UNIT - II: ACTIVITIES, INTENTS, AND ANDROID USER INTERFACE

Topics Covered

- **Activities**
- **Linking Activities Using Intents**
- **Calling Built-in Applications Using Intents**
- **Displaying Notifications**
- **Components of a Screen**
- **Adapting to Display Orientation**
- **Managing Changes to Screen Orientation**
- **Utilizing the Action Bar**
- **Listening for UI Notifications**

Introduction to Activities in Android

In Android development, an **Activity** is essentially a screen where user interaction happens. Android applications can have multiple activities, but usually, each activity serves a specific function or displays a part of the user interface (UI). The main objective of an activity is to handle user interaction through different UI components.

An understanding of the **Activity Lifecycle** is crucial in Android development as it outlines the stages an activity goes through from creation to destruction. These stages allow developers to manage resources and ensure smooth transitions between activities. When the activity is created, it undergoes a series of life cycle events that dictate how it behaves as it interacts with the user.

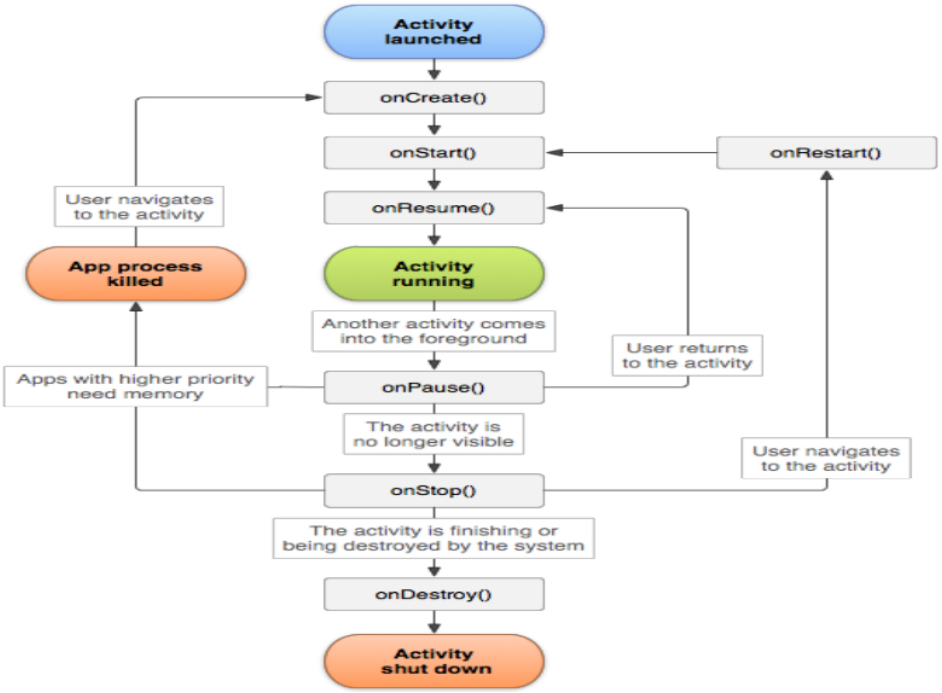
Activity Lifecycle Methods

1. **onCreate()**: This method is called when the activity is first created. It's used to initialize the activity, set up the UI, and load any essential data.
2. **onStart()**: Triggered when the activity becomes visible to the user but is not yet interactive.
3. **onResume()**: Called when the activity is in the foreground and is ready to interact with the user.
4. **onPause()**: This method is invoked when the activity is being partially obscured (e.g., by another activity) and gives developers a chance to pause tasks.
5. **onStop()**: Triggered when the activity is no longer visible. At this point, it's essential to release any resources that are not needed.
6. **onDestroy()**: This method is called when the activity is about to be destroyed, providing a chance to clean up any resources before the activity is completely removed.
7. **onRestart()**: This is called when an activity that was previously stopped is about to become active again.

MainActivity.kt

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding:ActivityMainBinding
    override fun onCreate(savedInstanceState: Bundle?) {
        binding=ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        super.onCreate(savedInstanceState)
    }
}
```



Note: When the Configuration Changes, the Activity will get Destroyed(i.e.onDestroy()) is called and Killed and Again the Activity Start form the Beginning (i.e. onCreate())

INTENTS IN ANDROID

In Android, an **Intent** acts as a mechanism for facilitating communication between different components of an app, such as activities, services, or broadcast receivers. Intents enable the transfer of data, allowing activities from different applications to work together seamlessly. They function as a "glue" between various app components and are used to start activities, services, or broadcast messages.

Linking Activities Using Intents

When an Android application contains multiple activities, navigation between them is achieved using **Intents**. Here's how different types of intents can be used to facilitate communication between activities:

1. Starting a New Activity/ Explicit Intent

To start a new activity from an existing one, an **Explicit Intent** is used.

```
val intent = Intent(this, TargetActivity::class.java)
startActivity(intent)
```

2. Passing Data Between Activities

We can pass data by using putExtra() with the intent.

```
val intent = Intent(this, TargetActivity::class.java)
intent.putExtra("key_name", "value")
startActivity(intent)
```

In the **TargetActivity**, retrieve the data like

```
val value = intent.getStringExtra("key_name")
```

3. Receiving Data Back

First, we need to register for an activity result launcher in your Activity

```
MainActivity.kt

class MainActivity : AppCompatActivity() {
// Register the ActivityResultLauncher
  private val startForResult =
registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
result: ActivityResult ->
      if (result.resultCode == Activity.RESULT_OK) {
          // Retrieve data from the intent
```

```
            val data = result.data
            val returnedData = data?.getStringExtra("result_key")
            // Handle the result (for example, display the returned data)
            println("Result received: $returnedData")
        }
    }
    // Start TargetActivity when a button is clicked or any other event
        val intent = Intent(this, TargetActivity::class.java)
        startForResult.launch(intent)
    }
}
```

TargetActivity.kt

```
class TargetActivity : AppCompatActivity() {

    // Create the result intent with data
    val resultIntent = Intent()
    resultIntent.putExtra("result_key", "This is the result from
TargetActivity")

    // Set the result and finish the activity
    setResult(Activity.RESULT_OK, resultIntent)
    finish()  // Close the activity and return to the calling activity
  }
}
```

Key points:

- **ActivityResultContracts.StartActivityForResult()**: Replaces the deprecated startActivityForResult().
- **Lambda-based result handling**: The result is processed directly in the callback, improving readability and reducing boilerplate code.

This method aligns with modern Android practices and is recommended for handling activity results as it avoids using onActivityResult() and simplifies activity result management.

4. Using Implicit Intents

Implicit Intents allow us to request an action without specifying which component will handle it.

For example, opening a webpage

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("http://www.example.com"))
startActivity(intent)
```

5.Sending a Broadcast Intent

To send a broadcast in Kotlin, you create an Intent with a specific action, then send it using sendBroadcast().

```
val intent = Intent("com.example.broadcast.MY_NOTIFICATION")
intent.putExtra("data_key", "data_value")
sendBroadcast(intent)
```

Receiving a Broadcast Intent

To receive a broadcast, we need to create a BroadcastReceiver. This receiver listens for specific broadcast intents, reacts to them, and processes any data that is attached to the broadcast.

```
class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val data = intent.getStringExtra("data_key")
        // Handle the broadcast (e.g., log the data or update the UI)
        Log.d("BroadcastReceiver", "Received data: $data")
    }
}
```

Registering the Receiver:

There are two ways to register a BroadcastReceiver:

1. **In the Manifest (for global broadcasts or when using static broadcasts)**:

    - Add the receiver to your AndroidManifest.xml file

```
Manifest.xml

<receiver android:name=".MyBroadcastReceiver">
<intent-filter>
<action
android:name="com.example.broadcast.MY_NOTIFICATION"/>
</intent-filter>
</receiver>
```

2. **Dynamically at Runtime (for local broadcasts)**:

    - In the activity or service, register the receiver dynamically:

```
val intentFilter = IntentFilter("com.example.broadcast.MY_NOTIFICATION")
val receiver = MyBroadcastReceiver()
registerReceiver(receiver, intentFilter)
```

Note: We should unregister the receiver when no longer needed, for example, in the onDestroy() method.

```
override fun onDestroy() {
   super.onDestroy()
   unregisterReceiver(receiver)
}
```

Calling Built-In Applications Using Intents in Android

One of the key features of Android programming is the ability to interact with built-in applications using **intents**. This allows your application to leverage existing functionalities, such as maps, phone calls, contact management, and web browsing, without the need to implement these features from scratch. For example, if your application requires displaying a web page, you can use an **Intent** to invoke the built-in web browser instead of creating your own.

Components of an Intent

1. **Action**: Specifies the task to be performed, such as viewing or editing an item.
2. **Data**: Defines the target of the action, often represented as a Uniform Resource Identifier (**URI**).

The combination of an **action** and **data** describes the operation to be executed.

Common Actions and Data Examples

Below are some examples of commonly used actions and their corresponding data:

Action	Data Example	Purpose
ACTION_VIEW	geo:37.827500,-122.481670	Open a location on a map.
ACTION_DIAL	tel: +91 6598521456	Open the phone dialer with a phone number.
ACTION_PICK	content://contacts	Pick a contact from the contact list.
ACTION_VIEW	www.google.com	Display a webpage in a browser.

Examples of Using Intents in Android

Below are examples of how to use intents to call built-in applications

1. Dialing a Phone Number

To open the phone dialer with a pre-filled phone number

```
val dialIntent = Intent(Intent.ACTION_DIAL).apply {
    data = Uri.parse("tel:+1234567890")
}
startActivity(dialIntent)
```

2. Opening a Web Page

To launch the web browser and open a specified webpage

```
val webIntent = Intent(Intent.ACTION_VIEW).apply {
    data = Uri.parse("https://mbu.com")
}
startActivity(webIntent)
```

3. Displaying a List of Contacts

To display the contacts stored on the device

```
val contactsIntent = Intent(Intent.ACTION_VIEW).apply {
    data = Uri.parse("content://contacts/people")
}
startActivity(contactsIntent)
```

4. Selecting a Contact

To allow the user to pick a contact from their contact list

```
val pickContactIntent = Intent(Intent.ACTION_PICK).apply {
    data = Uri.parse("content://contacts")
}
pickContactLauncher.launch(pickContactIntent)
```

Note: For using this method, we have to Register the Activity Result Launcher

5. Opening a Location in Maps

To open a specific location in a map application using coordinates

```
val locationIntent = Intent(Intent.ACTION_VIEW).apply {
    data = Uri.parse("geo:37.7749,-122.4194")
}
startActivity(locationIntent)
```

Displaying Notifications in Android

Notifications are a key feature in Android that allow applications to send persistent messages to the user. Unlike Toast messages, which are temporary and disappear after a few seconds, notifications remain in the status bar (also called the notification bar) until the user interacts with them or clears them. Notifications are typically used to provide important or time-sensitive information.

For displaying the notification there are four important concepts that we have to know.

1. NOTIFICATION CHANNEL

2. NOTIFICATION BUILDER

3. NOTIFICATION MANAGER

4. PENDING INTENT

Steps to Display a Notification

1. Creating an Intent

An Intent is used to specify the target activity that will launch when the user clicks on the notification. For example, this could navigate the user to a specific screen in your application.

```
val intent = Intent(this, NotificationViewActivity::class.java)
```

2. Creating a PendingIntent

Since notifications might need to interact with the app even if it's not running, you need a PendingIntent. A PendingIntent acts as a wrapper for the Intent and allows it to be executed at a later time on behalf of your application. Use the getActivity() method to create a PendingIntent for launching an activity

```
val pendingIntent = PendingIntent.getActivity(this, 0, intent,
    PendingIntent.FLAG_IMMUTABLE)
```

3. Building the Notification

To create the notification, use the **NotificationCompat.Builder** class, which provides an easy way to construct notifications.

```
val notification = NotificationCompat.Builder(this,
    NOTIFICATION_CHANNEL_ID)
    .setContentTitle("MBU Id Registered Successfully")
    .setContentText("Click here to explore")
    .setSmallIcon(R.drawable.notification_foreground)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
```

```
    .setCategory(NotificationCompat.CATEGORY_MESSAGE)
    .setContentIntent(pendingIntent)
    .setAutoCancel(true)
    .build()
```

4. Displaying Notification with Notification Manager

Obtain an instance of the NotificationManager class and use the notify() method to display the notification.

```
val notificationManager = NotificationManagerCompat.from(this)
    val button = findViewById<Button>(R.id.button)
    button.setOnClickListener {
        notificationManager.notify(NOTIFICATION_ID, notification)
    }

val manager = getSystemService(Context.NOTIFICATION_SERVICE) as
        NotificationManager
manager.createNotificationChannel(channel)
```

5. Configuring Vibration or Sound

We can add vibration or sound to the notification using the setVibrate() and setSound() methods

```
val vibrationPattern = longArrayOf(0, 500, 1000)
notification.vibrate = vibrationPattern
```

Key Points

1. **PendingIntent**: Allows the notification to perform an action later, even if the app is not active.
2. **NotificationCompat.Builder**: Simplifies the creation of notifications.
3. **Notification Channels**: Mandatory for devices running Android 8.0 (API level 26) and above.
4. **Vibration and Sound**: You can enhance notifications by adding vibration patterns or custom sounds.

Understanding the Components of a Screen in Android

The fundamental unit of an Android application is an **Activity**. It serves as the entry point for interacting with the user and typically displays the user interface (UI). The UI consists of various components like widgets (buttons, text boxes, labels, etc.) organized within layouts.

Views and ViewGroups

An **Activity** manages UI components through **Views** and **ViewGroups**:

- **View**: Represents an individual UI element that has a visible appearance on the screen. Examples include buttons, text views, and image views.
- **ViewGroup**: Acts as a container for multiple views and organizes them into a specific layout. Examples of ViewGroups are LinearLayout and RelativeLayout.

Common ViewGroups in Android

1. **LinearLayout**

- Organizes child views in either horizontally or vertically.
- Controlled using the orientation attribute.

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

2. **RelativeLayout**

- Positions child views relative to one another or the parent container.
- Examples: Positioning view A to the right of view B.

```
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
</RelativeLayout>
```

3. **ConstraintLayout**

- Similar to RelativeLayout but more powerful with constraints.
- Allows complex layouts without nesting.

```
<androidx.constraintlayout.widget.ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent">
</androidx.constraintlayout.widget.ConstraintLayout>
```

4. **FrameLayout**

- Stacks child views on top of each other, displaying only one view at a time.

```
<FrameLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
</FrameLayout>
```

5. **TableLayout**

- Organizes child views into rows and columns.

```
<TableLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <!-- Rows and Cells -->
</TableLayout>
```

Common Attributes for Views and ViewGroups

Attribute	**Description**
layout_width	Defines the width of a View or ViewGroup.
layout_height	Defines the height of a View or ViewGroup.
layout_marginTop	Adds extra space above the View or ViewGroup.
layout_marginBottom	Adds extra space below the View or ViewGroup.
layout_marginLeft	Adds extra space to the left of the View or ViewGroup.
layout_marginRight	Adds extra space to the right of the View or ViewGroup.
layout_gravity	Specifies how child views are aligned within the parent ViewGroup.
layout_weight	Determines the proportion of extra space a View takes in a layout.
layout_x	Specifies the X-coordinate of the View or ViewGroup.
layout_y	Specifies the Y-coordinate of the View or ViewGroup.

Screen Densities in Android

Android devices vary in screen size and resolution, which impacts pixel density (measured in dots per inch or DPI). The platform classifies screen densities into four categories.

Density	Description	DPI
Low density	Small screens	120 dpi
Medium density	Standard screens	160 dpi
High density	Larger screens	240 dpi
Extra high density	High-resolution screens	320 dpi

Using **density-independent pixels (dp)** ensures consistent UI dimensions across different screen densities.

Units of Measurement in Android

1. **dp (Density-independent Pixel)**
   - A virtual pixel unit recommended for defining UI dimensions.
   - Scales automatically across different screen densities.
2. **sp (Scale-independent Pixel)**
   - Similar to dp but scales based on the user's preferred text size.
   - Recommended for defining font sizes.

3. **pt (Point)**
   - Based on physical screen size, where 1 pt equals 1/72 inch.
4. **px (Pixel)**
   - Corresponds to actual screen pixels.
   - Avoid using as it may result in inconsistent layouts on different devices.

Adapting to Display Orientation in Android

Modern smartphones provide users with the flexibility to switch screen orientation between **portrait** and **landscape** modes. Android natively supports this functionality, and developers need to ensure their applications adapt seamlessly to these changes for an optimal user experience.

Default Behaviour of Orientation Changes

By default, when the device orientation changes:

- The activity is destroyed and recreated.
- The system redraws the layout to fit the new screen dimensions. While this ensures that the UI adjusts to the new orientation, it may lead to performance overhead and loss of temporary data if not handled properly.

Techniques for Handling Orientation Changes

1. **Anchoring Views to Screen Edges**
   - This is the simplest way to handle orientation changes.
   - Views are aligned to the four edges of the screen using layout attributes such as

android:layout_alignParentLeft,android:layout_alignParentRight, android:layout_alignParentTop, android:layout_alignParentBottom.

Advantages

- Minimal effort is required for implementation.
- The layout adapts naturally to the new orientation.

Use Cases

- Static layouts where the placement of elements does not require complex adjustments.

2. **Resizing and Repositioning Views**

➢ Resizing and repositioning is a more advanced technique that adjusts the size and position of every view dynamically based on the orientation.

➢ **Steps to Implement**

- Create separate layout files for portrait and landscape orientations.
- Use the res/layout folder for portrait layouts and res/layout-land folder for landscape layouts.
- Each layout file can define the size, position, and alignment of views differently based on the orientation.

Advantages

- Provides maximum control over the layout.
- Allows the use of additional screen space in landscape mode.

Use Cases

- Applications where layout changes significantly in landscape mode, such as media players, gaming apps, or apps requiring multi-column designs.

Handling Orientation Without Activity Recreation

By default, the system destroys and recreates the activity during an orientation change. To avoid this, developers can:

1. Use the android:configChanges attribute in the AndroidManifest.xml

```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize" />
```

This prevents the activity from being recreated. Instead, the onConfigurationChanged() method is called, where you can manually handle the changes.

2. Override onConfigurationChanged()

```
override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE)
{
        // Handle landscape-specific changes
    } else if (newConfig.orientation ==
Configuration.ORIENTATION_PORTRAIT) {
        // Handle portrait-specific changes
    }
}
```

Utilizing the Action Bar in Android

The **Action Bar** serves as a key component in Android apps, providing a consistent user interface for performing various operations. It enhances user experience by integrating branding, navigation, and frequently used actions in a central location.

Key Features of the Action Bar

1. **Navigation:** The Action Bar can include navigation options such as **tabs** or **drop-down menus**, allowing users to move between different sections of the app easily.
2. **Branding:** The Action Bar provides a space where you can place the app's **logo** or **title**, helping reinforce the app's brand identity and making it easily recognizable to users.
3. **User Actions:** It provides quick access to essential **user actions** such as **search** or **settings**, allowing users to perform these tasks quickly without navigating through the app.
4. **Overflow Menu:** When there are more actions than can fit on the Action Bar, they are placed in an **overflow menu**, accessible by tapping the three-dot icon (⋮), which helps keep the interface organized and free from clutter.

Customizing the Action Bar

The Action Bar can be customized programmatically to suit the app's requirements.

```
// Get a reference to the Action Bar
val actionBar = supportActionBar

if (actionBar != null) {
    // Set a custom title
    actionBar.title = "My Custom Title"
```

```
    // Enable the 'Up' button for navigation
    actionBar.setDisplayHomeAsUpEnabled(true)

    // Optionally, set a custom logo
    actionBar.setLogo(R.drawable.ic_logo)
    actionBar.setDisplayUseLogoEnabled(true)
}
```

Key Points

- title: Sets a custom title for the Action Bar.
- setDisplayHomeAsUpEnabled(true): Adds a back button for easier navigation.
- setLogo(): Sets a logo to be displayed alongside the title.
- setDisplayUseLogoEnabled(true): Ensures the logo is shown.

Listening for UI Notifications

In Android, we can listen for various UI notifications to respond to user interactions. These notifications are typically triggered by user actions such as clicks, touches, or changes in UI elements. Below are some examples demonstrating how to handle these UI notifications.

Example 1: Handling a Button Click

To handle a button click, an OnClickListener is used to detect the event when a user taps the button.

```
val myButton = findViewById<Button>(R.id.myButton)
myButton.setOnClickListener {
    Toast.makeText(
        applicationContext,
        "Button clicked!",
```

```
        Toast.LENGTH_SHORT
    ).show()
}
```

Example 2: Listening for Text Input Changes

Android also allows us to listen for changes in text input. The TextWatcher interface provides methods to track text changes in an **EditText** field.

```
val editText = findViewById<EditText>(R.id.editText)
editText.addTextChangedListener(object : TextWatcher {
    override fun onTextChanged(s: CharSequence, start: Int, before: Int, count:
Int) {

    }
    override fun beforeTextChanged(s: CharSequence, start: Int, count: Int,
after: Int) {

    }
    override fun afterTextChanged(s: Editable) {

    }
})
```

UNIT III: ADVANCED USER INTERFACE AND DATA PERSISTENCE

Introduction

This unit focuses on creating enriched user interfaces and managing data storage effectively in Android applications. It emphasizes the use of **Basic Views** (e.g., TextView, EditText, Button) and interactive elements like **Picker Views** (DatePicker, TimePicker) and **ListView** for displaying dynamic data. It also introduces **ImageView** for media content and **WebView** for integrating web content within the app. On the data persistence side, it covers saving user settings with **SharedPreferences**, persisting data to files, and managing databases for long-term data storage. The aim is to ensure intuitive user interfaces and reliable data handling techniques for seamless app functionality.

Basic Views in Android

Basic Views are fundamental components for building Android user interfaces. They enable interaction and display content effectively. Below are some essential views with examples:

TextView in Android

TextView is used to display static text. It is often paired with other views like EditText or CheckBox for labelling purposes in forms or layouts.

```
<TextView
  android:id="@+id/textView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello, World!"
  android:textSize="18sp"/>
```