Arba Minch University

Institute of Technology

Faculty of Computing and Software Engineering

Mobile Application Development Course Notes
For Exit Exam Tutorial

Prepared and Compiled by

Mr. Bikila A.

May, 2023

TABLE OF CONTENTS

LEARNING OBJECTIVES

1) Describe the basic **components** of an Android application.
2) Define the **lifecycle methods** of Android application components.
3) Describe the basics of **event handling** in Android.
4) Describe the basics of **graphics** and **multimedia support** in Android.
5) Demonstrate basic skills of using an integrated development environment (Android Studio) and Android Software Development Kit (SDK) for implementing Android applications.
6) Demonstrate through a **simple application** the understanding of the basic concepts of Android

INTRODUCTION

The Android Platform

Android is an open source and Linux-based Operating System for mobile devices such as smartphones and tablet computers. Android was developed by the *Open Handset Alliance (OHA)*, led by Google, and other companies. Android includes a Linux kernel-based OS, a rich UI, end-user applications, code libraries, application frameworks, multimedia support, and much more. And, yes, even telephone functionality is included! Whereas components of the underlying OS are written in C or C++, user applications are built for Android in Java.

A Linux kernel provides a foundational hardware abstraction layer, as well as core services such as process, memory, and file-system management.

- ✓ The kernel is where hardware-specific drivers are implemented—capabilities such as Wi-Fi and Bluetooth are here.

Prominent code libraries, including the following:

- Browser technology from WebKit, the same open source engine powering Mac's Safari and the iPhone's Mobile Safari browser. WebKit has become the de facto standard for most mobile platforms.
- Database support via SQLite, an easy-to-use SQL database.
- Advanced graphics support, including 2D, 3D, animation from Scalable Games Language (SGL), and OpenGL ES.
- Audio and video media support from PacketVideo's OpenCORE, and Google's own Stagefright media framework.
- Secure Sockets Layer (SSL) capabilities from the Apache project.

An array of managers that provide services for

- Activities and views
- Windows
- Location-based services
- Telephony
- Resources

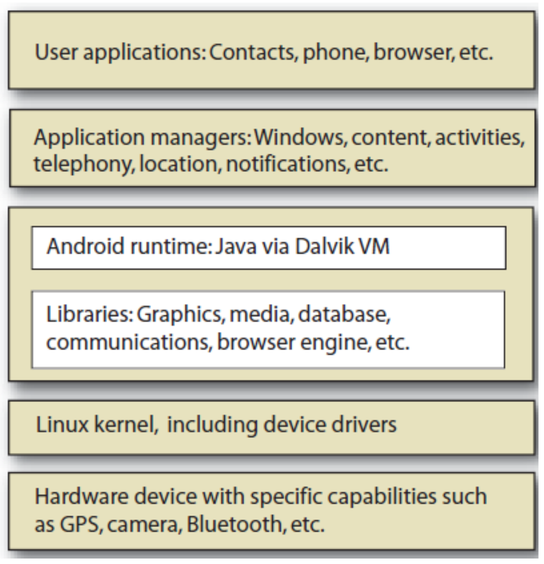

User applications: Contacts, phone, browser, etc.
Application managers: Windows, content, activities, telephony, location, notifications, etc.
Android runtime: Java via Dalvik VM
Libraries: Graphics, media, database, communications, browser engine, etc.
Linux kernel, including device drivers
Hardware device with specific capabilities such as GPS, camera, Bluetooth, etc.

Fig 1-The Android stack

The Android runtime, which provides

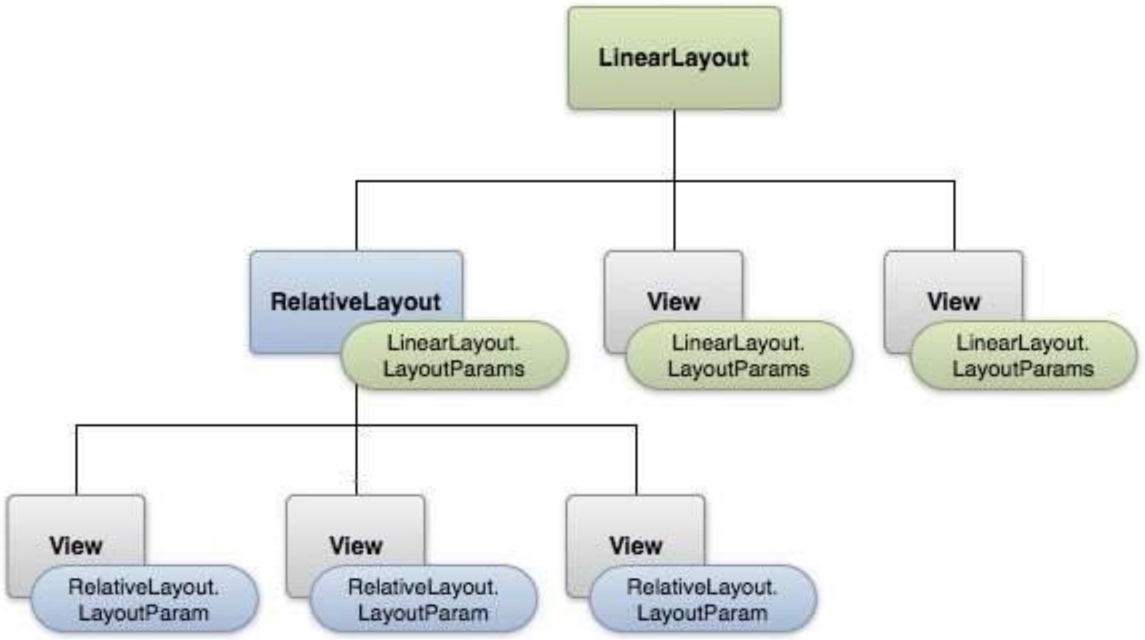
- Core Java packages for a nearly full-featured Java programming environment.
- The Dalvik Virtual Machine, which employs services of the Linux-based kernel to provide an environment to host Android applications.

ANDROID UI LAYOUTS

The basic building block for user interface is a **View** object which is created from the View class and 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 like buttons, text fields, etc.

The **ViewGroup** is a subclass of **View** and provides invisible container that hold other Views or other ViewGroups and define their layout properties.

At third level we have different layouts which are subclasses of ViewGroup class and a typical layout defines the visual structure for an Android user interface and can be created either at run time using **View/ViewGroup** objects or you can declare your layout using simple XML file **main_layout.xml** which is located in the res/layout folder of your project.


LinearLayout
RelativeLayout
LinearLayout.LayoutParams
View
LinearLayout.LayoutParams
View
LinearLayout.LayoutParams
View
RelativeLayout.LayoutParam
View
RelativeLayout.LayoutParam
View
RelativeLayout.LayoutParam

Layout params

This tutorial is more about creating your GUI based on layouts defined in XML file. A layout may contain any type of widgets such as buttons, labels, textboxes, and so on. Following is a simple example of XML file having LinearLayout −

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <TextView android:id="@+id/text"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="This is a TextView" />

   <Button android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="This is a Button" />

   <!-- More GUI components go here  -->

</LinearLayout>
```

Once your layout has created, you can load the layout resource from your application code, in your *Activity.onCreate()* callback implementation as shown below −

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

Android Layout Types

There are number of Layouts provided by Android which you will use in almost all the Android applications to provide different view, look and feel.

S.No	Layout & Description
1	Linear Layout LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
2	Relative Layout RelativeLayout is a view group that displays child views in relative positions.
3	Table Layout TableLayout is a view that groups views into rows and columns.
4	Absolute Layout AbsoluteLayout enables you to specify the exact location of its children.
5	Frame Layout The FrameLayout is a placeholder on screen that you can use to display a single view.
6	List View ListView is a view group that displays a list of scrollable items.
7	Grid View GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and there are other attributes which are specific to that layout. Following are common attributes and will be applied to all the layouts:

S.No	Attribute & Description
1	**android:id** This is the ID which uniquely identifies the view.
2	**android:layout_width**

	This is the width of the layout.
3	**android:layout_height** This is the height of the layout
4	**android:layout_marginTop** This is the extra space on the top side of the layout.
5	**android:layout_marginBottom** This is the extra space on the bottom side of the layout.
6	**android:layout_marginLeft** This is the extra space on the left side of the layout.
7	**android:layout_marginRight** This is the extra space on the right side of the layout.
8	**android:layout_gravity** This specifies how child Views are positioned.
9	**android:layout_weight** This specifies how much of the extra space in the layout should be allocated to the View.
10	**android:layout_x** This specifies the x-coordinate of the layout.
11	**android:layout_y** This specifies the y-coordinate of the layout.
12	**android:layout_width** This is the width of the layout.
13	**android:paddingLeft** This is the left padding filled for the layout.
14	**android:paddingRight** This is the right padding filled for the layout.
15	**android:paddingTop** This is the top padding filled for the layout.
16	**android:paddingBottom** This is the bottom padding filled for the layout.

Here, width and height are the dimension of the layout/view which can be specified in terms of dp (Density-independent Pixels), sp (Scale-independent Pixels), pt (Points which is 1/72 of an inch), px(Pixels), mm (Millimeters) and finally in (inches).

You can specify width and height with exact measurements but more often, you will use one of these constants to set the width or height −

- **android:layout_width=wrap_content** tells your view to size itself to the dimensions required by its content.
- **android:layout_width=fill_parent** tells your view to become as big as its parent view.

Gravity attribute plays important role in positioning the view object and it can take one or more (separated by '|') of the following constant values.

Constant	Value	Description
top	0x30	Push object to the top of its container, not changing its size.
bottom	0x50	Push object to the bottom of its container, not changing its size.
left	0x03	Push object to the left of its container, not changing its size.
right	0x05	Push object to the right of its container, not changing its size.
center_vertical	0x10	Place object in the vertical center of its container, not changing its size.
fill_vertical	0x70	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	0x01	Place object in the horizontal center of its container, not changing its size.
fill_horizontal	0x07	Grow the horizontal size of the object if needed so it completely fills its container.
center	0x11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
fill	0x77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
clip_vertical	0x80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
clip_horizontal	0x08	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.

start	0x00800003	Push object to the beginning of its container, not changing its size.
end	0x00800005	Push object to the end of its container, not changing its size.

View Identification

A view object may have a unique ID assigned to it which will identify the View uniquely within the tree. The syntax for an ID, inside an XML tag is −

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

Following is a brief description of @ and + signs −

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources. To create an instance of the view object and capture it from the layout, use the following −

```
Button myButton = (Button) findViewById(R.id.my_button);
```

ANDROID RESOURCES

There are many more items which you use to build a good Android application. Apart from coding for the application, you take care of various other **resources** like static content that your code uses, such as bitmaps, colors, layout definitions, user interface strings, animation instructions, and more. These resources are always maintained separately in various sub-directories under **res/** directory of the project.

This tutorial will explain you how you can organize your application resources, specify alternative resources and access them in your applications.

Organize resource in Android Studio

```
MyProject/
  app/
    manifest/
      AndroidManifest.xml
  java/
    MyActivity.java
    res/
      drawable/
        icon.png
      layout/
        activity_main.xml
        info.xml
      values/
        strings.xml
```

Sr.No.	Directory & Resource Type
1	**anim/** XML files that define property animations. They are saved in res/anim/ folder and accessed from the **R.anim** class.
2	**color/** XML files that define a state list of colors. They are saved in res/color/ and accessed from the **R.color** class.
3	**drawable/** Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the **R.drawable** class.
4	**layout/** XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class.
5	**menu/** XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the **R.menu** class.
6	**raw/** Arbitrary files to save in their raw form. You need to call *Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files.
7	**values/** XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources you can create in this directory − • arrays.xml for resource arrays, and accessed from the **R.array** class. • integers.xml for resource integers, and accessed from the **R.integer** class. • bools.xml for resource boolean, and accessed from the **R.bool** class. • colors.xml for color values, and accessed from the **R.color** class. • dimens.xml for dimension values, and accessed from the **R.dimen** class. • strings.xml for string values, and accessed from the **R.string** class. • styles.xml for styles, and accessed from the **R.style** class.
8	**xml/** Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. You can save various configuration files here which will be used at run time.

Alternative Resources

Your application should provide alternative resources to support specific device configurations. For example, you should include alternative drawable resources (i.e.images) for different screen resolution and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources for your application.

To specify configuration-specific alternatives for a set of resources, follow the following steps −

- Create a new directory in res/ named in the form **<resources_name>-<config_qualifier>**. Here **resources_name** will be any of the resources mentioned in the above table, like layout, drawable etc. The **qualifier** will specify an individual configuration for which these resources are to be used. You can check official documentation for a complete list of qualifiers for different type of resources.
- Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files as shown in the below example, but these files will have content specific to the alternative. For example though image file name will be same but for high resolution screen, its resolution will be high.

Below is an example which specifies images for a default screen and alternative images for high resolution screen.

```
MyProject/
  app/
    manifest/
      AndroidManifest.xml
  java/
    MyActivity.java
    res/
      drawable/
        icon.png
        background.png
      drawable-hdpi/
        icon.png
        background.png
      layout/
        activity_main.xml
        info.xml
      values/
        strings.xml
```

Below is another example which specifies layout for a default language and alternative layout for Arabic language.

```
MyProject/
  app/
    manifest/
      AndroidManifest.xml
  java/
    MyActivity.java
```

```
res/
  drawable/
    icon.png
    background.png
  drawable-hdpi/
    icon.png
    background.png
  layout/
    activity_main.xml
    info.xml
  layout-ar/
    main.xml
  values/
    strings.xml
```

Accessing Resources

During your application development you will need to access defined resources either in your code, or in your layout XML files. Following section explains how to access your resources in both the scenarios −

Accessing Resources in Code

When your Android application is compiled, a **R** class gets generated, which contains resource IDs for all the resources available in your **res/** directory. You can use R class to access that resource using sub-directory and resource name or directly resource ID.

Example

To access *res/drawable/myimage.png* and set an ImageView you will use following code −

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

Here first line of the code make use of *R.id.myimageview* to get ImageView defined with id *myimageview* in a Layout file. Second line of code makes use of *R.drawable.myimage* to get an image with name **myimage** available in drawable sub-directory under **/res**.

Example

Consider next example where *res/values/strings.xml* has following definition −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string  name="hello">Hello, World!</string>
</resources>
```

Now you can set the text on a TextView object with ID msg using a resource ID as follows −

```
TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText(R.string.hello);
```

Example

Consider a layout *res/layout/activity_main.xml* with the following definition −

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical" >

   <TextView android:id="@+id/text"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello, I am a TextView" />

   <Button android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello, I am a Button" />

</LinearLayout>
```

This application code will load this layout for an Activity, in the onCreate() method as follows −

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

Accessing Resources in XML

Consider the following resource XML *res/values/strings.xml* file that includes a color resource and a string resource −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="opaque_red">#f00</color>
   <string name="hello">Hello!</string>
</resources>
```

Now you can use these resources in the following layout file to set the text color and text string as follows −

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:textColor="@color/opaque_red"
   android:text="@string/hello" />
```

Now if you will go through previous chapter once again where I have explained **Hello World!** example, and I'm sure you will have better understanding on all the concepts explained in this chapter. So I highly recommend to check previous chapter for working example and check how I have used various resources at very basic level.

ANDROID INTENTS

Intents and Intent-Filters bring the "**click it**" paradigm to the core of mobile application use (and development) for the Android platform:

- An Intent is a **declaration of need**. It's made up of a number of pieces of information that describe the desired action or service. Usually intent is made up of the **requested action** and, generically, the **data that accompanies the requested action**.
- An Intent-Filter is a **declaration of capability and interest in offering assistance to those in need**. It can be **generic** or **specific** with respect to which Intents it offers to service.

The ***action*** attribute of an Intent is typically a **verb**: for example, VIEW, PICK, or EDIT. A number of built-in Intent actions are defined as members of the Intent class. For example, to view a piece of information, an application employs the following Intent action:
android.content.Intent.ACTION_VIEW
The ***data*** component of an Intent is expressed in the form of a **URI** and can be virtually any piece of information, such as a contact record, a website location, or a reference to a media clip.

The Intent-Filter defines the *relationship* between the Intent and the application.

- Intent-Filters can be specific to the data portion of the Intent, the action portion, or both.
- Intent-Filters also contain a field known as a ***category***. The category **helps classify the action**.

For example, the category named CATEGORY_LAUNCHER instructs Android that the Activity containing this Intent-Filter should be visible in the main application launcher or home screen.

Intent-Filters are often defined in an application's **AndroidManifest.xml** file with the <intent-filter> tag.

There are three types of Intents, ***Implicit and Explicit intents, and Pending-Intents***.

- ✓ Implicit intents **rely on intent filters** and **the Android environment to dispatch the Intent to the appropriate recipient**
- ✓ Explicit intent **specifies the exact class** that will handle the Intent.

See example below for an explicit intent:

```
public void onClick(View v) {
try {
startActivityForResult(new Intent(v.getContext(),RefreshJobs.class),0);
} catch (Exception e) {
. . .
}
}
```

A **PendingIntent** is a type of Intent that allows an application to perform an operation on behalf of another application or at a later time. It is typically used when an application wants to perform an action in response to a user action or system event, but the application is not currently running or in the foreground. For example, a messaging app may create a PendingIntent to send a message at a later time, or a notification may create a PendingIntent to launch an activity when the user taps on it.

Pending-Intent vs Regular Intent

A Pending-Intent is a type of Intent in Android that differs from a regular Intent in the following ways:

- Context: A Pending-Intent is associated with a specific context, such as an activity, service, or broadcast receiver, whereas a regular Intent can be executed from any context.
- Permissions: A Pending-Intent preserves the permissions of the original caller, whereas a regular Intent may require additional permissions to execute.
- Execution time: A Pending-Intent is executed at a later time, such as when a user interacts with a notification or widget, whereas a regular Intent is executed immediately.
- Reusability: A Pending-Intent can be reused to perform the same operation multiple times, whereas a regular Intent is typically used for a single operation.
- Security: A Pending-Intent provides an additional layer of security by allowing an application to delegate an operation to another application or system component, without granting that component unnecessary permissions.

BASIC COMPONENTS OF AN ANDROID APPLICATION

In the world of mobile application development, an Android application is the most popular choice of developers. It offers a wide range of components that work together to deliver an excellent user experience.

The four basic components that make up an Android application are

a) Activities,
b) Services,
c) Broadcast Receivers, and
d) Content Providers.

Activities

Activities are responsible for managing the *user interface (UI)* and *handling user interactions*.

- ✓ The Activity employs one or more Views to present the actual UI elements to the user.
- ✓ The *Activity* class is extended by user classes.
- ✓ One of the primary tasks an Activity performs is displaying UI elements, which are implemented as Views and are typically defined in XML layout files.

Example of an Activity with one EditText and Button:

```
public class AWhereDoYouLive extends Activity {
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                final EditText addressfield =(EditText) findViewById(R.id.address);
                final Button button = (Button) findViewById(R.id.launchmap);
                button.setOnClickListener( new Button.OnClickListener() {
                        public void onClick(View view) {
                        try {
                        String address = addressfield.getText().toString();
                        address = address.replace(' ', '+');
                        Intent geoIntent = new Intent(android.content.Intent.ACTION_VIEW,
                        Uri.parse("geo:0,0?q=" + address));
                        startActivity(geoIntent);
                        } catch (Exception e) {
                        }
                        }
                });
        }
}
```

- ✓ Resources are precompiled into a special class known as the **R** class.

The UI for the above java code is created in the following xml file with a Linear layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Please enter your home address."/>

<EditText
android:id="@+id/address"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:autoText="true"/>

<Button
android:id="@+id/launchmap"
android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:text="Show Map"/>

</LinearLayout>
```

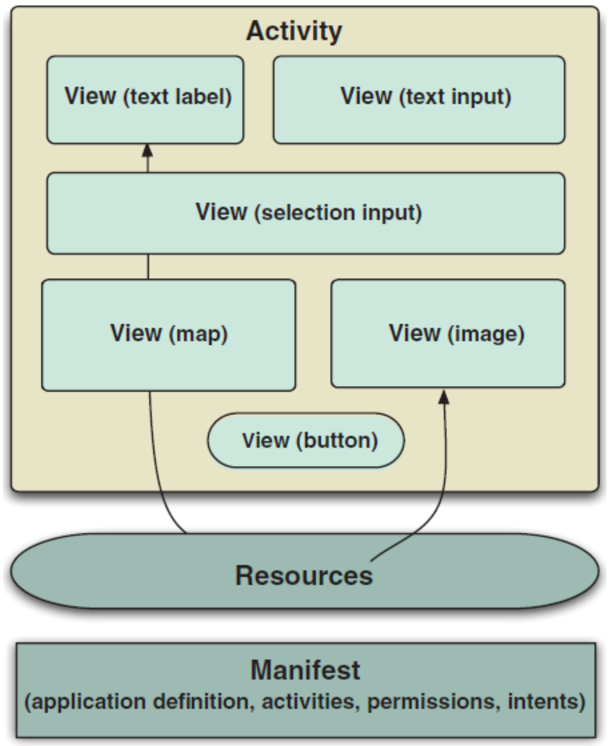


Figure – high level diagram of Activity , and its relation with views , resources and manifest

Creating a new activity

To create a new activity in android studio, follow the following steps:

- Open android studio and create a new project or open an existing one
- In the project panel on the left side of the screen, right click on the app folder and select new -> Activity -> Empty activity
- In the create new activity dialog box, enter a name for the activity and select the layout file for it
- Click finish to create the activity. Android studio will generate the necessary files for the activity, including the Java class file and the xml layout file
- Open the java class file for the activity and add the necessary code to handle to handle the activity's behavior. This may include, initialization of views, handling user input and interacting with other components of the app.
- Open the xml layout file for the activity and design the user interface for it. This may include adding views such as buttons, text fields, images and arranging them on the screen

The role of setContentView() method

In Android, the `setContentView()` method is used to set the user interface (UI) of an Activity. When an Activity is created, it does not have any UI components by default. The `setContentView()` method is used to inflate a layout file and add UI components to the Activity's view hierarchy.

The `setContentView()` method takes a layout resource ID as its parameter, which is the ID of the XML layout file that defines the UI components of the Activity. The method inflates the layout and adds the UI components to the Activity's view hierarchy.

Here's an example of how to use the `setContentView()` method in an Activity:

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

In this example, the `setContentView()` method is called in the `onCreate()` method of the Activity, passing the `R.layout.activity_main` layout resource ID as its parameter. The `activity_main.xml` layout file defines the UI components of the Activity and is located in the `res/layout` directory of the app's resources.

It's important to note that the `setContentView()` method should only be called once in the `onCreate()` method of an Activity. If you call it multiple times, the UI components from the previous call will be replaced with the components from the latest call.

In summary, the `setContentView()` method is used to inflate a layout file and set the UI components of an Activity. It's an important method that's called in the `onCreate()` method of an Activity to set up the initial UI.

Role of saveInstanceState() method

In Android, the `onSaveInstanceState()` method is a callback method that is called by the system when an Activity is about to be destroyed due to a configuration change, such as a screen rotation or the user changing the device's language. The purpose of this method is to save the current state of the Activity's UI and data so that it can be restored when the Activity is recreated.

When an Activity is destroyed due to a configuration change, the system calls the `onSaveInstanceState()` method and passes in a `Bundle` object as a parameter. The `Bundle` object is used to store key-value pairs that represent the state of the Activity's UI and data. The key-value pairs are typically used to store things like the text entered into an EditText field or the position of a SeekBar.

Here's an example of how to use the `onSaveInstanceState()` method in an Activity:

```
public class MyActivity extends AppCompatActivity {
    private int mCount = 0;

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

        if (savedInstanceState != null) {
            mCount = savedInstanceState.getInt("count");
        }
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("count", mCount);
    }

    public void incrementCount(View view) {
        mCount++;
        TextView countTextView = findViewById(R.id.count_text_view);
        countTextView.setText(String.valueOf(mCount));
    }
}
```

In this example, the Activity has a `TextView` that displays a count value, and a button that increments the count when clicked. The `onSaveInstanceState()` method is overridden to save the current count value in the `Bundle` object. The `onCreate()` method retrieves the count value from the `Bundle` object and sets it as the initial value of the count variable.

By saving the state of the Activity's UI and data in the `onSaveInstanceState()` method, the state can be restored when the Activity is recreated, preserving the user's progress and preventing data loss.

Launching an Activity

In Android, you can launch an Activity from another application by using an `Intent`. The `Intent` is used to specify the package name and class name of the target Activity, as well as any additional data or flags that should be passed to the Activity.

Here's an example of how to launch an Activity from another application:
```
Intent intent = new Intent();
intent.setClassName("com.example.myapp", "com.example.myapp.MainActivity");
startActivity(intent);
```

In this example, the `setClassName()` method is used to specify the package name and class name of the target Activity. The `startActivity()` method is then called to launch the Activity.

You can also include additional data or flags in the `Intent` to pass to the target Activity. For example, to pass a String value to the target Activity, you can use the `putExtra()` method:

```
Intent intent = new Intent();
intent.setClassName("com.example.myapp", "com.example.myapp.MainActivity");
intent.putExtra("message", "Hello from another app!");
startActivity(intent);
```

In this example, the `putExtra()` method is used to add a String value with the key "message" to the `Intent`. This value can then be retrieved by the target Activity using the `getStringExtra()` method.

It's important to note that launching an Activity from another application requires that the target Activity be declared as "exported" in the manifest file of the target application. If the Activity is not exported, the system will not allow it to be launched from another application.

In Android, both startActivity() and startActivityForResult() are used to launch a new activity from an existing activity. However, they differ in how they handle the result of the launched activity.

Below is a brief overview of the differences between startActivity() and startActivityForResult() methods:

1. startActivity(): This method is used to launch a new activity from an existing activity, without expecting any result from the launched activity.
   - Once the new activity is launched, the existing activity goes into the background and the new activity becomes the foreground activity.

2. startActivityForResult(): This method is used to launch a new activity from an existing activity, with the expectation of **getting a result** back from the launched activity.
   - Once the new activity is launched, the existing activity goes into the background and the new activity becomes the foreground activity.
   - When the new activity finishes and returns a result, the onActivityResult() method of the existing activity is called, allowing it to handle the result.

In summary, the key difference between startActivity() and startActivityForResult() is that the latter method expects to receive a result from the launched activity, whereas the former does not. Use startActivity() when you simply want to launch a new activity, and use startActivityForResult() when you want to launch a new activity and receive a result back from it.

Activity Launch Modes

In Android, an Activity's launch mode determines how the system launches the Activity and manages it in the activity stack. There are four launch modes for an Activity:

1. Standard: This is the default launch mode for an Activity. Each time a new instance of the Activity is launched, a new instance is created on top of the activity stack, regardless of whether an existing instance of the Activity already exists.

2. SingleTop: In this launch mode, if an instance of the Activity already exists at the top of the activity stack, the system reuses that instance by calling its `onNewIntent()` method instead of creating a new instance. If the Activity is not at the top of the stack, a new instance is created.

3. SingleTask: In this launch mode, the system creates a new task and adds the Activity to the task as the root Activity. If an instance of the Activity already exists in the task, the system removes all Activities on top of it and reuses it by calling its `onNewIntent()` method. If no instance of the Activity exists in the task, a new instance is created.

4. SingleInstance: This is the most restrictive launch mode. In this mode, the system creates a new task and adds the Activity to the task as the root Activity. However, the system does not allow any other Activities to be launched into the same task. If an Activity with the same launch mode already exists in a separate task, the system routes the Intent to the existing task with that Activity, creating a new instance of the task if necessary.

To set the launch mode of an Activity, you can use the `android:launchMode` attribute in the Activity's manifest file. For example, to set the launch mode to SingleTop, you can add the following line to the Activity's manifest entry:

```
android:launchMode="singleTop"
```

Passing data between Activities

In Android, data can be passed between activities using various methods:

1. Intent: An Intent is a messaging object that is used to communicate between components of an application. It can be used to start a new activity, launch a service, send a broadcast, etc. To pass data between activities using an intent, you can add extra data to the intent using the `putExtra()` method. The receiving activity can then retrieve the data using the `getIntent()` method.

Here is an example of passing data using an Intent:

In the sending activity:
```
Intent intent = new Intent(this, ReceivingActivity.class);
intent.putExtra("key", "value");
startActivity(intent);
```

In the receiving activity:
```
Intent intent = getIntent();
String data = intent.getStringExtra("key");
```

2. Bundle: A Bundle is a container for a set of values. It can be used to pass data between activities using the `putExtra()` method, just like an Intent. The difference is that a Bundle can contain multiple values and can be used to pass complex data types.

Here is an example of passing data using a Bundle:

In the sending activity:
```
Intent intent = new Intent(this, ReceivingActivity.class);
Bundle bundle = new Bundle();
bundle.putString("key1", "value1");
bundle.putInt("key2", 123);
intent.putExtras(bundle);
startActivity(intent);
```

In the receiving activity:
```
Intent intent = getIntent();
Bundle bundle = intent.getExtras();
String data1 = bundle.getString("key1");
int data2 = bundle.getInt("key2");
```

3. Static variable: A static variable can be used to store data that needs to be accessed by multiple activities. The variable can be declared in a separate class and accessed by both activities.

Here is an example of passing data using a static variable:

In a separate class:
```
```

```
public class DataHolder {
    public static String data;
}
```

In the sending activity:

```
DataHolder.data = "value";
startActivity(new Intent(this, ReceivingActivity.class));
```

In the receiving activity:

```
String data = DataHolder.data;
```

Handling Back Button Press

In Android, the back button is a hardware button that is used to navigate back to the previous screen or Activity. When the back button is pressed, the system calls the `onBackPressed()` method of the current Activity by default. However, you can override this method to provide custom behavior when the back button is pressed.

Here's an example of how to override the `onBackPressed()` method in an Activity:

```
public class MyActivity extends AppCompatActivity {
    @Override
    public void onBackPressed() {
        // Add custom behavior here
        super.onBackPressed();
    }
}
```

In this example, the `onBackPressed()` method is overridden to provide custom behavior when the back button is pressed. You can add your own code in this method to handle the back button press, such as displaying a confirmation dialog or navigating back to a specific screen.

It's important to call the `super.onBackPressed()` method at the end of the method to ensure that the default behavior of the back button is still executed. This will ensure that the user can still navigate back to the previous screen or Activity if necessary.

You can also override the `onKeyDown()` method to handle the back button press. The `onKeyDown()` method is called when any key is pressed, including the back button. Here's an example of how to override the `onKeyDown()` method:

```
public class MyActivity extends AppCompatActivity {
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_BACK) {
            // Add custom behavior here
            return true;
        }
        return super.onKeyDown(keyCode, event);
    }
}
```

In this example, the `onKeyDown()` method is overridden to handle the back button press specifically. The method checks if the key code is `KeyEvent.KEYCODE_BACK` to determine if the back button was pressed. If the back button was pressed, you can add your own code to handle the press and return `true` to indicate that the back button press was handled.

Handling configuration changes of Activities

In Android, a configuration change occurs when the device's configuration, such as the screen orientation or language, changes. By default, when a configuration change occurs, the current Activity is destroyed and recreated with the new configuration. However, this can cause problems with data loss and UI flickering.

To handle configuration changes in an Activity and avoid these problems, you can override the `onSaveInstanceState()` method and the `onRestoreInstanceState()` method. These methods allow you to save and restore the state of the Activity's UI and data when a configuration change occurs.

Here's an example of how to handle configuration changes in an Activity:

```
public class MyActivity extends AppCompatActivity {
    private int mCount = 0;

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

        if (savedInstanceState != null) {
            mCount = savedInstanceState.getInt("count", 0);
        }

        TextView countTextView = findViewById(R.id.count_text_view);
        countTextView.setText(String.valueOf(mCount));
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("count", mCount);
    }

    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);
        mCount = savedInstanceState.getInt("count", 0);
    }

    public void incrementCount(View view) {
        mCount++;
        TextView countTextView = findViewById(R.id.count_text_view);
        countTextView.setText(String.valueOf(mCount));
    }
}
```

In this example, the Activity has a `TextView` that displays a count value, and a button that increments the count when clicked. The `onSaveInstanceState()` method is overridden to save the current count value in the `Bundle` object. The `onRestoreInstanceState()` method is overridden to restore the count value from the `Bundle` object.

By saving the state of the Activity's UI and data in the `onSaveInstanceState()` method and restoring it in the `onRestoreInstanceState()` method, the state can be preserved when a configuration change occurs, preventing data loss and UI flickering.

It's important to note that not all configuration changes trigger a restart of the Activity. For example, changes to the device's screen size or density do not trigger a restart. To handle these types of configuration changes, you can use the `android:configChanges` attribute in the Activity's manifest entry to specify which configuration changes you want to handle manually. For example, to handle screen orientation changes manually, you can add the following line to the Activity's manifest entry:

```
android:configChanges="orientation"
```

In summary, you can handle configuration changes in an Activity by overriding the `onSaveInstanceState()` method and the `onRestoreInstanceState()` method to save and restore the state of the Activity's UI and data. You can also use the `android:configChanges` attribute to handle configuration changes manually.

The Action bar

In Android, the ActionBar is a UI component that provides a consistent navigation and user interface experience across an app. The ActionBar typically appears at the top of the screen and contains app-specific actions, navigation options, and a title or logo.

The ActionBar was introduced in Android 3.0 (API level 11) as a replacement for the previous options menu and title bar. It provides a more modern and flexible way to display app-specific actions and navigation options.

The ActionBar can be customized to fit the design and branding of an app. You can add custom icons, change the background color, and add custom views to the ActionBar.

Here's an example of how to add an ActionBar to an Activity:

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

        ActionBar actionBar = getSupportActionBar();
        actionBar.setTitle("My App");
        actionBar.setDisplayHomeAsUpEnabled(true);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case android.R.id.home:
                onBackPressed();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

In this example, the `getSupportActionBar()` method is called to retrieve a reference to the Activity's ActionBar. The `setTitle()` method is used to set the title of the ActionBar, and the `setDisplayHomeAsUpEnabled()` method is called to enable the Up button for navigation.

The `onOptionsItemSelected()` method is overridden to handle the selection of menu items in the ActionBar. In this example, the method checks for the selection of the Up button and calls the `onBackPressed()` method to navigate back to the previous Activity.

FRAGMENTS

A **Fragment** is a piece of an activity which enable more modular activity design. It will not be wrong if we say, a fragment is a kind of **sub-activity**.

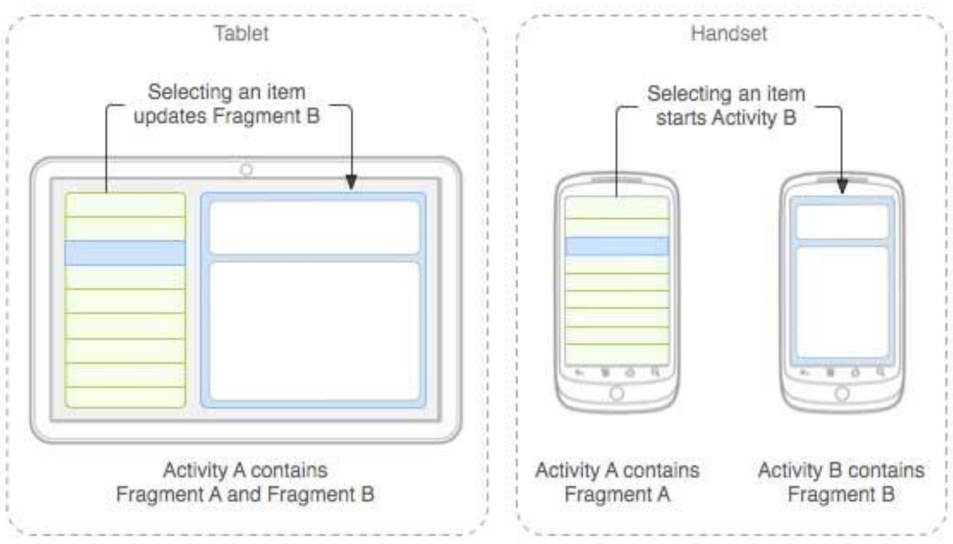
Following are important points about fragment −

- A fragment has its own layout and its own behaviour with its own life cycle callbacks.
- You can add or remove fragments in an activity while the activity is running.
- You can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- Fragment life cycle is closely related to the life cycle of its host activity which means when the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behaviour that has no user interface component.
- Fragments were added to the Android API in Honeycomb version of Android which API version 11.

You create fragments by extending **Fragment** class and You can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a **<fragment>** element.

Prior to fragment introduction, we had a limitation because we can show only a single activity on the screen at one given point in time. So we were not able to divide device screen and control different parts separately. But with the introduction of fragment we got more flexibility and removed the limitation of having a single activity on the screen at a time. Now we can have a single activity but each activity can comprise of multiple fragments which will have their own layout, events and complete life cycle.

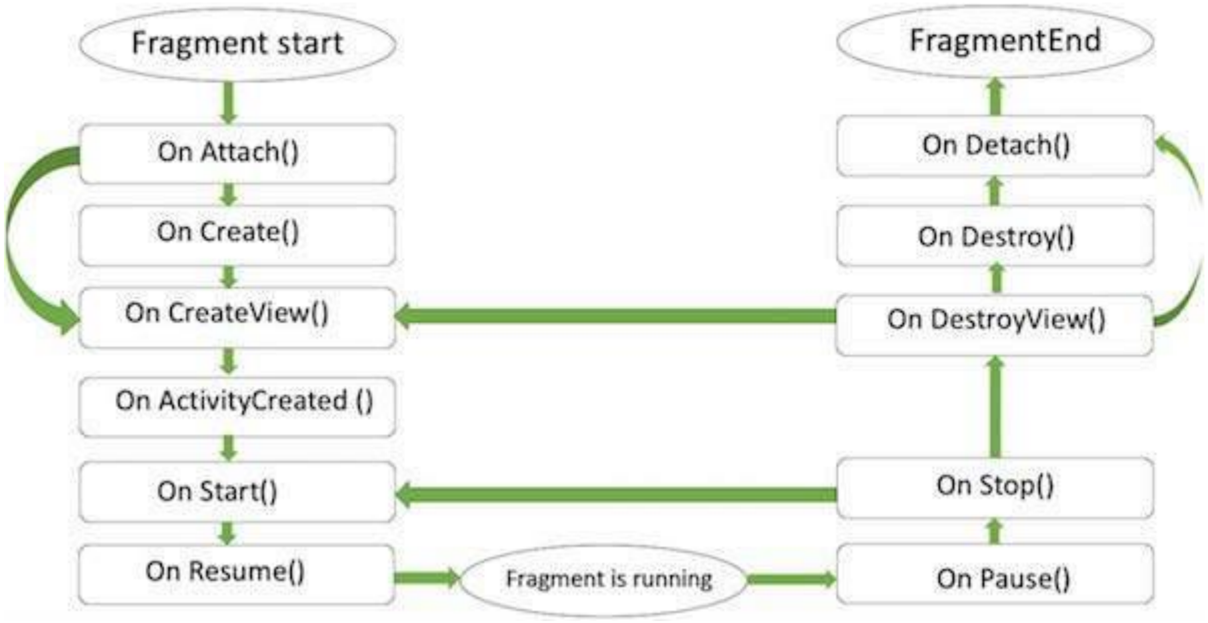
Following is a typical example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.


Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

The application can embed two fragments in Activity A, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so Activity A includes only the fragment for the list of articles, and when the user selects an article, it starts Activity B, which includes the second fragment to read the article.

Fragment Life Cycle

Android fragments have their own life cycle very similar to an android activity. This section briefs different stages of its life cycle.



Here is the list of methods which you can to override in your fragment class −

- **onAttach()**The fragment instance is associated with an activity instance.The fragment and the activity is not fully initialized. Typically you get in this method a reference to the activity which uses the fragment for further initialization work.
- **onCreate()** The system calls this method when creating the fragment. You should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView()** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a **View** component from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- **onActivityCreated()**The onActivityCreated() is called after the onCreateView() method when the host activity is created. Activity and fragment instance have been created as well as the view hierarchy of the activity. At this point, view can be accessed with the findViewById() method. example. In this method you can instantiate objects which require a Context object
- **onStart()**The onStart() method is called once the fragment gets visible.

- **onResume()**Fragment becomes active.
- **onPause()** The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session.
- **onStop()**Fragment going to be stopped by calling onStop()
- **onDestroyView()**Fragment view will destroy after call this method
- **onDestroy()**onDestroy() called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

How to use Fragments?

This involves number of simple steps to create Fragments.

- First of all decide how many fragments you want to use in an activity. For example let's we want to use two fragments to handle landscape and portrait modes of the device.
- Next based on number of fragments, create classes which will extend the *Fragment* class. The Fragment class has above mentioned callback functions. You can override any of the functions based on your requirements.
- Corresponding to each fragment, you will need to create layout files in XML file. These files will have layout for the defined fragments.
- Finally modify activity file to define the actual logic of replacing fragments based on your requirement.

Types of Fragments

Basically fragments are divided as three stages as shown below.

- Single frame fragments − Single frame fragments are using for hand hold devices like mobiles, here we can show only one fragment as a view.
- List fragments − fragments having special list view is called as list fragment
- Fragments transaction − Using with fragment transaction. we can move one fragment to another fragment.

Difference between fragments and activities

An Activity is a single screen with a user interface that the user can interact with, while a Fragment is a portion of an Activity's user interface that can be combined with other fragments to create a complete user interface.

In other words, an Activity typically represents a complete user interface that can contain multiple fragments, whereas a Fragment represents a portion of that user interface. While an Activity is standalone and can exist on its own, a Fragment must always be hosted by an Activity.

Activities are used to define the entry point of an application and handle user interactions, such as responding to button clicks or menu selections. Fragments, on the other hand, are used to modularize the user interface and make it more flexible and reusable.

Another difference between Activities and Fragments is their lifecycle. Activities have their own lifecycle, which is managed by the operating system, and they can be destroyed and recreated as needed. Fragments also have their own lifecycle, which is tied to the lifecycle of the hosting Activity, and they can be added, removed, and replaced dynamically at runtime.

Overall, Activities and Fragments serve different purposes in the Android framework, with Activities defining the overall structure and behavior of the application and Fragments providing a modular and flexible way to design user interfaces.

Review questions and Answers

Here are some common questions about Android fragments:

1. What is an Android Fragment?
A Fragment is a reusable portion of an Android user interface that can be combined with other fragments to create a single activity. It allows developers to create more dynamic and flexible user interfaces that can adapt to different screen sizes and orientations.

2. How do I create a Fragment in Android?
To create a Fragment in Android, you need to create a subclass of the Fragment class and implement the onCreateView() method to define the layout of the Fragment. You can then add the Fragment to an activity using a FragmentTransaction.

3. What is the lifecycle of an Android Fragment?
The lifecycle of an Android Fragment is similar to that of an activity. It has the following lifecycle methods: onAttach(), onCreate(), onCreateView(), onActivityCreated(), onStart(), onResume(), onPause(), onStop(), onDestroyView(), and onDestroy(). Each of these methods is called at a specific point in the Fragment's lifecycle.

4. How do I communicate between Fragments in Android?
You can communicate between Fragments in Android by using a shared ViewModel, using a callback interface, or using the FragmentManager to find the other Fragment by its tag and calling a public method on it.

5. How do I handle configuration changes with Fragments in Android?
You can handle configuration changes with Fragments in Android by using the setRetainInstance(true) method in the Fragment's onCreate() method, which will retain the Fragment instance across configuration changes. You can also save the Fragment's state in the onSaveInstanceState() method and restore it in the onCreate() method.

6. How do I add a Fragment to an Activity in Android?

To add a Fragment to an Activity in Android, you need to create an instance of the Fragment and add it to the Activity using a FragmentTransaction. You can add the Fragment to a ViewGroup using its ID or by using a tag to find the container.

7. How do I remove a Fragment from an Activity in Android?
To remove a Fragment from an Activity in Android, you need to call the remove () method on the Fragment Transaction and pass in the Fragment instance to be removed. You can also use the replace() method to replace the Fragment with another Fragment or null to remove it.

SERVICES

In many applications, certain tasks are being performed **without using any UI** i.e., the task is being performed in the background. For example, the Music app of our mobile device or any other. Music application runs in the background and while using the Music app, you can use any other application normally. So, this feature is implemented using the **Service** or **IntentService**.

It is an Android component that is used to perform some long-running operations in the **background.** The best part is that **you don't need to provide some UI** for the operations to be performed in the background. By using service, we can make inter-process communication (IPC).

Types of services

There are **many types** of services in android, the following are some of them:

a) **A started service**

A service that is started by calling the **startService**() method from another component. Once started, the service can continue to run in the background even if the component that started it is destroyed.

b) **A Bound service**

Works by binding a service to an application using **bindService()** is stopped by calling the **unbindService**() method.

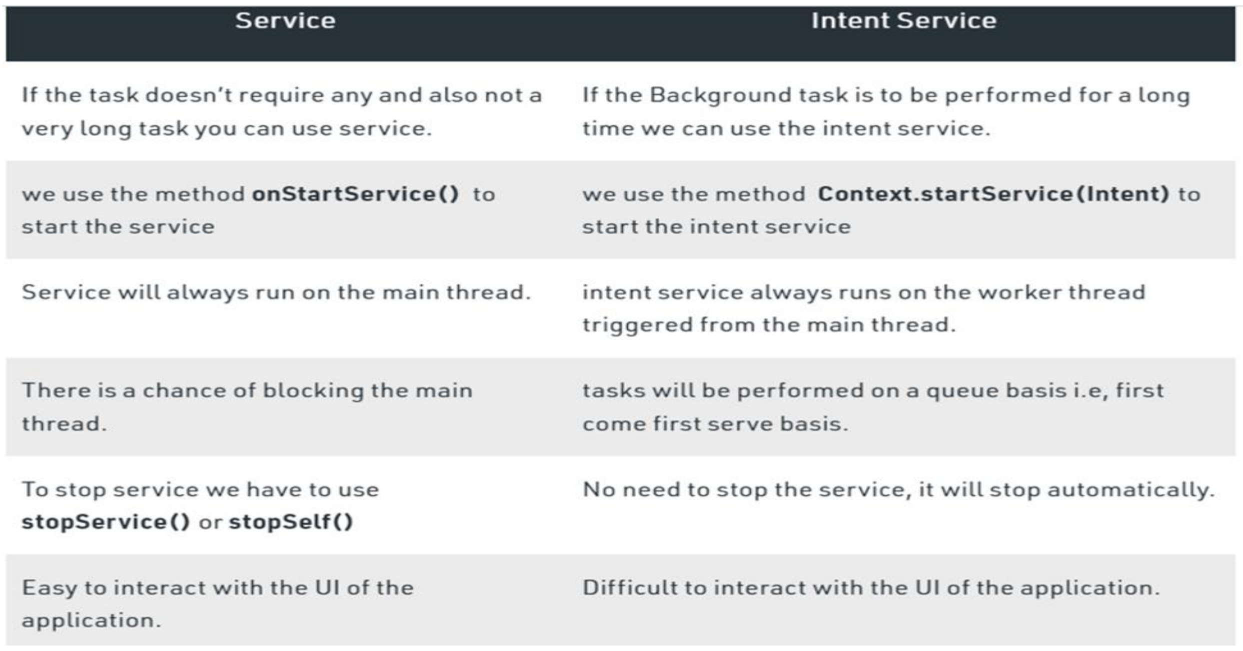
c) **A Background service**

A background service performs an **operation that isn't directly noticed by the user**. For example, if an app used a service to compact its storage, that would usually be a background service.

d) **A Foreground service**

A foreground service performs some **operation that is noticeable to the user**. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification.

Table - Difference between a Service and Intent-Service



Service	Intent Service
If the task doesn't require any and also not a very long task you can use service.	If the Background task is to be performed for a long time we can use the intent service.
we use the method **onStartService()** to start the service	we use the method **Context.startService(Intent)** to start the intent service
Service will always run on the main thread.	intent service always runs on the worker thread triggered from the main thread.
There is a chance of blocking the main thread.	tasks will be performed on a queue basis i.e, first come first serve basis.
To stop service we have to use **stopService()** or **stopSelf()**	No need to stop the service, it will stop automatically.
Easy to interact with the UI of the application.	Difficult to interact with the UI of the application.

Table – difference between main thread and a worker thread

Main thread	**Worker Thread**
● Runs the UI and handles user input	● It is additional thread to handle workload of services
● Performing long-running tasks on main thread is bad experience, and causes UI to freeze or bring ANR error	● Run separately and can perform intensive operations without affecting the UI performance

Creating a service

To create an Android Service, you can follow the following steps:

- Create a new Android project in Android Studio.
- Create a new Java class for your Service by right-clicking on your app package in the Project Explorer, selecting New -> Java Class, and giving it a name.
- In your Service class, extend the Android **Service** class by adding "extends Service" to the class declaration line.
- Override the **onStartCommand**() method in your Service class. This method is called when the Service is started and should contain the code that you want to execute when the Service is started.
- **Implement** all the methods that you have overridden by adding the necessary code to each one.
- Declare your Service in the **AndroidManifest**.xml file by adding a <service> element inside the <application> element. Make sure to include the name of your Service class as the "android:name" attribute of the <service> element.

Here is an example code snippet for a basic Service class:

```
public class MyService extends Service {

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      // Code to execute when Service is started
      return START_STICKY;
   }

   @Override
   public IBinder onBind(Intent intent) {
      // Not used in this Service
      return null;
   }

   @Override
   public void onDestroy() {
      // Code to execute when Service is destroyed
      super.onDestroy();
   }
}
```

Note: Remember to declare your Service in the AndroidManifest.xml file, otherwise it won't work.

Stopping services

To stop an Android service, you can call the `stopService()` method or the `stopSelf()` method from within the service itself.

Here's how you can use the `stopService()` method to stop a service:

1. Create an `Intent` that identifies the service you want to stop.
   ```
   Intent intent = new Intent(context, MyService.class);
   ```

2. Call `stopService()` and pass in the `Intent`.
   ```
   stopService(intent);
   ```

This will stop the service if it's currently running. If the service is not running, calling `stopService()` has no effect.

Here's how you can use the `stopSelf()` method to stop a service from within the service itself:

1. Call `stopSelf()` from within the service.
   ```
   stopSelf();
   ```

This will stop the service if it's currently running.

Note that if you want the service to stop itself automatically when it has completed its work, you can call `stopSelf()` from within the `onDestroy()` method of the service. This method is called when the service is about to be destroyed and is a good place to do any cleanup work that needs to be done.

BROADCAST RECEIVER

Broadcast Receiver is a dormant component of android that listens to system-wide broadcast events or intents. When any of these events occur it brings the application into action by either creating a status bar notification or performing a task. If an application wants to receive and respond to a global event, such as a ringing phone or an incoming text message, it must register as a Broadcast Receiver.

The following are some of the important system wide generated intents:

1. **android.intent.action.BATTERY_LOW** : Indicates low battery condition on the device.
2. **android.intent.action.BOOT_COMPLETED** : This is broadcast once, after the system has finished booting
3. **android.intent.action.CALL** : To perform a call to someone specified by the data
4. **android.intent.action.DATE_CHANGED** : The date has changed
5. **android.intent.action.REBOOT** : Have the device reboot
6. **android.net.conn.CONNECTIVITY_CHANGE** : The mobile network or wifi connection is changed(or reset)

```
public class MyReceiver extends BroadcastReceiver {

    public MyReceiver() {  }

    @Override

    public void onReceive(Context context, Intent intent)

    {

    Toast.makeText(context,"Action:"+intent.getAction(),

    Toast.LENGTH_SHORT).show();
```

```
        }
}
```

Registering broadcast recievers

A BroadcastReceiver can be registered in two ways.

1) By defining it in the AndroidManifest.xml file as shown below.

```
<receiver android:name=".ConnectionReceiver" >
            <intent-filter>
    <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
 </intent-filter>
</receiver>
```

2) By defining it programmatically

```
IntentFilter filter = new IntentFilter();

intentFilter.addAction(getPackageName()+"android.net.conn.CONNECTIVITY_CHANGE"
);

MyReceiver myReceiver = new MyReceiver();

registerReceiver(myReceiver, filter);
```

Like Services, BroadcastReceivers **don't have a UI**. Even more important, the code running in the onReceive method of a BroadcastReceiver **should make no assumptions about persistence or long-running operations**.

Creating custom broadcast reciever

To define a custom `BroadcastReceiver` in your Android app, you need to create a new class that extends the `BroadcastReceiver` class and override the `onReceive()` method. The `onReceive()` method is called when the `BroadcastReceiver` receives a broadcast.

Here are the steps to define a custom `BroadcastReceiver`:

1. Create a new class that extends `BroadcastReceiver`.
  ```
  ```

```
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // ...
    }
}
```

2. Override the `onReceive()` method. This method is called when the `BroadcastReceiver` receives a broadcast. You can use the `Context` and `Intent` parameters to access information about the broadcast and perform any necessary actions in response.
```
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    if (action.equals("com.example.myapp.MY_ACTION")) {
        // Do something in response to the broadcast
    }
}
```

3. Register the `BroadcastReceiver` in your app. You can do this either in the manifest file or in code using the `registerReceiver()` method.

   To register the `BroadcastReceiver` in the manifest file, add a `<receiver>` element to your manifest file with the appropriate attributes:
```
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="com.example.myapp.MY_ACTION" />
    </intent-filter>
</receiver>
```

   To register the `BroadcastReceiver` in code, create an instance of your `BroadcastReceiver` class and call `registerReceiver()` on your `Context` object:
```
MyReceiver receiver = new MyReceiver();
IntentFilter filter = new IntentFilter("com.example.myapp.MY_ACTION");
context.registerReceiver(receiver, filter);
```

Once you have defined your custom `BroadcastReceiver`, you can send broadcasts to it from other components in your app using the `sendBroadcast()` method. When a broadcast is sent, your `BroadcastReceiver`'s `onReceive()` method will be called, allowing you to perform any necessary actions in response.

Sending broadcast messages from an app

To send a broadcast from your Android app, you can create an `Intent` object and call the `sendBroadcast()` method on your `Context` object. Here are the steps:

1. Create an `Intent` object. You should specify the action that you want to broadcast using the `setAction()` method. You can also include any other data that you want to send with the broadcast using the various `putExtra()` methods.

```
Intent intent = new Intent();
intent.setAction("com.example.myapp.MY_ACTION");
intent.putExtra("message", "Hello, world!");
```

2. Call the `sendBroadcast()` method on your `Context` object, passing in the `Intent` as a parameter.

```
context.sendBroadcast(intent);
```

When you call `sendBroadcast()`, the Android system will broadcast the `Intent` to all registered `BroadcastReceiver` components that have a matching intent filter. Your `BroadcastReceiver`'s `onReceive()` method will be called if it matches the intent filter.

Note that you can also use other methods to send broadcasts, such as `sendOrderedBroadcast()` or `sendStickyBroadcast()`, depending on your requirements. However, these methods are less commonly used and have some limitations, so `sendBroadcast()` is usually the best choice for most use cases.

Receiving a broadcast message

To receive a broadcast in your Android app, you need to create a `BroadcastReceiver` class and register it to receive broadcasts with an appropriate intent filter.

Here are the steps to receive a broadcast in your app:

1. Create a new class that extends `BroadcastReceiver`.

```
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Handle the broadcast here
    }
}
```

2. Override the `onReceive()` method. This method is called when the `BroadcastReceiver` receives a broadcast. You can use the `Context` and `Intent` parameters to access information about the broadcast and perform any necessary actions in response.

3. Register the `BroadcastReceiver` in your app. You can do this either in the manifest file or in code using the `registerReceiver()` method.

To register the `BroadcastReceiver` in the manifest file, add a `<receiver>` element to your manifest file with the appropriate attributes:

```
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="com.example.myapp.MY_ACTION" />
    </intent-filter>
</receiver>
```

In this example, the `MyReceiver` class is registered to receive broadcasts with the action `com.example.myapp.MY_ACTION`.

To register the `BroadcastReceiver` in code, create an instance of your `BroadcastReceiver` class and call `registerReceiver()` on your `Context` object:

```
MyReceiver receiver = new MyReceiver();
IntentFilter filter = new IntentFilter("com.example.myapp.MY_ACTION");
context.registerReceiver(receiver, filter);
```

In this example, the `MyReceiver` class is registered to receive broadcasts with the action `com.example.myapp.MY_ACTION`.

Once you have registered your `BroadcastReceiver`, it will receive broadcasts with the specified action. When a broadcast is received, your `BroadcastReceiver`'s `onReceive()` method will be called, allowing you to perform any necessary actions in response.

Note that you can also use intent filters to specify additional criteria for the broadcasts that your `BroadcastReceiver` should receive, such as data URI or MIME type. This allows you to receive only the broadcasts that are relevant to your app.

Exchanging data between broadcast recievers and other components

To pass data between a `BroadcastReceiver` and other components in your Android app, you can use various methods depending on your requirements. Here are some commonly used methods:

1. Using Intent extras: You can include data in the `Intent` that is broadcasted from your `BroadcastReceiver` using the `putExtra()` method. Other components that receive the broadcast can then retrieve the data from the `Intent` using the corresponding `getExtra()` method.

For example, in your `BroadcastReceiver` you can include data like this:

```
Intent intent = new Intent("com.example.myapp.MY_ACTION");
intent.putExtra("message", "Hello, world!");
sendBroadcast(intent);
```
And in the component that receives the broadcast, you can retrieve the data like this:
```
String message = intent.getStringExtra("message");
```

2. Using the Application class: If you need to share data between different components of your app, you can store the data in the `Application` class. The `Application` class is a singleton that is created when your app starts, and can be accessed from any component in your app.

For example, in your `Application` class you can define a public method to set and get data:
```
public class MyApp extends Application {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```
And in your `BroadcastReceiver` or other components, you can access the data like this:
```
MyApp app = (MyApp) getApplication();
String message = app.getMessage();
```

3. Using a local broadcast: If you need to pass data between components within your app only, you can use a local broadcast instead of a global broadcast. Local broadcasts are delivered only to components within your own app, and are not visible to other apps.

To send a local broadcast, you can use the `LocalBroadcastManager` class:
```
Intent intent = new Intent("com.example.myapp.MY_LOCAL_ACTION");
intent.putExtra("message", "Hello, world!");
LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
```
And to receive a local broadcast, you can register a `BroadcastReceiver` with the `LocalBroadcastManager` instead of the global `BroadcastManager`:

```
BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String message = intent.getStringExtra("message");
        // Do something with the message
    }
};
IntentFilter filter = new IntentFilter("com.example.myapp.MY_LOCAL_ACTION");
LocalBroadcastManager.getInstance(context).registerReceiver(receiver, filter);
```

Note that you need to use the `LocalBroadcastManager` to register and unregister the `BroadcastReceiver` for local broadcasts, instead of the global `BroadcastManager`.

Handling different types of broadcasts

To handle different types of broadcasts in your `BroadcastReceiver`, you can use the `IntentFilter` class to specify the actions that your `BroadcastReceiver` should receive. An intent filter is a set of criteria that specifies the type of broadcast that your `BroadcastReceiver` is interested in.

Here are the steps to handle different types of broadcasts in your `BroadcastReceiver`:

1. Define an action string for each type of broadcast that your `BroadcastReceiver` should handle. You can define these as constants in your code:

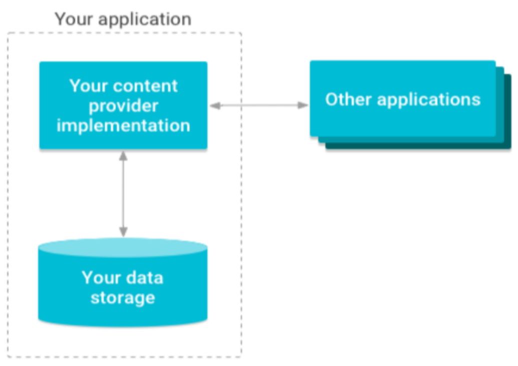
```
public static final String ACTION_ONE = "com.example.myapp.ACTION_ONE";
public static final String ACTION_TWO = "com.example.myapp.ACTION_TWO";
```

2. Create an `IntentFilter` object and add the actions that your `BroadcastReceiver` should handle using the `addAction()` method:

```
IntentFilter filter = new IntentFilter();
filter.addAction(ACTION_ONE);
filter.addAction(ACTION_TWO);
```

3. Override the `onReceive()` method in your `BroadcastReceiver` and check the action of the received `Intent` using the `getAction()` method. You can then perform any necessary actions based on the action of the received `Intent`:

```
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    if (action.equals(ACTION_ONE)) {
```

        // Handle the first type of broadcast
      } else if (action.equals(ACTION_TWO)) {
        // Handle the second type of broadcast
      }
    }
  ```

4. Register your `BroadcastReceiver` with the appropriate `IntentFilter` using the `registerReceiver()` method:
  ```
  MyReceiver receiver = new MyReceiver();
  IntentFilter filter = new IntentFilter();
  filter.addAction(MyReceiver.ACTION_ONE);
  filter.addAction(MyReceiver.ACTION_TWO);
  context.registerReceiver(receiver, filter);
  ```

When a broadcast is received, the `onReceive()` method of your `BroadcastReceiver` will be called. You can then check the action of the received `Intent` and perform any necessary actions based on the action.

Note that you can also use other criteria in your `IntentFilter`, such as data URI or MIME type, to further specify the type of broadcast that your `BroadcastReceiver` should handle. This allows you to receive only the broadcasts that are relevant to your app.

## CONTENT PROVIDERS

Content Providers manage access to data stored within the application. Content providers are the standard interface that connects data in one process with code running in another process. Implementing a content provider has many advantages. Most importantly you can configure a content provider to allow other applications to securely access and modify your app data.



These components work together seamlessly to create an engaging and functional mobile application.
  ```

Understanding how these components interact with each other is crucial to deliver high-quality apps that meet the needs of users.

Here are some common questions that are usually asked about Android Content Providers:

Review questions and answers

1. What is a Content Provider in Android?

A Content Provider in Android is a component that manages access to a shared set of app data. It provides a standard interface for other apps to query, insert, update, and delete data in your app's database. Content Providers are used to securely share data between different apps in the Android system.

2. What are the benefits of using a Content Provider?

Using a Content Provider provides several benefits, including:

- Securely exposing app data to other apps in the Android system.

- Providing a standardized interface for other apps to access your app's data.

- Enforcing data access policies, such as read-only or write-only access.

- Allowing your app to be integrated with other apps in the Android system, such as the Contacts app or the Gallery app.

3. How do I create a Content Provider in Android?

To create a Content Provider in Android, you need to define a subclass of the `ContentProvider` class and implement the necessary methods, such as `query()`, `insert()`, `update()`, and `delete()`. You also need to define a `URI` scheme that identifies the data that your Content Provider manages.

4. How do I access data from a Content Provider in my app?

To access data from a Content Provider in your app, you need to use a `ContentResolver` object. You can use the `ContentResolver` to query, insert, update, and delete data in the Content Provider's database. You also need to define the `URI` scheme that identifies the data you want to access.

5. How do I secure my Content Provider?

To secure your Content Provider, you can define permissions that restrict access to specific data or operations. You can also use a `ProviderInfo` object to specify the permissions that other apps need to access your Content Provider.

6. How do I test my Content Provider?

To test your Content Provider, you can use unit tests to verify that the `query()`, `insert()`, `update()`, and `delete()` methods work correctly. You can also use integration tests to verify that your Content Provider works correctly with other apps in the Android system.

7. How do I optimize the performance of my Content Provider?

To optimize the performance of your Content Provider, you can use techniques such as using indexes and caching to improve query performance. You can also use a `CursorLoader` to asynchronously load data from the Content Provider in the background, which can help to improve the responsiveness of your app.

LIFECYCLE METHODS OF APPLICATION COMPONENTS

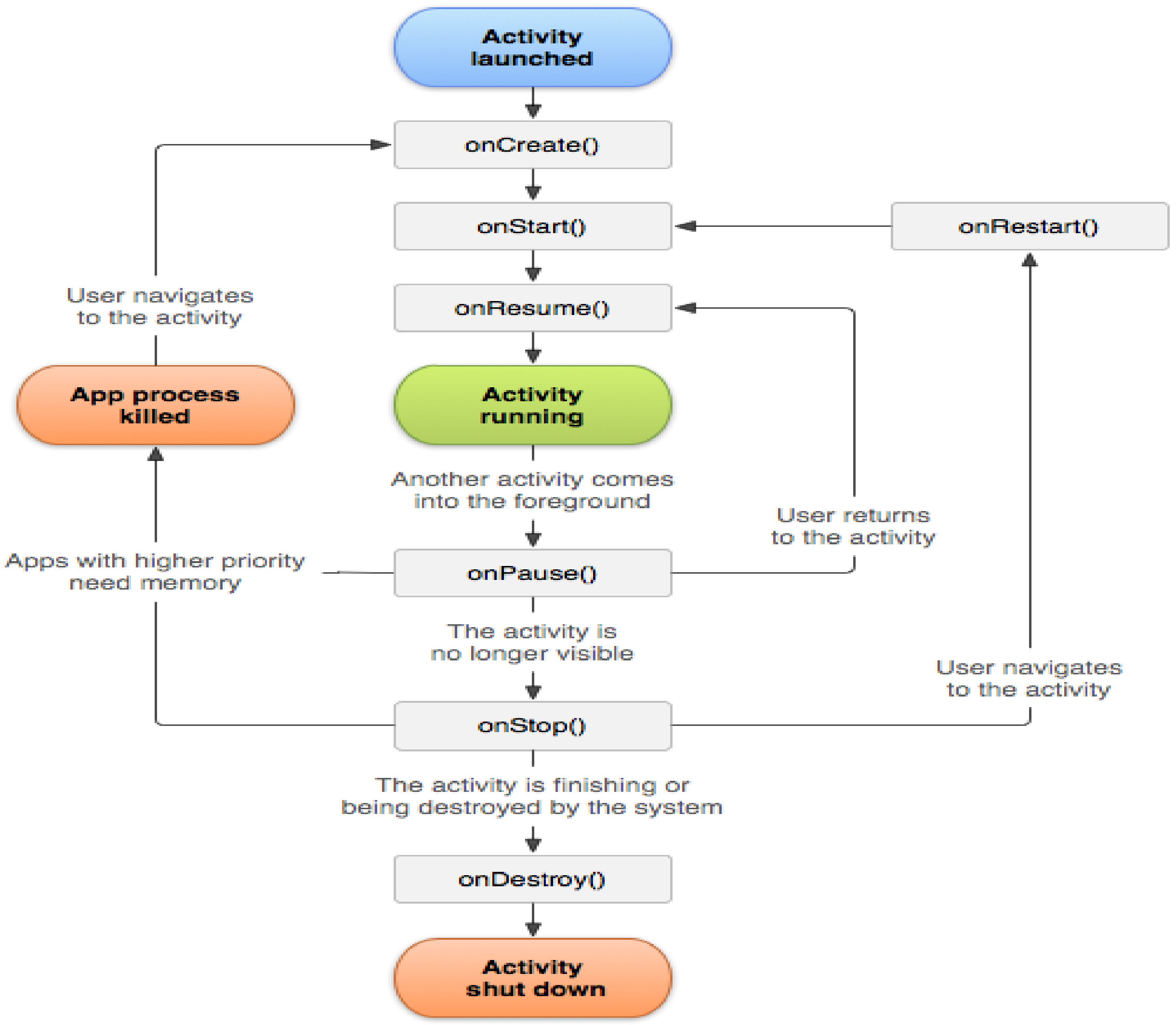
ANDROID ACTIVITIES LIFE CYCLE

Android activities are the building blocks of any Android application. They are responsible for managing the UI and user interactions. The lifecycle methods of activities govern the behavior of the activities during runtime.

The lifecycle of an activity **begins when it is first created** and **ends when it is destroyed**. During this lifecycle, the activity goes through various states, and the lifecycle methods are called accordingly.

The following are the seven lifecycle methods of an activity:

1. onCreate(): This is the first method that is called when the activity is created. This method is responsible for **initializing the activity** and **setting up the UI**.
2. onStart(): This method is called when the activity becomes visible to the user. This is the point where the **activity is ready to interact with the user**.
3. onResume(): This method is called when the activity is **in the foreground** and **has focus**. At this point, the activity is fully interactive, and the user can interact with the UI.
4. onPause(): This method is called when the **activity loses focus** but is **still visible to the user**. This can happen when another activity is launched on top of the current activity.
5. onStop(): This method is called when the activity is **no longer visible to the user**. This can happen when the user navigates to another activity or when the activity is destroyed.
6. onRestart(): This method is called when the **activity is stopped and then started again**. This can happen when the user navigates back to the activity.
7. onDestroy(): This is the final method that is called when the activity is destroyed. This method is **responsible for releasing any resources** that were allocated by the activity.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

.It is important to note that the lifecycle methods are not always called in a predictable order.

ANDROID SERVICES LIFE CYCLE

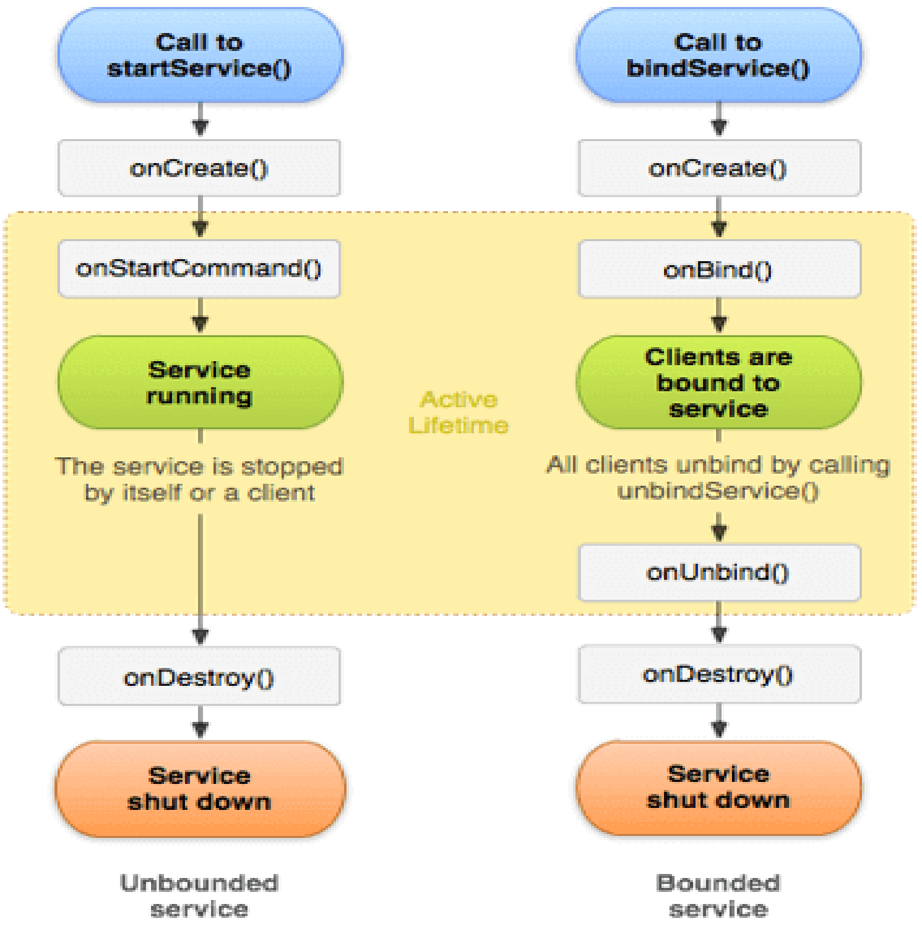
Android services are components that run in the background and perform tasks without requiring user interaction. Services can be used to perform tasks such as playing music, downloading files, or updating data in the background.

The following are the four lifecycle methods of an Android service:

1. onCreate(): This is the first method that is called when the service is created. This method is responsible for **initializing** the service and setting up any resources that are needed.

2. onStartCommand(): This method is called when the service is started. This method is responsible for **performing the task** that the service was created for. This method can also return a value that specifies how the system should handle the service if it is killed.
3. onHandleIntent(): This method is used to add any code you want the service to do in the background here.
4. onDestroy(): This is the final method that is called when the service is destroyed. This method is responsible for **releasing** any resources that were allocated by the service.

It is important to note that services can be **started or bound**.



When a service is started, it runs independently of the activity that started it.

When a service is bound, it runs in the same process as the activity that bound it, and the activity can communicate with the service.

When a service is started, it will continue to run until it is stopped by calling the **stopService()** method.

When a service is bound, it will continue to run until all clients unbind from it by calling the **unbindService()** method.

It is important to note that services can run indefinitely, so developers must be careful not to create services that consume too many resources.

ANDROID BROADCAST RECEIVERS LIFE CYCLE

Android Broadcast Receivers are components that enable the system to deliver events or messages to the application outside of a regular user interface. They are used to listen to system-wide events, such as the completion of a phone call or the receipt of a text message. Understanding the lifecycle methods of broadcast receivers is essential for building robust and efficient Android applications.

The following are the two lifecycle methods of an Android Broadcast Receiver:

1. onReceive(): This is t**he only method that is called when a broadcast event is received by the receiver**. This method is responsible for **performing the task** that the receiver was created for.
2. onDestroy(): This method is called when the receiver is destroyed. This method is responsible for **releasing** any resources that were allocated by the receiver.

It is important to note that broadcast receivers are typically registered **dynamically at runtime** and **are not always active**. When a broadcast event is received by the receiver, the system wakes up the receiver if it is not already active.

Developers must be careful not to create receivers that consume too many resources, as it can result in poor performance and battery life. To avoid this, receivers can be registered with certain restrictions, such as only running when the device is charging or when the screen is on.

It is important to note that broadcast receivers are intended for short-lived operations, and any long-running operations should be performed by services.

ANDROID CONTENT PROVIDERS LIFE CYCLE

Android Content Providers are components that enable applications to share data with other applications. They provide a standard interface to access a shared data set, such as a database. Understanding the lifecycle methods of content providers is essential for building robust and efficient Android applications.

The following are the four lifecycle methods of an Android Content Provider:

1. onCreate(): This is the first method that is called when the provider is created. This method is responsible for **initializing** the provider and setting up any resources that are needed.
2. query(): This method is called when an application **requests data** from the provider. This method is responsible for returning a **cursor** object that contains the requested data.
3. insert(): This method is called when an application wants to **add data** to the provider. This method is responsible for inserting the data into the provider's data set.

4. delete(): This method is called when an application wants to **delete data** from the provider. This method is responsible for deleting the specified data from the provider's data set.

It is important to note that Content Providers are typically used in conjunction with other Android components, such as Activities and Services. Content Providers can also be used to share data between different applications. Developers must be careful when designing Content Providers, as they can expose sensitive data to other applications if not properly secured.

It is important to note that Content Providers can be resource-intensive, so developers must be careful not to create providers that consume too many resources.

BASICS OF EVENT HANDLING IN ANDROID

Event handling is an essential part of building Android applications. It refers to the process of **detecting and responding to user interactions** with the UI, such as clicking a button or swiping a screen. In Android, event handling is done through the use of **event listeners and event handlers**.

Event listeners are **objects** that are attached to UI components, such as buttons and text fields. When a user interacts with a UI component, the listener detects the event and triggers the appropriate event handler. E.g. Button

Event handlers are **methods** that are defined in the code and are responsible for responding to specific events. E.g. onClickListener(context)

Event Listeners Registration is the process by which an Event Handler gets registered with an Event Listener so that the handler is called when the Event Listener fires the event.

For example, if a user clicks a button, the event listener attached to the button detects the click event and triggers the appropriate event handler. The event handler then performs the desired action, such as displaying a message or launching another activity.

In Android, there are several types of events that can be handled, including:

1. Click events: These are triggered when a user clicks a button or other UI component.
2. Touch events: These are triggered when a user touches the screen, such as swiping or scrolling.
3. Key events: These are triggered when a user presses a key on the device's keyboard.
4. Focus events: These are triggered when a UI component gains or loses focus.

EVENT LISTENERS REGISTRATION

Events Listeners can be registered in **two** (2) ways:

a) Inside the xml layout code

In Android, it is possible to register event listeners for views directly in the XML layout file using the `android:onClick` attribute. This can be a convenient way to register simple event listeners without having to write any Java code.

Here's an example of how to register a click listener for a button directly in the XML layout:

```
<Button

    android:id="@+id/myButton"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Click me!"

    android:onClick="onButtonClick"/>
```

In this example, the `android:onClick` attribute is set to "onButtonClick", which is the name of a method that will be called when the button is clicked. To define the method, you can add the following code to your activity:

```
public void onButtonClick(View view) {

    // Do something when the button is clicked

}
```

Note that the method must be public, have a void return type, and take a single `View` parameter. The parameter represents the view that was clicked, which is passed automatically by the system when the method is called.

This approach can be a handy way to register simple event listeners, but it does have some limitations. For example, it only works for certain types of events (such as clicks), and it can be more difficult to handle more complex events or perform more advanced logic. For those cases, it may be necessary to register the event listener in Java code instead.

b) By implementing event listener interfaces (in the java code)

Can be achieved

- Using an Anonymous Inner Class
- Activity class implements the Listener interface.

In Android, you can register an event listener by implementing listener interfaces in your Java code. This approach allows you to handle events more flexibly and perform more advanced logic than using the XML `android:onClick` attribute.

Here's an example of how to register a click listener for a button by implementing the `View.OnClickListener` interface:

```

Button myButton = findViewById(R.id.myButton);

myButton.setOnClickListener(new View.OnClickListener() {

   @Override

   public void onClick(View v) {

      // Do something when the button is clicked

   }

});
```

In this example, a reference to the button is obtained using `findViewById`, and a new `View.OnClickListener` instance is created and passed to the `setOnClickListener` method. The `onClick` method of the `View.OnClickListener` interface is then implemented to define what should happen when the button is clicked.

You can also implement other listener interfaces to handle different types of events. For example, to handle long clicks, you can implement the `View.OnLongClickListener` interface:

```

Button myButton = findViewById(R.id.myButton);
```

```
myButton.setOnLongClickListener(new View.OnLongClickListener() {

    @Override

    public boolean onLongClick(View v) {

        // Do something when the button is long-clicked

        return true; // true to indicate that the event has been consumed

    }

});
```

In this example, the `onLongClick` method of the `View.OnLongClickListener` interface is implemented to define what should happen when the button is long-clicked. The method returns `true` to indicate that the event has been consumed, which prevents other listeners from receiving the event.

Using listener interfaces allows you to handle events more flexibly and perform more advanced logic than using the XML `android:onClick` attribute. It also allows you to handle a wider range of events than what is supported by the `android:onClick` attribute.

BASICS OF GRAPHICS AND MULTIMEDIA SUPPORT IN ANDROID

Android provides robust support for graphics and multimedia, allowing developers to create rich and engaging applications. Multimedia support has moved from OpenCORE to Stagefright as of Android 3.0. Some of the key features of graphics and multimedia support in Android include:

1. 2D and 3D graphics: Android provides support for both 2D and 3D graphics, with libraries such as OpenGL ES and Canvas.

   The graphics package(android.graphics) supports such things as bitmaps (which hold pixels), canvases (what your draw calls draw on), primitives (such as rectangles and text), and paints (which you use to add color and styling).

   Drawing graphics is possible in two ways:
   a) Using java's ShapeDrawable class
      Use ShapeDrawable class to support including shape , then use Shape which support general shape types, or use RectShape(), for rectangle shape etc.
      ShapeDrawable mDrawable = new ShapeDrawable();
      this.mDrawable = new ShapeDrawable(new RectShape());
   b) Using xml

With Android XML drawable shapes, the default is a rectangle, but you can choose a different shape by using the type tag and selecting the value oval, rectangle, line, or arc. To use your XML shape, you need to reference it in a layout.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
     <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
     <ImageView android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/simplerectangle" />
     </LinearLayout>
</ScrollView>
```

2. Image and video playback: Android provides support for playing back images and videos in a variety of formats, including JPEG, PNG, and MP4.

   Use the method called
   takePicture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback jpeg) for taking images.

   Playing a video is slightly more complicated than playing audio with the MediaPlayer API, in part because you have to provide a view surface for your video to play on. Android has a VideoView widget that handles that task for you; you can use it in any layout manager.

   <VideoView android:id="@+id/video"
   android:layout_width="320px"
   android:layout_height="240px" />

3. Audio playback: Android provides support for playing back audio in a variety of formats, including MP3 and AAC.

   Probably the most basic need for multimedia on a cell phone is the ability to play audio files, whether new ringtones, MP3s, or quick audio notes. Android's MediaPlayer is easy to use. At a high level, all you need to do to play an MP3 file is follow these steps:
   a. Put the MP3 in the res/raw directory in a project (note that you can also use a URI to access files on the network or via the internet).
   b. Create a new instance of the MediaPlayer, and reference the MP3 by calling MediaPlayer.create().
   c. Call the MediaPlayer methods prepare() and start().

   MediaPlayer mp = MediaPlayer.create(this, R.raw.halotheme);

mp.start();

By using these features, developers can create applications that are more visually appealing and engaging for users. It is important to note that graphics and multimedia can be resource-intensive, so developers must be careful not to create applications that consume too many resources.

BASICS OF DATA STORAGE IN ANDROID

There are different Android data storage options depending on data privacy and storage space requirements in android.
One can choose among the following data saving options:

SHARED PREFERENCES

Shared preference stores key-value pair of data.

The SharedPreferences class provides a general framework to store and retrieve persistent key-value pairs of data types. You can save any primitive data types like Booleans, floats, ints, longs and strings. The data will persist across user sessions (even if the app is killed).

Below are the steps for working with shared preferences in Android:

i. Create a SharedPreferences instance:
To create a SharedPreferences instance, you need to call the getSharedPreferences() method and pass it a name for the shared preferences and a mode. The mode specifies the access mode, which can be private or public.

```
SharedPreferences sharedPreferences =
getSharedPreferences("MyAppPreferences", Context.MODE_PRIVATE);
```

ii. Write data to shared preferences:
To write data to shared preferences, you need to use the edit() method to get an instance of the SharedPreferences.Editor class, and then call the putXXX() method to store the data. XXX can be any data type like String, Boolean, Int, Float, Long, etc. Finally, call the commit() method to save the data.

```
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("username", "John");
editor.putInt("age", 30);
editor.putBoolean("is_logged_in", true);
editor.commit();
```

iii. Read data from shared preferences:

To read data from shared preferences, you need to use the getXXX() method, where XXX is the data type you want to retrieve. If the key doesn't exist, it will return a default value.

```
String username = sharedPreferences.getString("username", "");
int age = sharedPreferences.getInt("age", 0);
boolean isL = sharedPreferences.getBoolean("is_logged_in", false);
```

iv. Remove data from shared preferences:
To remove data from shared preferences, you need to call the remove() method and pass it the key for the data you want to remove. Finally, call the commit() method to save the changes.

```
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.remove("username");
editor.commit();
```

It's important to note that shared preferences are not suitable for storing large amounts of data or sensitive data like passwords. It's best to use other storage options like internal storage or SQLite databases for those purposes.

Internal storage

Used to store private data on device memory.

To create and write a private file in storage - Call openFileOutput( ) with name of file and mode, Write to the file with write( ), Close the stream with close( ).
To read a file , Call openFileInput( ) with name of file, Read bytes from file using read() Close the stream with close( ).

External storage

Stores public data on shared external storage

External storage can be removable media, an internal non-removable media.
Caution: External files can disappear if the user mounts the external storage on a computer or removes the media, before you work with it; check media availability by calling getExternalStorageState( ). The media might be mounted on a computer and it might be; Missing , Readonly , or in some other state.

Use getExternalFilesDir() to open a file that represents an external storage directory where you saved your file. This method takes a type parameter to specify the type of subdirectory you want.

For Example:
DIRECTORY_MUSIC for music files

DIRECTORY_RINGTONES for ringtone files
null – to access the root directory

SQLITE DATABASE

Stores a structured data in private database. Android provides full support for SQLite database.
Working with SQLite databases in Android involves the following steps:

1. Create a database helper class:
   To work with SQLite databases, you need to create a subclass of the SQLiteOpenHelper class. This class provides methods for creating, upgrading, and downgrading the database schema.

```
public class MyDatabaseHelper extends SQLiteOpenHelper {
   private static final int DATABASE_VERSION = 1;
   private static final String DATABASE_NAME = "MyDatabase.db";

   public MyDatabaseHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      // Create tables and initial data
   }

   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      // Upgrade database schema
   }
}
```

2. Create tables and initial data:
   In the onCreate() method of the database helper class, you can create tables and add initial data to the database.

```
@Override
public void onCreate(SQLiteDatabase db) {
   db.execSQL("CREATE TABLE users (_id INTEGER PRIMARY KEY, name
TEXT, email TEXT)");
   db.execSQL("INSERT INTO users (name, email) VALUES ('John',
'john@example.com')");
}
```

3. Query data from the database:

To query data from the database, you need to create a SQLiteDatabase instance and use the query() or rawQuery() method to execute a SQL query.

```
SQLiteDatabase db = myDatabaseHelper.getReadableDatabase();
Cursor cursor = db.rawQuery("SELECT * FROM users", null);
while (cursor.moveToNext()) {
   int id = cursor.getInt(cursor.getColumnIndex("_id"));
   String name = cursor.getString(cursor.getColumnIndex("name"));
   String email = cursor.getString(cursor.getColumnIndex("email"));
   // Do something with the data
}
cursor.close();
```

4. Insert, update, or delete data from the database:
   To insert, update, or delete data from the database, you can use the insert(), update(), or delete() methods of the SQLiteDatabase class.

```
SQLiteDatabase db = myDatabaseHelper.getWritableDatabase();
ContentValues values = new ContentValues();
values.put("name", "Mary");
values.put("email", "mary@example.com");
long newRowId = db.insert("users", null, values);
```

5. Close the database:
   To close the database, you need to call the close() method of the SQLiteDatabase class.

```
db.close();
```

It's important to note that when working with SQLite databases, you should use parameterized queries to protect against SQL injection attacks.

NETWORK CONNECTION/CLOUD STORAGE

To store data on the web server/cloud e.g firebase database
Firebase provides a cloud-based storage solution that allows you to store and sync data across multiple devices in real-time. Here are the basic steps for working with Firebase network data storage in Android:

1. Set up Firebase: To use Firebase, you need to create a Firebase project and add the Firebase SDK to your Android app. You can follow the instructions in the Firebase documentation to set up Firebase for Android.

2. Initialize the Firebase database: To initialize the Firebase database, you need to call the FirebaseDatabase.getInstance() method and pass it the Firebase app instance.

```java
FirebaseDatabase database = FirebaseDatabase.getInstance();
```

3. Write data to the database: To write data to the Firebase database, you need to get a reference to the database location where you want to store the data. You can do this by calling the child() method on the database instance and passing it the path to the location.

```java
DatabaseReference myRef = database.getReference("users");
myRef.child("userId1").setValue("John");
```

4. Read data from the database: To read data from the Firebase database, you can use the addValueEventListener() method to listen for changes to the data at a particular location.

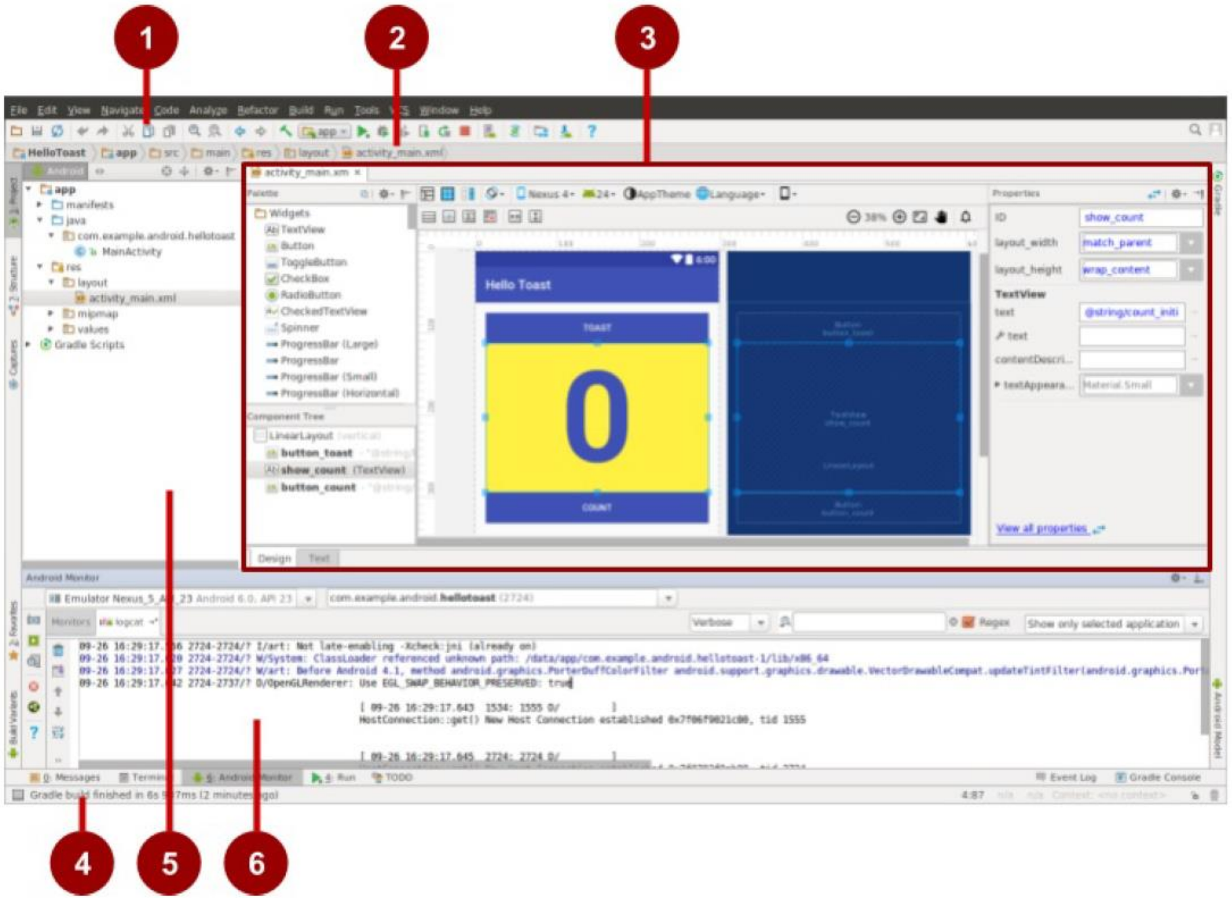
```java
DatabaseReference myRef = database.getReference("users/userId1");
myRef.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        String username = dataSnapshot.getValue(String.class);
        Log.d(TAG, "Username: " + username);
    }

    @Override
    public void onCancelled(DatabaseError error) {
        Log.w(TAG, "Failed to read value.", error.toException());
    }
});
```

5. Update or delete data in the database: To update or delete data in the Firebase database, you need to get a reference to the location of the data and use the updateChildren() or removeValue() method.

```java
DatabaseReference myRef = database.getReference("users/userId1");
myRef.updateChildren(Collections.singletonMap("name", "Mary"));
myRef.removeValue();
```

It's important to note that when working with Firebase database, you need to handle security rules to protect your data. Firebase provides a set of security rules that allow you to control who can read and write to your database. You should also consider using Firebase Authentication to authenticate users and restrict access to certain parts of the database.

ANDROID STUDIO ENVIRONMENT

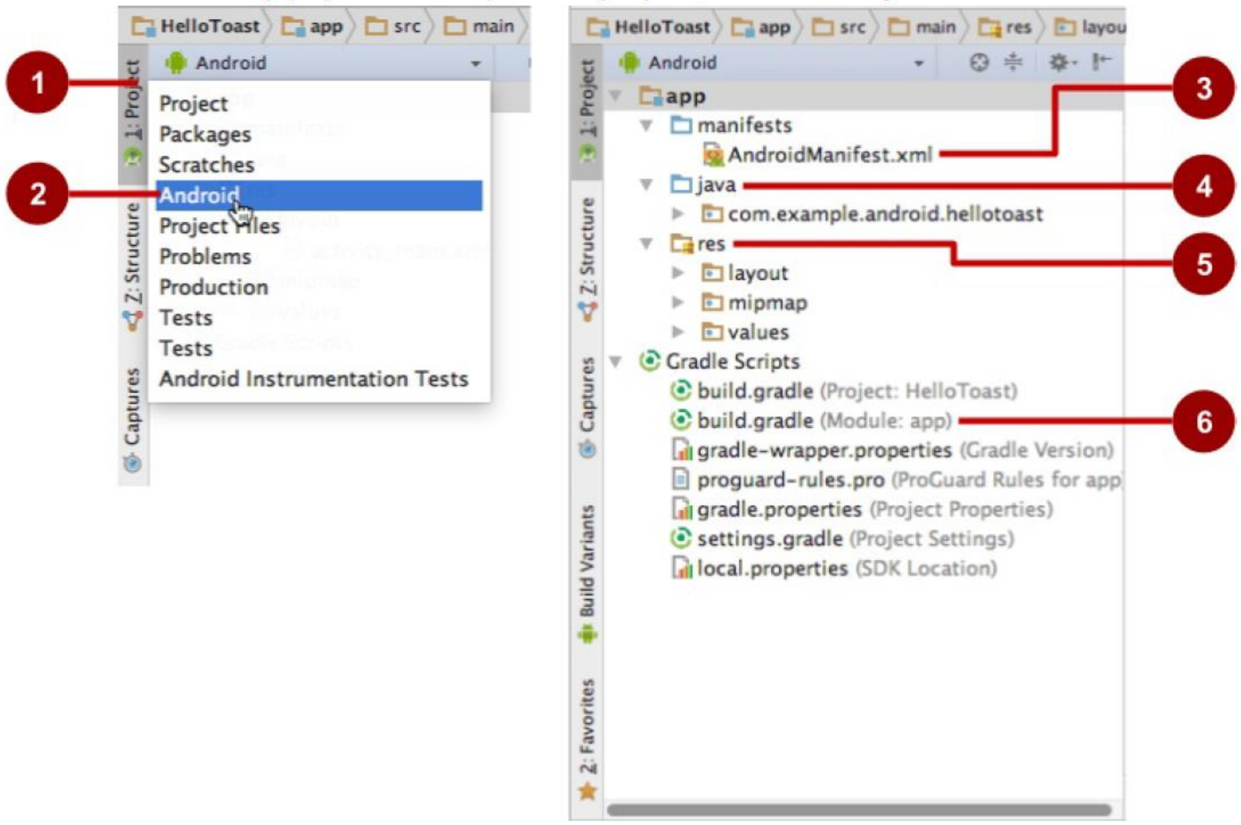

1
2
3
4
5
6
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloToast app src main res layout activity_main.xml
Palette
Component Tree
Properties
Hello Toast
TOAST
0
COUNT
Design Text
Android Monitor
0: Messages Terminal 6: Android Monitor 4: Run TODO Event Log Gradle Console

```
09-26 16:29:17.?56 2724-2724/? I/art: Not late-enabling -Xcheck:jni (already on)
09-26 16:29:17.?20 2724-2724/? W/System: ClassLoader referenced unknown path: /data/app/com.example.android.hellotoast-1/lib/x86_64
09-26 16:29:17.?27 2724-2724/? W/art: Before Android 4.1, method android.graphics.PorterDuffColorFilter android.support.graphics.drawable.VectorDrawableCompat.updateTintFilter(android.graphics.Por
09-26 16:29:17.?42 2724-2737/? D/OpenGLRenderer: Use EGL_SWAP_BEHAVIOR_PRESERVED: true

                        [ 09-26 16:29:17.643  1534: 1555 D/         ]
                        HostConnection::get() New Host Connection established 0x7f06f9021c80, tid 1555

                        [ 09-26 16:29:17.645  2724: 2724 D/         ]
```

Gradle build finished in 6s ?7ms (2 minutes ago)

In the above figure:
1. **The Toolbar**.The toolbar carries out a wide range of actions, including running the Android app and launching Android tools.
2. **The Navigation Bar**. The navigation bar allows navigation through the project and open files for editing. It provides a more compact view of the project structure.
3. **The Editor Pane**. This pane shows the contents of a selected file in the project. For example, after selecting a layout (as shown in the figure), this pane shows the layout editor with tools to edit the layout. After selecting a Java code file, this pane shows the code with tools for editing the code.
4. **The Status Bar**. The status bar displays the status of the project and Android Studio itself, as well as any warnings or messages. You can watch the build progress in the status bar.
5. **The Project Pane**. The project pane shows the project files and project hierarchy.

6. **The Monitor Pane**. The monitor pane offers access to the TODO list for managing tasks, the Android Monitor for monitoring app execution (shown in the figure), the logcat for viewing log messages, and the Terminal application for performing Terminal activities.


HelloToast
app
src
main
res
layou
Android
1: Project
Z: Structure
Captures
Build Variants
2: Favorites
Project
Packages
Scratches
Android
Project Files
Problems
Production
Tests
Tests
Android Instrumentation Tests
app
manifests
AndroidManifest.xml
java
com.example.android.hellotoast
res
layout
mipmap
values
Gradle Scripts
build.gradle (Project: HelloToast)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
1
2
3
4
5
6

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

The AndroidManifest.xml

The AndroidManifest.xml file exists in the root of an application directory and contains all the design-time relationships of a specific application and Intents. AndroidManfest.xml files act as deployment descriptors for Android applications. Example: see the following xml code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">
    <application android:icon="@drawable/icon">
       <activity android:name=".Activity1" 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>
```

From the above file, we can understand that , The android application

- contains 1 activity called ‘Activity1’
- has one Intent-filter, which makes Activity1 the main activity (action.MAIN) and puts it in launcher window (category.LAUNCHER)
- can have additional components in its files as follows:
    - <uses-permission> tag, used to allow certain permissions
    - <services> tag, to represent services
    - <reciever> tag, to represent broadcast reciever

✓ The Android Debug Bridge (adb) utility permits you to interact with the Android emulator directly from the command line or script. Have you ever wished you could navigate the filesystem on your smartphone? Now you can with adb! It works as a client/server TCP-based application.

REFERENCE BOOKS SUGGESTED FOR READING

1. **Android in Action Book** – 3rd Edition, by F. Ableson et al, 2012
2. **Head First Android Development** – By Dawn Griffiths and David Griffiths, 2017

REVIEW EXERCISES

1. Which of these was the first-ever phone released that was able to run the Android OS?

a. HTC Hero

b. Motorola Droid

c. T-Mobile G1

d. Google gPhone

Answer: (c) T-Mobile G1

2. Which of these does NOT refer to a nickname of the Android version?

a. Muffin

b. Honeycomb

c. Gingerbread

d. Cupcake

Answer: (a) Muffin

3. The .apk extension stands for which of these?

a. Application Program Kit

b. Android Package

c. Application Package

d. Android Proprietary Kit

Answer: (c) Application Package

4. Which of these is an XML file that consists of all the text used in an application?

a. string.java

b. string.xml

c. text.xml

d. stack.xml

Answer: (b) string.xml

5. Which of these does NOT refer to a state in a service's lifecycle?

a. Paused

b. Destroyed

c. Running

d. Starting

Answer: (a) Paused

6. How can we stop the services in android?

a. By using the stopSelf() and stopService() method
b. By using the finish() method
c. By using system.exit() method
d. None of the above

Answer (a)

7. How can we kill an activity in android?

a. Using finish() method
b. Using finishActivity(int requestCode)
c. Both (a) and (b)
d. Neither (a) nor (b)

Answer: (c) Both (a) and (b)

8. Which of the following is the first callback method that is invoked by the system during an activity life-cycle?

a. onClick() method
b. onCreate() method
c. onStart() method
d. onRestart() method

Answer: (b) onCreate() method

9. What is the use of content provider in Android?

a. For storing the data in the database

a. For sharing the data between applications
b. For sending the data from an application to another application
c. None of the above

Answer: (c)

10. Which of the following android component displays the part of an activity on screen?

a. View
b. Manifest

c. Intent
d. Fragment

Answer: (d)

11. Which of the layer is the lowest layer of android architecture?

a. System Libraries and Android Runtime
b. Linux Kernel
c. Applications
d. Applications Framework

Answer: (b)

12. What is contained in manifest.xml?

a. Source code
b. List of strings used in the app
c. Permission that the application requires
d. None of the above

Answer: (c)

13. In which state the activity is, if it is not in focus, but still visible on the screen?

a. Stopped state
b. Destroyed state
c. Paused state
d. Running state

Answer: (c)

14. In Android studio, which of the following callback is called when an activity starts interacting with the user?

a. onDestroy
b. onCreate
c. onResume

d. onStop

Answer: (c)

15. Which of the following class in android displays information for a short period of time and disappears after some time?

a. toast class
b. log class
c. maketest class
d. None of the above

Answer: (a)

16. All layout classes are the subclasses of -

a. android.view.View
b. android.view.ViewGroup
c. android.widget
d. None of the above

Answer: (b)

17. Which of the following layout in android aligns all children either vertically or horizontally?

a. RelativeLayout
b. TableLayout
c. FrameLayout
d. LinearLayout

Answer: (d)

18. What is the default layout type used by Android Studio ?

a) Linear Layout
b) Constraint Layout
c) Relative Layout
d) Card Layout

Answer (b)

19. Which of the following statements is **not** correct about the onCreate() function?

a. The onCreate method is not required to run an adroid app.
b. The onCreate method is called when your app is run.
c. The role of the onCreate() function is to create the view and initialize the actions of the activity.
d. The onCreate method is used to link your java file to the xml layout file

Answer (a)

20. Which of the following is known as Androids build system?

a. Gradle
b. AndroidManifest
c. AVD
d. SDK

Answer (a)