UNIT 5 ANDROID APPLICATION FUNDAMENTALS

5.0 INTRODUCTION

In this unit we describe of the basic app components, additional components, resources and the manifest file using Android Studio, which is the Open Source platform provided for application developers. This unit also provides you an overview of the interaction and the association between the components and the application.

5.1 OBJECTIVES

Outcomes

- Explain the activity Components
- Outline the manifest file
- Explain the introduction to AVD
- Create an Android Virtual Device to simulate a device and to display on the development computer

Terminology

Manifest Filbe:	file containing metadata for a group files that are part of a coherent unit
Bound Services:	Server in a client-server interface
RPC:	Remote Procedure Call
API:	Application Programming Interface

After studying this Unit, you should be able to:

5.2 BASIC APP COMPONENTS

As you already learnt in Unit 2, there are four types of app components and they are:

- Activities
- Services
- Content Providers
- Broadcast Receivers

Activities

An activity is the entry point for interacting with the user. As explained in unit 2,it represents a single screen with a user interface.An activity facilitates the following key interactions between system and app:

- Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
- Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
- Helping the app handle having its process is killed so the user can return to activities with their previous state restored.
- Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI withsetContentView(View). While activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows (via a theme with windowIsFloating set) or embedded inside of another activity (using ActivityGroup).

There are two methods almost all subclasses of Activity will implement:

- onCreate(Bundle) is where you initialize your activity. Most importantly, here you will usuallycall setContentView(int) with a layout resource defining your UI, and using findViewById(int) to retrieve the widgets in that UI that ryou need to interact with programmatically.
- onPause() is where you deal with the user leaving your activity. Most importantly, any changes made by the user should at this point be committed (usually to the ContentProvider holding the data).

To be of use with Context.startActivity(),all activity classes must have a corresponding <activity> declaration in their package's AndroidManifest.xmlas shown in Program5.1. An activity must be implemented as a subclass of the Activity class.

```
publicclassMainActivityextendsActivity{
/** Called when the activity is first created. */
@Override
publicvoidonCreate(Bundle savedInstanceState){
super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
```

```
}
/** Called when the activity is about to be visible.*/
@Override
protectedvoidonStart(){
super.onStart();
}
/** Called when the activity has become visible. */
@Override
protectedvoidonResume(){
super.onResume();
}
/** Called when another activity is taking focus. */
@Override
protectedvoidonPause(){
super.onPause();}
/** Called when the activity is no longer visible. */
@Override
protectedvoidonStop(){
super.onStop();
}
/** Called just before the activity is destroyed. */
@Override
publicvoidonDestroy(){
super.onDestroy();}
}
```

Program 5.1 Methods in Activity Class

Source(http://www.androdevelopment.com/android-activities/)

Services

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. As stated in unit 2, it is a component that runs in the background to perform longrunning operations or to perform work for remote processes. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. There are two very distinct semantics services that tell the system about how to manage an app: Started services tell the system to keep them running until their work is completed. This could be to sync some data in the background or play music even after the user leaves the app. Syncing data in the background or playing music also represents two different types of started services that modify how the system handles them:

- Music playback is something the user is directly aware of, so the app tells the system this by saying it wants to be foreground with a notification to tell the user about it; in this case the system knows that it should try really hard to keep that service's process running, because the user will be unhappy if it goes away.
- A regular background service is not something the user is directly aware as running, so the system has more freedom in managing its process. It may allow it to be killed (and then restarting the service sometime later) if it needs RAM for things that are of more immediate concern to the user.

Bound services run because some other app (or the system) has said that it wants to make use of the service. This is basically the service providing an API to another process. The system thus knows there is a dependency between these processes, so if process A is bound to a service in process B, it knows that it needs to keep process B

(and its service) running for A. Further, if process A is something the user cares about, then it also knows to treat process B as something the user also cares about. Because of their flexibility (for better or worse), services have turned out to be a really useful building block for all kinds of higher-level system concepts. Live wallpapers, notification listeners, screen savers, input methods, accessibility services, and many other core system features are all built as services that applications implement and the system binds to when they should be running.

To create a service, you must create a subclass of Service or use one of its existing subclasses. In your implementation, you must override some callback methods that handle key aspects of the service lifecycle and provide a mechanism that allows the components to bind to the service, if appropriate. You need to have prior knowledge of applying object oriented concepts to do this. These are the most important callback methods that you should override:

onStartCommand()

The system invokes this method by calling startService() when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling stopSelf() or stopService(). If you only want to provide binding, you don't need to implement this method.

onBind()

The system invokes this method by calling bindService() when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an IBinder. You must always implement this method; however, if you don't want to allow binding, you should return null.

onCreate()

The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either onStartCommand() or onBind()). If the service is already running, this method is not called.

onDestroy()

The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

If a component starts the service by calling startService() (which results in a call to onStartCommand()), the service continues to run until it stops itself with stopSelf() or another component stops it by calling stopService().

If a component calls bindService() to create the service and onStartCommand() is not called, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys it.

Traditionally, there are two classes you can extend to create a started service named Service and IntentService.

The Service is the base class for all services (shown in Program 5.2). When you extend this class, it is important to create a new thread in which the service can complete all of its work; the service uses your application's main thread by default, which can slow the performance of any activity that your application is running.

```
publicclassHelloServiceextendsService{
  privateLooper mServiceLooper;
  privateServiceHandler mServiceHandler;

  // Handler that receives messages from the thread
  privatefinalclassServiceHandlerextendsHandler{
      publicServiceHandler(Looper looper){
          super(looper);
      }
      @Override
      publicvoid handleMessage(Message msg){

 // Normally we would do some work here, likedownload a //file.For our
sample, we just sleep for 5 seconds.
          try{
              Thread.sleep(5000);
          }catch(InterruptedException e){
              // Restore interrupt status.
              Thread.currentThread().interrupt();
          }
// Stop the service using the startId, so that we don't stop//the
service in the middle of handling another job
          stopSelf(msg.arg1);
      }
  }
  @Override
  publicvoid onCreate(){
// Start up the thread running the service.  Note that we //create
aseparate thread because the service normally runs //in the
process'smain thread, which we don't want to //block.  We also make
itbackground priority so CPU-//intensive work will not disrupt our UI.

    HandlerThread thread
=newHandlerThread("ServiceStartArguments",
            Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

// Get the HandlerThread's Looper and use it for our //Handler
    mServiceLooper = thread.getLooper();
    mServiceHandler =newServiceHandler(mServiceLooper);
  }
  @Override
  publicint onStartCommand(Intent intent,int flags,int
startId){
```

```
      Toast.makeText(this,"service
starting",Toast.LENGTH_SHORT).show();

// For each start request, send a message to start a // job and
deliver thestart ID so we know which     // request we're
stopping when we finish the job
      Message msg = mServiceHandler.obtainMessage();
      msg.arg1 = startId;
      mServiceHandler.sendMessage(msg);

// If we get killed, after returning from here, restart
      return START_STICKY;
  }
  @Override
  publicIBinder onBind(Intent intent){
      // We don't provide binding, so return null
      returnnull;
  }
  @Override
  publicvoid onDestroy(){
    Toast.makeText(this,"service
done",Toast.LENGTH_SHORT).show();
  }
}
```

Program 5.2 Service class declaration

Source(https://developer.android.com/guide/components/services.html)

The IntentService is a subclass of Service that uses a worker thread to handle all of the start requests, one at a time (Shown in Program 5.3). This is the best option if you don't require that your service handle multiple requests simultaneously. Implement onHandleIntent(), which receives the intent for each start request so that you can complete the background work.

```
publicclassHelloIntentServiceextendsIntentService{
/** A constructor is required, and must call the super
 * IntentService(String)constructor with a name for the worker
thread.*/
  publicHelloIntentService(){
      super("HelloIntentService");
  }
/* The IntentService calls this method from the default worker
thread with the intent that started the service. When this
method returns, IntentService stops the service, as
appropriate. */
  @Override
  protectedvoid onHandleIntent(Intent intent){
 // Normally we would do some work here, likedownload a //file.For our
sample, we just sleep for 5 seconds.
      try{
```

```
            Thread.sleep(5000);
        }catch(InterruptedException e){
            // Restore interrupt status.
            Thread.currentThread().interrupt();
        }
    }
}
```

Program 5.3 Intent Service subclass declaration
Source (https://developer.android.com/guide/components/services.html)

Broadcast receivers

As already stated in unit 2, broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. Many broadcasts originate from the system for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured.

A broadcast receiver is implemented as a subclass (shown in program 5.4) of BroadcastReceiver and each broadcast is delivered as an Intent object.

```
publicclassMyReceiverextendsBroadcastReceiver{
@Override
publicvoid onReceive(Context context,Intent intent){
// Implement action for received broadcast.
}
}
```

Program 5.4 Broadcast receiver subclass

Source(https://developer.android.com/samples/AppShortcuts/src/com.example.android.appshortcuts/MyReceiver.html?hl=pt-br)

Content providers

As already stated in unit 2, content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. There are a few particular things this allows the system to do in managing an app:

- Assigning a Uniform Resource Identifier (URI) does not require that the app remain running, so URIs can persist after their owning apps have exited. The system only needs to make sure that an owning app is still running when it has to retrieve the app's data from the corresponding URI.

- These URIs also provide an important fine-grained security model. For example, an app can place the URI for an image it has on the clipboard, but leave its content provider locked up so that other apps cannot freely access it. When a second app attempts to access that URI on the clipboard, the system can allow that app to access the data via a temporary URI permission grant so that it is allowed to access the data only behind that URI, but nothing else in the second app.
- A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that and your app can use it instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can simply start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To the user, it seems as if the camera is actually a part of your app.
- When the system starts a component, it starts the process for that app if it's not already running and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that belongs to the camera app, not in your app's
- process. Therefore, unlike apps on most other systems, Android apps don't have a single entry point (there's no main() function).
- Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. However, the Android system can. To activate a component in another app, deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

5.2.1 Video -V6: **Android Development**

URL: **https://tinyurl.com/y9xuh62v**

In this video, we will be introducing the components of the Android Application Fundamentals. You may watch the video while reading this unit in order to understand the content better. After watching the video answer the question in Activity 5.1.

5.2.2 Check Your Progress

Identify the element that is not part of the basic application component of an Android application.

- ○ Activities
- ○ Services
- ○ Content Providers
- ○ Screencast Receivers
- ○ Broadcast Receivers

5.3 ADDITIONAL COMPONENTS

Other than basic four app components there are few additional components as well. In this section we will discuss these additional components.

Application Class

The Application class in Android is Base class for maintaining global application state which contains all other components such as activities and services. The Application class, or any subclass of the Application class, is instantiated before any other class when the process for your application/package is created.

Defining Your Application Class

If we do want a custom application class, we start by creating a new class which extends Android.app.Application as the following Program 5.5:

```
import android.app.Application;

publicclassMyCustomApplicationextends Application {
// Called when the application is starting, before any // other
application objects have been created.
// Overriding this method is totally optional!
        @Override
        publicvoid onCreate(){
        super.onCreate();
// Required initialization logic here!
        }
// Called by the system when the device configuration changes
//while your component is running.
// Overriding this method is totally optional!
        @Override
        publicvoid onConfigurationChanged(Configuration
newConfig){
        super.onConfigurationChanged(newConfig);
        }
// This is called when the overall system is running //low on
memory, and would like actively running //processes to tighten
their belts.
// Overriding this method is totally optional!
         @Override
        publicvoidonLowMemory(){
        super.onLowMemory();
                }
}
```

Program 5.5 Application class

Source(https://guides.codepath.com/android/Understanding-the-Android-Application-Class)

And specify the android:name property in the <application> node in AndroidManifest.xmlas in code snippet given below.

```
<application
   android:name=".MyCustomApplication"
   android:icon="@drawable/icon"
   android:label="@string/app_name"
   ...>
```

Source(https://guides.codepath.com/android/Understanding-the-Android-Application-Class)

That is all what you need to get started with your custom application.

Fragment

A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity. Interaction with fragments is done through FragmentManager, which can be obtained
via Activity.getFragmentManager() andFragment.getFragmentManager().

The Fragment class can be used many ways to achieve a wide variety of results. In its core, it represents a particular operation or interface that is running within a larger Activity. A Fragment is closely tied to the Activity it is in, and cannot be used apart from one. Though Fragment defines its own lifecycle, that lifecycle is dependent on its activity: if the activity is stopped, no fragments inside of it can be started; when the activity is destroyed, all fragments will be destroyed.

All subclasses of Fragment must include a public no-argument constructor. The framework will often re-instantiate a fragment class when needed, in particular during state restore, and needs to be able to find this constructor to instantiate it. If the no-argument constructor is not available, a runtime exception will occur in some cases during state restore.

Fragment Lifecycle

Though a Fragment's lifecycle is tied to its owning activity, it has its own wrinkle on the standard activity lifecycle. It includes basic activity lifecycle methods such as onResume(), but also important are methods related to interactions with the activity and UI generation.

The core series of lifecycle methods that are called to bring a fragment up to resumed state (interacting with the user) are:

1. **onAttach(Activity)** called once the fragment is associated with its activity.
2. **onCreate(Bundle)** called to do initial creation of the fragment.
3. **onCreateView(LayoutInflater, ViewGroup, Bundle)** creates and returns the view hierarchy associated with the fragment.
4. **onActivityCreated(Bundle)** tells the fragment that its activity has completed itsown Activity.onCreate().
5. **onViewStateRestored(Bundle)** tells the fragment that all of the saved state of its view hierarchy has been restored.

6. **onStart()** makes the fragment visible to the user (based on its containing activity being started).

7. **onResume()** makes the fragment begin interacting with the user (based on its containing activity being resumed).

As a fragment is no longer being used, it goes through a reverse series of callbacks:

1. **onPause()** fragment is no longer interacting with the user either because its activity is being paused or a fragment operation is modifying it in the activity.

2. **onStop()** fragment is no longer visible to the user either because its activity is being stopped or a fragment operation is modifying it in the activity.

3. **onDestroyView()** allows the fragment to clean up resources associated with its View.

4. **onDestroy()** called to do final cleanup of the fragment's state.

5. **onDetach()** called immediately prior to the fragment no longer being associated with its activity.

Fragment Layout

Fragments can be used as part of your application's layout, allowing you to better modularize your code and more easily adjust your user interface to the screen it is running on. As an example, we can look at a simple program consisting of a list of items, and display of the details of each item.

An activity's layout XML can include <fragment> tags to embed fragment instances inside of the layout. For example, here is a simple layout that embeds one fragment shown in code snippet below:

```
<FrameLayoutxmlns:android="http://schemas.android.com/apk/res/a
ndroid"
    android:layout_width="match_parent"android:layout_height="m
atch_parent">
    <fragmentclass="com.example.android.apis.app.FragmentLayout
$TitlesFragment"
            android:id="@+id/titles"
            android:layout_width="match_parent"android:layout_h
eight="match_parent"/>
</FrameLayout>
```

Source :(https://developer.android.com/reference/android/app/Fragment.html)

The layout is installed in the activity in the normal way as shown in code snippet below:

```
@Override
protectedvoid onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.fragment_layout);
}
```

Source: (https://developer.android.com/reference/android/app/Fragment.html

View

This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for *widgets*, which are used to create interactive UI components (buttons, text fields, etc.). The ViewGroup subclass is the base class for *layouts*, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties.

Using Views

All of the views in a window are arranged in a single tree. You can add views either from code or by specifying a tree of views in one or more XML layout files. There are many specialized subclasses of views that act as controls or are capable of displaying text, images, or other content.

Set properties: for example, setting the text of a TextView. The available properties and the methods that set them will vary among the different subclasses of views. Note that properties that are known at build time can be set in the XML layout files.

Set focus: The framework will handle moving focus in response to user input. To force focus to a specific view, call requestFocus().

Set up listeners: Views allow clients to set listeners that will be notified when something interesting happens to the view. For example, all views will let you set a listener to be notified when the view gains or loses focus. You can register such a listener using setOnFocusChangeListener(android.view.View.OnFocusChangeListener). Other view subclasses offer more specialized listeners. For example, a Button exposes a listener to notify clients when the button is clicked.

Set visibility: You can hide or show views using setVisibility(int).

Note: The Android framework is responsible for measuring, laying out and drawing views. You should not call methods that perform these actions on views yourself unless you are actually implementing a *ViewGroup*.

Intents and Intent Filters

An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use-cases:

- **To start an activity:** An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to startActivity(). The Intent describes the activity to start and carries any necessary data. If you want to receive a result from the activity when it finishes, call startActivityForResult(). Your activity receives the result as a separate Intent object in your activity's onActivityResult() callback. For more information, see the Activities guide.

- **To start a service:** A Service is a component that performs operations in the background without a user interface. You can start a service to perform a one-time operation (such as download a file) by passing an Intent tostartService(). The Intent describes the service to start and carries any necessary data. If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to bindService(). For more information, see the Services guide

- **To deliver a broadcast:**A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging.You can deliver a broadcast to other apps by passing an Intent to sendBroadcast(), sendOrderedBroadcast(), or sendStickyBroadcast().

Intents Types

There are two types of intents:

- **Explicit intents** specify the component to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.
- **Implicit intents** do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

When you create an explicit intent to start an activity or service, the system immediately starts the app component specified in the Intent object.

When you create an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the *intent filters* declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

5.4 RESOURCES

This sits on top of the asset manager of the application (accessible through getAssets()) and provides a high-level API for getting typed data from the assets.

The Android resource system keeps track of all non-code assets associated with an application. You can use this class to access your application's resources. You can generally acquire the Resources instance associated with your application with getResources().

The Android SDK tools compile your application's resources into the application binary at build time. To use a resource, you must install it correctly in the source tree

(inside your project's `res/` directory) and build your application. As part of the build process, the SDK tools generate symbols for each resource, which you can use in your application code to access the resources.

Using application resources makes it easy to update various characteristics of your application without modifying code, and—by providing sets of alternative resources—enables you to optimize your application for a variety of device configurations (such as for different languages and screen sizes). This is an important aspect of developing Android applications that are compatible on different types of devices.

You should always externalize resources such as images and strings from your application code, so that you can maintain them independently. Externalizing your resources also allows you to provide alternative resources that support specific device configurations such as different languages or screen sizes, which becomes increasingly important as more Android-powered devices become available with different configurations. In order to provide compatibility with different configurations, you must organize resources in your project's `res/` directory, using various sub-directories that group resources by type and configuration.

For any type of resource, you can specify *default* and multiple *alternative* resources for your application:

- Default resources are those that should be used regardless of the device configuration or when there are no alternative resources that match the current configuration.
- Alternative resources are those that you've designed for use with a specific configuration. To specify that a group of resources are for a specific configuration, append an appropriate configuration qualifier to the directory name.

For example, while your default UI layout is saved in the res/layout/ directory, you might specify a different layout to be used when the screen is in landscape orientation, by saving it in the res/layout-land/ directory. Android automatically applies the appropriate resources by matching the device's current configuration to your resource directory names.

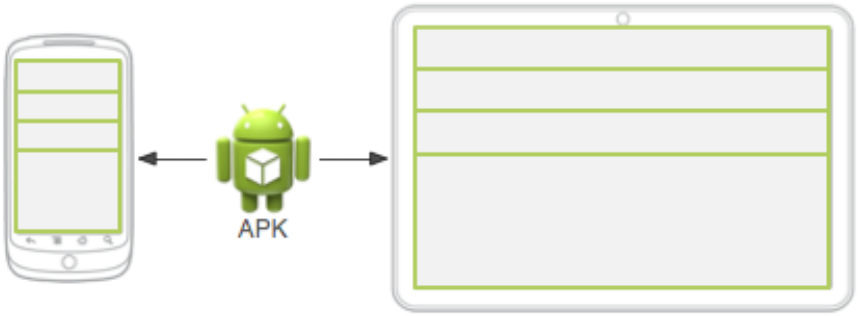

Figure 5.1 Two different devices, each using the default layout (the app provides no alternative layouts).

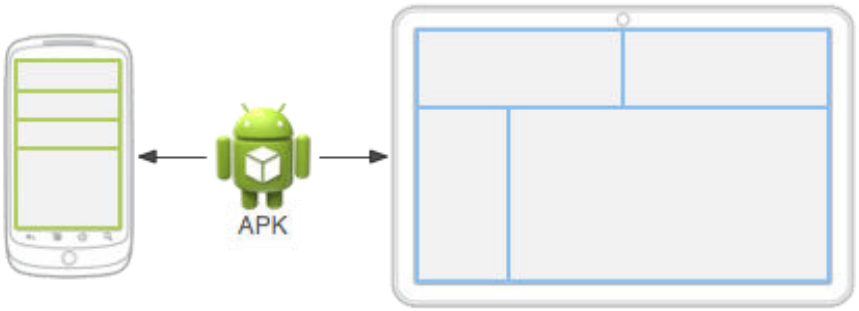

Figure 5.2 Two different devices, each using a different layout provided for different screen sizes.

Figure 5.1: illustrates how the system applies the same layout for two different devices when there are no alternative resources available. Figure 2 shows the same application when it adds an alternative layout resource for larger screens.

The following section provide a complete guide to how you can organize your application resources, specify alternative resources, access them in your application, and more:

Providing Resources

What kinds of resources you can provide in your app, where to save them, and how to create alternative resources for specific device configurations.

Accessing Resources

How to use the resources you've provided, either by referencing them from your application code or from other XML resources.

Handling Runtime Changes

How to manage configuration changes that occur while your Activity is running.

Localization

A bottom-up guide to localizing your application using alternative resources. While this is just one specific use of alternative resources, it is very important in order to reach more users.

Complex XML Resources

An XML format for building complex resources like animated vector drawables in a single XML file.

Resource Types

A reference of various resource types you can provide, describing their XML elements, attributes, and syntax. For example, this reference shows you how to create a resource for application menus, drawables, animations, and more.

5.4.1 Check Your Progress

Explain how you can organize your application resources.

5.5 ANDROID MANIFEST

Every application must have an `AndroidManifest.xml` file (with precisely that name) in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code.

Among other things, the manifest file does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application, which include the activities, services, broadcast receivers, and content providers that compose the application. It also names the classes that implement each of the components and publishes their capabilities, such as the Intent messages that they can handle. These declarations inform the Android system of the components and the conditions in which they can be launched.
- It determines the processes that host the application components.
- It declares the permissions that the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application runs. These declarations are present in the manifest only while the application is being developed and are removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

Manifest file structure

The code snippet below shows the general structure of the manifest file and every element that it can contain. Each element, along with all of its attributes, is fully documented in a separate file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission/>
    <permission/>
    <permission-tree/>
    <permission-group/>
    <instrumentation/>
    <uses-sdk/>
    <uses-configuration/>
    <uses-feature/>
    <supports-screens/>
    <compatible-screens/>
    <supports-gl-texture/>
```

```
    <application>

        <activity>
            <intent-filter>
                <action/>
                <category/>
                <data/>
            </intent-filter>
            <meta-data/>
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </activity-alias>

<service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </receiver>

        <provider>
            <grant-uri-permission/>
            <meta-data/>
            <path-permission/>
        </provider>
        <uses-library/>
    </application>
</manifest>
```

Source: (https://developer.android.com/guide/topics/manifest/manifest-intro.html)

5.6 File conventions

This section describes the conventions and rules that apply generally to all of the elements and attributes in the manifest file.

Elements

Only the <manifest> and <application> elements are required. They each must be present and can occur only once. Most of the other elements can occur many times or not at all. However, at least some of them must be present before the manifest file becomes useful.

If an element contains anything at all, it contains other elements. All of the values are set through attributes, not as character data within an element.

Elements at the same level are generally not ordered. For example, the <activity>, <provider>, and <service> elements can be intermixed in any sequence. There are two key exceptions to this rule:

- An <activity-alias> element must follow the <activity> for which it is an alias.
- The <application> element must be the last element inside the <manifest> element. In other words, the `</application>` closing tag must appear immediately before the `</manifest>` closing tag.

Attributes

In a formal sense, all attributes are optional. However, there are some attributes that must be specified so that an element can accomplish its purpose. Use the documentation as a guide. For truly optional attributes, it mentions a default value or states what happens in the absence of a specification.

Except for some attributes of the root <manifest> element, all attribute names begin with an `android:`prefix. For example, `android:alwaysRetainTaskState`. Because the prefix is universal, the documentation generally omits it when referring to attributes by name.

Declaring class names

Many elements correspond to Java objects, including elements for the application itself (the <application> element) and its principal components: activities (<activity>), services (<service>), broadcast receivers (<receiver>), and content providers (<provider>).

If you define a subclass, as you almost always would for the component classes (Activity, Service, BroadcastReceiver, and ContentProvider), the subclass is declared through a `name` attribute. The name must include the full package designation. For example, a Service subclass might be declared as follows:

```
<manifest . . . >
    <application . . . >
        <serviceandroid:name="com.example.project.SecretService
" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

Source(https://developer.android.com/guide/topics/manifest/manifest-intro.html)

However, if the first character of the string is a period, the application's package name (as specified by the <manifest> element's package attribute) is appended to the string. The following assignment is the same as that shown above:

```
<manifestpackage="com.example.project" . . . >
    <application . . . >
        <serviceandroid:name=".SecretService" . . . >
            . . .
```

```
        </service>
        . . .
    </application>
</manifest>
```

Source*(https://developer.android.com/guide/topics/manifest/manifest-intro.html)*

When starting a component, the Android system creates an instance of the named subclass. If a subclass isn't specified, it creates an instance of the base class.

Multiple values

If more than one value can be specified, the element is almost always repeated, rather than multiple values being listed within a single element. For example, an intent filter can list several actions:

```
<intent-filter . . . >
    <actionandroid:name="android.intent.action.EDIT"/>
    <actionandroid:name="android.intent.action.INSERT"/>
    <actionandroid:name="android.intent.action.DELETE"/>
    . . .
</intent-filter>
```

Source**:***(https://developer.android.com/reference/android/content/Intent.html)*

Resource values

Some attributes have values that can be displayed to users, such as a label and an icon for an activity. The values of these attributes should be localized and set from a resource or theme. Resource values are expressed in the following format:

@[package:]type/name

You can ommit the *package* name if the resource is in the same package as the application. The *type* is a type of resource, such as *string* or *drawable*, and the *name* is the name that identifies the specific resource. Here is an example:

```
<activityandroid:icon="@drawable/smallPic" . . . >
```

Source (https://developer.android.com/guide/topics/manifest/manifest-intro.html)

The values from a theme are expressed similarly, but with an initial ? instead of @:

?[package:]type/name

String values

Where an attribute value is a string, you must use double backslashes (\\) to escape characters, such as \\n for a newline or \\uxxxx for a Unicode character.

5.6.1 Check Your Progress

Explain the role of AndroidManifest.xml file in an Android application.

5.7 File conventions

First part of this unit gave you an in-depth knowledge about the four main types of mobile application components. Thisunitalso introduced you to the methods that was facilitated by Android and the elements used in any Android application development such as fragments and intents. You will need to refer to this unit when you are writing your application to understand the process beneath each method while executing.

5.8 FURTHER READINGS

- https://developer.android.com/guide/components/fundamentals
- https://developer.android.com/guide/topics/manifest/manifest-intro