lifecycle

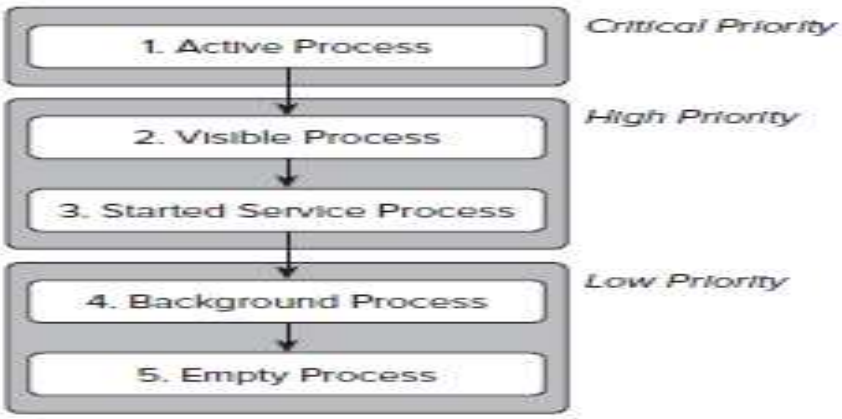
- Android applications have limited control over their own lifecycles.
- Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination.
- By default, each Android application runs in its own process, each of which is running a separate instance of Dalvik.
- Memory and process management is handled exclusively by the run time.

Android Application Priority

- The order in which processes are killed to reclaim resources is determined by the priority of their hosted applications.
- An application's priority is equal to that of its highest-priority component.
- If two applications have the same priority, the process that has been at that priority longest will be killed first.

Android Application Priority

- Figure shows the priority tree used to determine the order of application termination.



Android Application Priority

- **Active processes** — Active (foreground) processes have application components the user is interacting with. These are the processes Android tries to keep responsive by reclaiming resources from other applications. There are generally very few of these processes, and they will be killed only as a last resort.
- Active processes include the following:
  - **Activities in an active state** — that is, those in the foreground responding to user events.
    - Broadcast Receivers executing onReceive event handlers.
    - Services executing onStart, onCreate, or onDestroy event handlers.
    - Running Services that have been flagged to run in the foreground.

Android Application Priority

- Example: Imagine the user is using Whatsapp, so the Whatsapp app will be said to be in the foreground state. This process is of the highest priority and they can only be killed by the system if the memory is so low that even this process cannot continue their execution.

Android Application Priority

- **Visible processes** — Visible but inactive processes are those hosting "visible" Activities.
  - As the name suggests, visible Activities are visible, but they aren't in the foreground or responding to user events.
  - Example: When some application needs permission like camera access, storage access, etc a prompt or dialog box will appear and ask for the required permission. So at this time, the process corresponding to the activity of the app which is running previously will go in the visible state.

Android Application Priority

- **Started Service processes** — Processes hosting Services that have been started. Because these Services don't interact directly with the user, they receive a slightly lower priority than visible Activities or foreground Services.
- Applications with running Services are still considered foreground processes and won't be killed unless resources are needed for active or visible processes.
- Example: Uploading a PDF on the Whatsapp from the desktop is a service process that is done in the background.

Android Application Priority

- **Background processes** — Processes hosting Activities that aren't visible and that don't have any running Services.
  - There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern in order to obtain resources for foreground processes.
  - Example: user is using an app and suddenly presses the home button, so because of this action, the process goes from foreground state to background state.
- **Empty processes** — To improve overall system performance, Android will often retain an application in memory after it has reached the end of its lifetime.
  - Android maintains this cache to improve the start-up time of applications when they're relaunched.

Android Application Class

- Your application's Application object remains instantiated whenever your application runs.
- Unlike Activities, the Application is not restarted as a result of configuration changes.
- Extending the Application class with your own implementation enables you to do three things:
  - Respond to application level events broadcast by the Android run time such as low memory conditions.
  - Transfer objects between application components.
  - Manage and maintain resources used by several application components.

Extending and Using the Application Class

- Your application's Application object remains instantiated whenever your application runs.
- Below shows the skeleton code for extending the Application class and implementing it as a singleton.

```
import android.app.Application;
import android.content.res.Configuration;

public class MyApplication extends Application {

  private static MyApplication singleton;

  // Returns the application instance
  public static MyApplication getInstance() {
    return singleton;
  }

  @Override
  public final void onCreate() {
    super.onCreate();
    singleton = this;
  }
}
```

Extending and Using the Application Class

- When created, you must register your new **Application** class in the manifest's application node using a **name** attribute, as shown in the following snippet:

<application android:icon="@drawable/icon"
android:name="**.MyApplication**">

[... Manifest nodes ...]

</application>

- Your Application implementation will be instantiated when your application is started. Create new state variables and global resources for access from within the application components:

```
MyObject value =MyApplication.getInstance().getGlobalStateValue();
MyApplication.getInstance().setGlobalStateValue(myObjectValue);
```


Overriding Application Lifecycle Event

- **onCreate** — Called when the application is created. Override this method to initialize your application singleton and create and initialize any application state variables or shared resources.
- **onLowMemory** — Provides an opportunity for well-behaved applications to free additional memory when the system is running low on resources. This will generally only be called when background processes have already been terminated and the current foreground applications are still low on memory. Override this handler to clear caches or release unnecessary resources.

Overriding Application Lifecycle Event

- **onTrimMemory** — An application specific alternative to the onLowMemory handler introduced in Android 4.0 (API level 13). Called when the run time determines that the current application should attempt to trim its memory overhead – typically when it moves to the background.
- It includes a level parameter that provides the context around the request.
- **onConfigurationChanged** — Unlike Activities Application objects are not restarted due to configuration changes. If your application uses values dependent on specific configurations, override this handler to reload those values and otherwise handle configuration changes at an application level.

Activity States

- **Active —** When an Activity is at the top of the stack it is the visible, focused, foreground Activity that is receiving user input.
  - Android will attempt to keep it alive at all costs, killing Activities further down the stack as needed, to ensure that it has the resources it needs.
  - When another Activity becomes active, this one will be paused.

Activity States

- **Paused** — In some cases your Activity will be visible but will not have focus; at this point it's paused.
  - This state is reached if a transparent or non-full-screen Activity is active in front of it.
  - When paused, an Activity is treated as if it were active; however, it doesn't receive user input events.
  - In extreme cases Android will kill a paused Activity to recover resources for the active Activity.

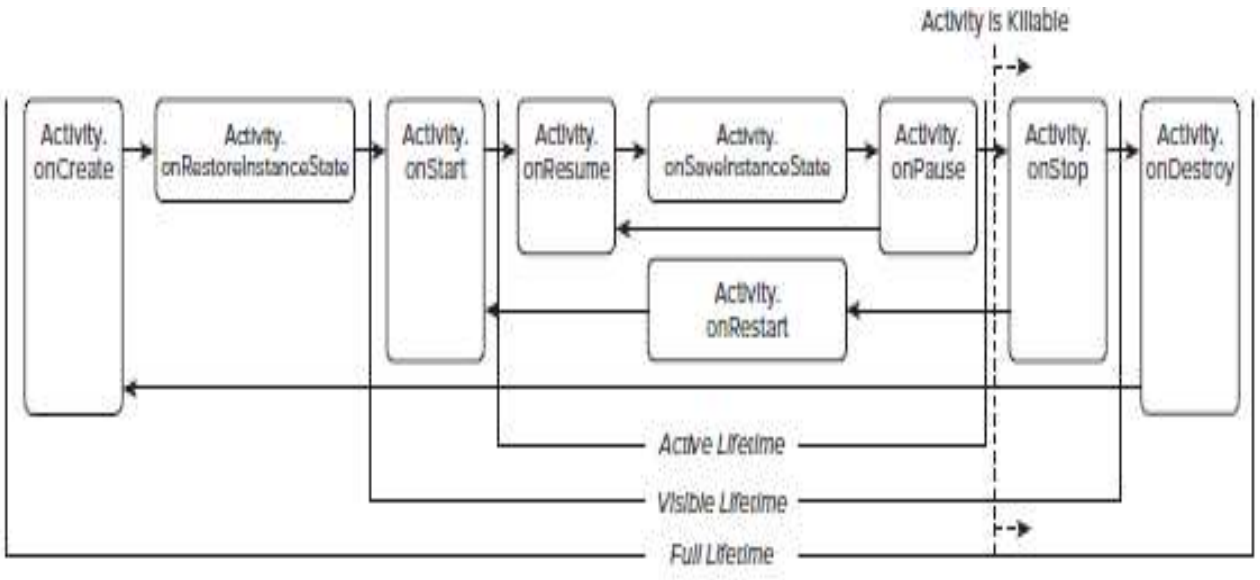
Activity States

- **Stopped** — When an Activity isn't visible, it "stops."
  - The Activity will remain in memory, retaining all state information; however, it is now a candidate for termination when the system requires memory elsewhere.
  - When an Activity is in a stopped state, it's important to save data and the current UI state, and to stop any non-critical operations.
  - Once an Activity has exited or closed, it becomes inactive.

Activity States

- **Inactive** — After an Activity has been killed, and before it's been launched, it's inactive.
  - Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used.

Monitoring States Changes



Understanding Activity Lifetimes

- **The Full Lifetime**
- The full lifetime of your Activity occurs between the first call to onCreate and the final call to onDestroy.
- It's not uncommon for an Activity's process to be terminated without the onDestroy method being called.
- Use the onCreate method to initialize your Activity: inflate the user interface, get references to Fragments, allocate references to class variables, bind data to controls, and start Services and Timers.
- If the Activity was terminated unexpectedly by the runtime, the onCreate method is passed a Bundle object containing the state saved in the last call to onSaveInstanceState.
- You should use this Bundle to restore the UI to its previous state, either within the onCreate method or onRestoreInstanceState.

Understanding Activity Lifetimes

- **The Visible Lifetime**
- An Activity's visible lifetimes are bound between calls to onStart and onStop. Between these calls your Activity will be visible to the user, although it may not have focus or not.
- Although it's unusual, in extreme cases the Android run time will kill an Activity during its visible lifetime without a call to onStop.
- The onStop method should be used to pause or stop animations, threads, Sensor listeners, GPS lookups, Timers, Services, or other processes that are used exclusively to update the UI.
- The onRestart method is called immediately prior to all but the first call to onStart.
- Use it to implement special processing that you want done only when the Activity restarts within its full lifetime.

Understanding Activity Lifetimes

- **The Active Lifetime**
- The active lifetime starts with a call to onResume and ends with a corresponding call to onPause.
- An active Activity is in the foreground and is receiving user input events.
- Your Activity is likely to go through many active lifetimes before it's destroyed, as the active lifetime will end when a new Activity is displayed, the device goes to sleep, or the Activity loses focus.
- Immediately before onPause, a call is made to onSaveInstanceState.
- Most Activity implementations will override at least the onSaveInstanceState method to commit unsaved changes, as it marks the point beyond which an Activity may be killed without warning.

Android Activity Classes

- **MapActivity** — Encapsulates the resource handling required to support a MapView widget within an Activity.
- **ListActivity** — Wrapper class for Activities that feature a ListView bound to a data source as the primary UI metaphor, and expose event handlers for list item selection.
- **ExpandableListActivity** — Similar to the ListActivity but supports an ExpandableListView.


THANKS