Unit 09

Broadcast Receivers and Intents

Structure:

9.1 Introduction

Intents are used as a message-passing mechanism that works both within your application and between applications. You can use Intents to do the following:

- Explicitly start a particular Service or Activity using its class name
- Start an Activity or Service to perform an action with (or on) a

particular piece of data.

- Broadcast that an event has occurred.

You can use Intents to support interaction among any of the application components installed on an Android device, no matter which application they're a part of. This turns your device from a platform containing a collection of independent components into a single, interconnected system.

One of the most common uses for Intents is to start new Activities, either explicitly (by specifying the class to load) or implicitly (by requesting that an action be performed on a piece of data). In the
In the latter case, the action does not need to be performed by an Activity within the calling application.

Objectives:

1. Define the concept of intent in Android and categorise their types (explicit and implicit).
2. Explain the mechanism of passing data between Activities using Intents and startActivityForResult().
3. Develop an Android app that utilises explicit and implicit Intents to navigate between different Activities.
4. Identify and classify commonly used native Android actions.
5. Customize Linkify to recognise specific patterns and link them to appropriate actions or URLs.
6. Create an Android app that implements Broadcast Receivers to listen for system events such as network connectivity changes, battery status updates, or custom broadcasts triggered within the app.

9.2 Utilising Intents for Component Communication

9.2.1 Using Intents to Launch Activities

The most common use of Intents is to bind your application components and communicate between them. Intents are used to start Activities, allowing you to create a workflow of different screens.

To create and display an Activity, call startActivity, passing in an Intent, as follows: startActivity(myIntent);

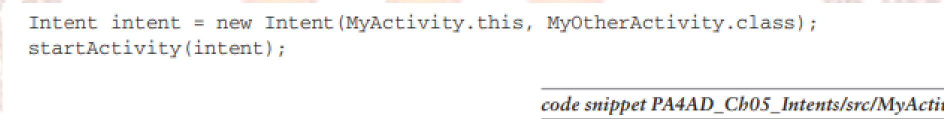
The startActivity method finds and starts the single Activity that best matches your Intent.

You can construct the Intent to explicitly specify the Activity class to open or to include an action that the target Activity must be able to perform.

In the latter case, the run time will choose an Activity dynamically using a process known as intent resolution. When you use startActivity, your application won't receive any notification when the newly launched Activity finishes. To track feedback from a sub-Activity, use startActivityForResult.

Explicitly Starting New Activities:

To select a specific Activity class to start, create a new Intent, specifying the current Activity's Context and the class of the Activity to launch. Pass this Intent into startActivity, as shown below:



```
Intent intent = new Intent(MyActivity.this, MyOtherActivity.class);
startActivity(intent);
```

code snippet PA4AD_Ch05_Intents/src/MyActivity.java

After startActivity is called, the new Activity (in this example, MyOtherActivity) will be created, started, and resumed — moving to the top of the Activity stack.

Calling finish on the new Activity or pressing the hardware back button closes it and removes it from the stack. Alternatively, you can continue to navigate to other Activities by calling startActivity. Note that each time you call startActivity, a new Activity will be added to the stack; pressing back (or calling finish) will remove each of these activities, in turn.

Implicit Intents:

Implicit Intents allow you to perform actions without specifying a particular component. They are helpful when you want to delegate a task to another app or component that can handle that task without knowing the specific implementation.

Example - Sharing Content:

Consider a scenario where you want to allow users to share content from your app. You might use an implicit intent to trigger the system's share functionality:

```
Intent shareIntent = new Intent(Intent.ACTION_SEND);
shareIntent.setType("text/plain");
shareIntent.putExtra(Intent.EXTRA_TEXT, "Check out this amazing
article!");
startActivity(Intent.createChooser(shareIntent, "Share via"));
```

In this example:

Intent.ACTION_SEND represents the action of sending something.

setType("text/plain") specifies the MIME type of the content you're sharing (text in this case).

putExtra(Intent.EXTRA_TEXT, "Check out this amazing article!") adds the actual text content to be shared.

startActivity(Intent.createChooser(shareIntent, "Share via")) initiates the sharing action and presents the user with a list of apps that can handle the sharing action.

Late Runtime Binding:
Late Runtime Binding allows the system to resolve the appropriate component dynamically at runtime. This happens when you want to perform an action, but you're not sure which specific component or app will handle it until runtime.

Example - Opening a File:
Suppose your app needs to open a file, but you want the user to choose how to handle it:

```
Intent openFileIntent = new Intent(Intent.ACTION_VIEW);
openFileIntent.setDataAndType(Uri.parse("file://path/to/your/document"),
"application/pdf");

if (openFileIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(openFileIntent);
} else {
    // Handle the case where no activity can handle the intent
```

In this case:

Intent.ACTION_VIEW indicates the action of viewing something.

setDataAndType(Uri.parse("file://path/to/your/document"), "application/pdf") sets the data (file path) and the type of file (PDF in this case).

resolveActivity(getPackageManager()) checks if there's an activity that can handle viewing a PDF file. If found, startActivity(openFileIntent) launches that activity; otherwise, you can handle the absence of a suitable app.

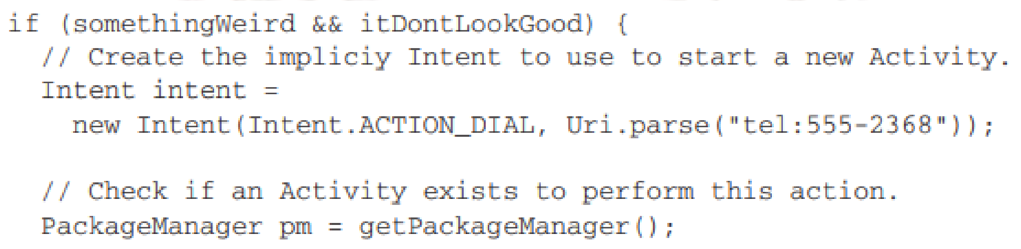
Advantages:

- Flexibility: Implicit Intents allow the system to find the best component dynamically to handle a given action.
- Decoupling: They facilitate loose coupling between components by not specifying a particular target explicitly, promoting modularity.

Determining If an Intent Will Resolve:

Incorporating the Activities and Services of a third-party application into your own is incredibly powerful; however, there is no guarantee that any particular application will be installed on a device, or that any application capable of handling your request is available.

As a result, it's good practice to determine if your call will resolve to an Activity before calling startActivity. You can query the Package Manager to determine which, if any, Activity will be launched to service a specific Intent by calling resolveActivity on your Intent object, passing in the Package Manager, as shown.



```
if (somethingWeird && itDontLookGood) {
  // Create the impliciy Intent to use to start a new Activity.
  Intent intent =
    new Intent(Intent.ACTION_DIAL, Uri.parse("tel:555-2368"));

  // Check if an Activity exists to perform this action.
  PackageManager pm = getPackageManager();
  ComponentName cn = intent.resolveActivity(pm);
  if (cn == null) {
    // If there is no Activity available to perform the action
    // Check to see if the Google Play Store is available.
    Uri marketUri =
      Uri.parse("market://search?q=pname:com.myapp.packagename");
    Intent marketIntent = new
      Intent(Intent.ACTION_VIEW).setData(marketUri);

    // If the Google Play Store is available, use it to download an application
    // capable of performing the required action. Otherwise log an
    // error.
    if (marketIntent.resolveActivity(pm) != null)
      startActivity(marketIntent);
    else
      Log.d(TAG, "Market client not available.");
  }
  else
    startActivity(intent);
}
```

9.2.2 Returning Results from Activities:

Launching Sub-Activities:
When you launch a sub-activity with the intention of receiving a result back, you use startActivityForResult() instead of startActivity(). This method allows the sub-activity to perform a task and return data to the calling activity.

Example - Launching a Sub-Activity:
Suppose you have a button click event in your main activity that triggers the start of a sub-activity to get some user input:

```
// In MainActivity.java
static final int REQUEST_CODE_SUB_ACTIVITY = 1;

// Method to start SubActivity
public void startSubActivity() {
    Intent intent = new Intent(this, SubActivity.class);
    startActivityForResult(intent, REQUEST_CODE_SUB_ACTIVITY);
}
```

Returning Results:
In the sub-activity, once the task is complete, you set the result that you want to send back to the calling activity using setResult() and then finish the sub-activity.

Example - Returning Results from SubActivity:

```
// In SubActivity.java
public void taskCompletedAndReturnResult() {
    Intent resultIntent = new Intent();
    resultIntent.putExtra("key_result", "Some result data");
    setResult(Activity.RESULT_OK, resultIntent);
    finish();
}
```

Handling Sub-Activity Results:
Back in the calling activity (MainActivity), you override the onActivityResult() method to receive the result sent by the sub-activity.

Example - Handling Result in MainActivity:

```
// In MainActivity.java
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == REQUEST_CODE_SUB_ACTIVITY) {
        if (resultCode == Activity.RESULT_OK) {
            if (data != null) {
                String result = data.getStringExtra("key_result");
                // Handle the result received from SubActivity
            }
        } else if (resultCode == Activity.RESULT_CANCELED) {
            // Handle the case where the user cancelled the operation in SubActivity
        }
    }
}
```

Explanation:

- startActivityForResult() is used to start a sub-activity and expects a result back, identified by a request code.

- In the sub-activity, setResult() is used to set the result code (e.g., RESULT_OK for success) and attach any necessary data to be sent back.

- onActivityResult() in the calling activity receives the result from the sub-activity. You can check the request code to determine which sub-activity is sending the result and handle it accordingly.

This mechanism provides a structured and organised way for activities to work together, facilitating communication and data exchange between different parts of your app.

Advantages:

- Interactivity between Activities: Enables activities to communicate and share data seamlessly.
- Controlled Flow: Allows the parent activity to control the flow based on the result received from the sub-activity.

9.2.3 Native Android Actions:

Native Android applications also use Intents to launch Activities and Sub-Activities.

The following (noncomprehensive) list shows some of the native actions available as static string constants in the Intent class. When creating implicit Intents, you can use these actions, known as Activity Intents, to start Activities and sub-activities within your own applications.

- ACTION_ALL_APPS — Opens an Activity that lists all the installed applications. Typically, this is handled by the launcher.

- ACTION_ANSWER — Opens an Activity that handles incoming calls. This is normally handled by the native in-call screen.

- ACTION_BUG_REPORT — Displays an Activity that can report a bug. This is normally handled by the native bug-reporting mechanism.

- ACTION_CALL — Brings up a phone dialer and immediately initiates a call using the number supplied in the Intent's data URI. This action should be used only for Activities that replace the native dialer application. In most situations, it is considered a better form to use ACTION_DIAL.

- ACTION_CALL_BUTTON — Triggered when the user presses a hardware "call button." This typically initiates the dialer Activity.

- ACTION_DELETE — Starts an Activity that lets you delete the data specified at the Intent's data URI.

- ACTION_DIAL — Brings up a dialer application with the number to dial prepopulated from the Intent's data URI. By default, this is handled by the native Android phone dialer. The dialer can normalise most number schemas — for example, tel:555-1234 and tel:(212) 555 1212 are both valid numbers.

- ACTION_EDIT — Requests an Activity that can edit the data at the Intent's data URI.

- ACTION_INSERT — Opens an Activity capable of inserting new items into the Cursor specified in the Intent's data URI. When called a sub-activity, it should return a URI to the newly inserted item.

- ACTION_PICK — Launches a sub-activity that lets you pick an item from the Content Provider specified by the Intent's data URI. When closed, it should return a URI to the item that was picked. The Activity launched depends on the data being picked — for example, passing content://contacts/people will invoke the native contacts list.

- ACTION_SEARCH — Typically used to launch a specific search Activity. When fired without a specific Activity, the user will be prompted to select from all applications that support search. Supply the search term as a string in the Intent's extras using SearchManager.QUERY is the key.

- ACTION_SEARCH_LONG_PRESS — Enables you to intercept long presses on the hardware search key. The system typically handles this to provide a shortcut to a voice search.

9.2.4 Introducing Linkify

Linkify is a utility class in Android that helps in converting textual information into clickable links. It's a convenient way to detect specific patterns within a TextView and automatically create clickable links for those patterns.

How Linkify Works:
You can use Linkify to identify patterns in a TextView and automatically convert them into clickable links. It supports various types of patterns like URLs, email addresses, phone numbers, and custom patterns defined using

regular expressions.

Basic Usage:
Here's an example demonstrating how to use Linkify to make URLs clickable within a TextView:

Example:

```
TextView textView = findViewById(R.id.textView);

String textWithLink = "Visit our website at https://www.example.com";
textView.setText(textWithLink);

Linkify.addLinks(textView, Linkify.WEB_URLS);
```

In this example:

- textView is a TextView in your layout.
- textWithLink contains the text where you want to make the URL clickable.
- Linkify.addLinks() is used to identify web URLs (Linkify.WEB_URLS) within the text and convert them into clickable links.

Customizing Linkify:
You can also customise Linkify to match and link other patterns like email addresses, phone numbers, or custom patterns using regular expressions.

Example - Linkifying Email Addresses:

```
TextView textView = findViewById(R.id.textView);

String textWithEmail = "Contact us at info@example.com";
textView.setText(textWithEmail);

Pattern pattern = Patterns.EMAIL_ADDRESS;
Linkify.addLinks(textView, pattern, "mailto:");
```

Here:

- Patterns.EMAIL_ADDRESS is a predefined pattern in Android for email addresses.

- Linkify.addLinks() with the pattern Patterns.EMAIL_ADDRESS links all email addresses in the text and adds a "mailto:" prefix, making them clickable for email composition.

Custom Linkify for Custom Patterns:
You can define custom patterns using regular expressions and apply Linkify accordingly.

Example - Custom Pattern:

```
TextView textView = findViewById(R.id.textView);

String textWithCustomPattern = "Tracking number: ABC123XYZ";
textView.setText(textWithCustomPattern);

Pattern customPattern = Pattern.compile("\\b[A-Z]{3}\\d{3}[A-Z]{3}\\b");
Linkify.addLinks(textView, customPattern, "https://tracking.example.com/");
```

Here:

- Pattern.compile("\\b[A-Z]{3}\\d{3}[A-Z]{3}\\b") defines a custom pattern using a regular expression to match strings like "ABC123XYZ".

- Linkify.addLinks() with this custom pattern converts the matching text into clickable links, appending the given URL prefix.

Advantages:
- Convenience: Makes specific patterns within TextViews clickable without the need for manually handling onClick events.
- Enhanced User Experience: This provides users with familiar behaviour where they can interact with recognised patterns directly.
- Linkify simplifies the process of detecting and creating clickable links within text, adding interactivity to your app's UI effortlessly.

9.2.5 Broadcasting events using Broadcast Intents

Broadcasting Events:
Broadcast Intents allow you to send messages or notifications throughout the system. This mechanism enables various parts of your app or even different apps to react to specific events or actions.

Example - Sending a Broadcast Intent:

```
// Create an Intent with a custom action
Intent broadcastIntent = new Intent("com.example.ACTION_CUSTOM_EVENT");

// Add extra data if needed
broadcastIntent.putExtra("key", "value");

// Send the broadcast
sendBroadcast(broadcastIntent);
```

In this example:

- An Intent is created with a custom action ("com.example.ACTION_CUSTOM_EVENT").
- Additional data can be added using putExtra() to provide more information.
- sendBroadcast() delivers the intent, notifying any registered components that are listening for this particular action.

Receiving Broadcasts:
To receive a broadcast, you need to register a BroadcastReceiver either statically in your manifest file or dynamically in your app's code.

Example - Registering a BroadcastReceiver in code:

```
// Create a BroadcastReceiver to handle the broadcast
BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Handle the received broadcast here
        if ("com.example.ACTION_CUSTOM_EVENT".equals(intent.getAction())) {
            String value = intent.getStringExtra("key");
            // Perform actions based on the received data
```

Lifecycle Considerations:
Remember to unregister the BroadcastReceiver when it's no longer needed to prevent memory leaks. For instance, unregister it in onDestroy() or onStop(), depending on your app's requirements.

Example - Unregistering the BroadcastReceiver:

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // Unregister the BroadcastReceiver when no longer needed
    unregisterReceiver(receiver);
}
```

Broadcast Types:
There are two types of broadcasts:

Normal broadcasts: Receivers get them asynchronously, making them more efficient but less secure.
Ordered broadcasts: Receivers get them sequentially, allowing for control of the order of receivers and the ability to modify data.

Advantages:
Loose Coupling: Enables different parts of an app to communicate without tight dependencies.
Inter-App Communication: Allows communication between different apps that listen for specific broadcast events.

9.2.6 Using Pending Intents

PendingIntents in Android are a powerful tool used to grant other applications permission to perform operations on behalf of your application.

They encapsulate an Intent and grant a foreign application the ability to perform the operation you specify in the Intent as if it were your application. PendingIntents are often used in scenarios like notifications, alarms, or when deferring an action for later execution.

Creating PendingIntents:
You can create a PendingIntent using the PendingIntent.getBroadcast(),

PendingIntent.getActivity(), PendingIntent.getService(), or PendingIntent.getForegroundService() methods, depending on whether you want to start a broadcast, activity, or service.

Example - Creating a PendingIntent for an Activity:

```
Intent intent = new Intent(this, YourActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_UPDATE_CURRENT);
```

Using PendingIntents:
Once you have a PendingIntent, you can use it in various contexts like notifications, alarms, or deferred actions.

Example - Using PendingIntent in a Notification:

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this,
"channel_id")
        .setContentIntent(pendingIntent)
        .setSmallIcon(R.drawable.ic_notification)
        .setContentTitle("Notification Title")
        .setContentText("Notification Content");

NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(this);
notificationManager.notify(notificationId, builder.build());
```

Understanding PendingIntents:

- Target Component: The Intent encapsulated by the PendingIntent defines the component (Activity, Service, BroadcastReceiver) and action to be performed.
- Security and Permissions: PendingIntents allow other apps to execute operations on your behalf, so it's essential to ensure that only trusted entities can use them.
- Flags: Flags like FLAG_UPDATE_CURRENT in the creation of PendingIntents define how the system should handle them. For instance, FLAG_UPDATE_CURRENT updates the existing PendingIntent if it already exists with the same data.

- Deferred Execution: PendingIntents allow you to defer an action until a later time or allow another app to execute an action on your behalf.

Use Cases:

- Notifications: PendingIntent allows you to define actions when a user interacts with a notification.
- Alarms: You can set up PendingIntent for scheduled actions, like triggering an alarm at a specific time.
- RemoteViews: Used for updating App Widgets, allowing you to define actions within the widget.

Advantages:

- Delegated Operations: Allows other applications to perform predefined actions on your behalf.
- Flexibility: Supports a variety of use cases, from notifications to deferred actions and alarms.

Self-Assessment questions

1. What is the purpose of startActivityForResult() in Android?
   A) It starts a new Activity without expecting any result.
   B) It starts a sub-activity and waits for a result from it.
   C) It launches an implicit Intent to perform an action.
   D) It triggers a broadcast event throughout the system.

`2. Which method is used to identify and convert specific patterns within a TextView into clickable links using Linkify?
   A) setClickable()
   B) convertToLinks()
   C) Linkify.addLinks()
   D) createClickableLinks()

3. What is the primary purpose of Broadcasting Intents in Android?

A) To start Activities dynamically
B) To communicate between different parts of an app
C) To schedule deferred actions
D) To handle UI interactions

4. Which type of broadcast allows receivers to get messages sequentially, controlling the order of receivers and enabling data modification?

A) Ordered broadcasts

B) Normal broadcasts
C) Sequential broadcasts
D) SequentialOrdered broadcasts

5. Which method is used to create a PendingIntent for an Activity in Android?

A) PendingIntent.getActivity()
B) PendingIntent.createActivity()
C) PendingIntent.startActivity()
D) PendingIntent.launchActivity()

6. Linkify is used to make ___________ within a TextView clickable.

7. startActivityForResult() is used to launch a sub-Activity and wait for a ___________ from it.

8. ___________ allow communication between different parts of an app or even different apps to react to specific events or actions.

9. PendingIntent encapsulates an Intent and grants a foreign application the ability to perform an operation on behalf of your application, often used in scenarios like ___________ or alarms.

10. PendingIntents support various use cases like notifications, alarms, or deferred actions and can be used with different components like ___________ or BroadcastReceiver.

11. Implicit Intents are helpful when you want to specify a particular component to perform an action.

12. startActivityForResult() does not expect any result back from the sub-Activity it launches.

13. Broadcast Intents facilitate tight coupling between different parts of an app.

14. PendingIntents are primarily used for handling user interactions within an app's UI.

9.3 Implementing Broadcast Receivers

9.3.1 An Introduction to Intent Filters and Broadcast Receivers

Intent Filters and Broadcast Receivers are core components in Android that facilitate inter-component communication and handle system-wide events or messages.

Intent Filters:

Intent Filters are used to declare the capabilities of a component (like an activity, service, or broadcast receiver) in your app's manifest file. They specify the type of intents that a component can respond to. Essentially, they act as a set of criteria that an Intent must match for the system to send it to the appropriate component.

Example - Declaring an Intent Filter for an Activity:

```
<activity android:name=".YourActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="http" />
    </intent-filter>
</activity>
```

In this example:

- <action> specifies the action the component can handle.
- <category> specifies additional categories to qualify the component.
- <data> specifies the data type (in this case, a URL with the "http" scheme) the component can handle.

Broadcast Receivers:

Broadcast Receivers are components that respond to broadcast messages system wide. They listen for broadcast Intents that match their declared Intent Filters and execute code when they receive a matching broadcast.

Example - Creating a BroadcastReceiver in code:

```
public class YourBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Handle the received broadcast here
        if ("com.example.CUSTOM_ACTION".equals(intent.getAction())) {
            // Perform actions based on the received broadcast
        }
    }
}
```

Registering a BroadcastReceiver:
Broadcast Receivers can be registered either statically in the manifest file or dynamically in code using registerReceiver().

Example - Registering a BroadcastReceiver in the manifest:

```
<receiver android:name=".YourBroadcastReceiver">
    <intent-filter>
        <action android:name="com.example.CUSTOM_ACTION" />
    </intent-filter>
</receiver>
```

Example - Registering a BroadcastReceiver dynamically in code:

```
YourBroadcastReceiver receiver = new YourBroadcastReceiver();
IntentFilter filter = new IntentFilter("com.example.CUSTOM_ACTION");
registerReceiver(receiver, filter);
```

Lifecycle Considerations:
Make sure to unregister dynamically registered receivers when they're no longer needed to avoid memory leaks.

Advantages:

- Decoupling Components: Allows different parts of the app or different apps to communicate without strong dependencies.
- System-Wide Notifications: Broadcast Receivers can react to system-wide events, even when your app is not actively running.

9.3.2 Extending application functionality using Intent Filters

Extending an application's functionality using Intent Filters is a powerful technique in Android. It allows your app to offer services to other apps and components by declaring the types of Intents it can handle. Other apps or system components can then use these Intents to interact with specific functionalities within your app.

Declaring Intent Filters:
You declare Intent Filters in your app's manifest file for components like Activities, Services, and BroadcastReceivers to specify the kinds of Intents they can respond to.

Example - Declaring an Intent Filter for an Activity:

```
<activity android:name=".YourActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="http" />
    </intent-filter>
</activity>
```

Extending Functionality:
By using Intent Filters, your app can provide services or respond to requests from other apps or system components that meet the specified criteria. For instance:

Opening URLs: An activity in your app can handle HTTP URLs, allowing other apps to launch your app to view web content.
Receiving Custom Actions: A BroadcastReceiver might listen for specific custom actions, enabling other apps to trigger certain actions within your app.
Example - Receiving Custom Actions:

Let's say your app has a BroadcastReceiver that listens for a custom action to perform a specific task:

```
XML

<receiver android:name=".YourBroadcastReceiver">
    <intent-filter>
        <action android:name="com.example.CUSTOM_ACTION" />
    </intent-filter>
</receiver>
```

Advantages:

- Inter-App Communication: Allows your app to offer services to other apps or components.
- Enhanced Functionality: Enables your app to handle specific tasks initiated by external triggers.

9.3.3 Listening for Broadcast Intents:

Listening for Broadcast Intents in Android involves using a BroadcastReceiver to intercept and respond to system-wide messages or custom broadcasts sent by other apps or components. This mechanism allows your app to react to various events or notifications that occur across the Android system.

Creating a BroadcastReceiver:
You can create a BroadcastReceiver by extending the BroadcastReceiver class and overriding its onReceive() method, which gets triggered when the specified Intent is received.

Example - Creating a BroadcastReceiver:

```
Java

public class YourBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if ("com.example.CUSTOM_ACTION".equals(intent.getAction())) {
            // Perform actions based on the received broadcast
        }
    }
```

Registering the BroadcastReceiver:
You can register your BroadcastReceiver either statically in the manifest file or dynamically in your app's code.

Example - Registering a BroadcastReceiver in the manifest:

```
xml
<receiver android:name=".YourBroadcastReceiver">
    <intent-filter>
        <action android:name="com.example.CUSTOM_ACTION" />
    </intent-filter>
</receiver>
```

Example - Registering a BroadcastReceiver dynamically in code:

```
Java

YourBroadcastReceiver receiver = new YourBroadcastReceiver();
IntentFilter filter = new
IntentFilter("com.example.CUSTOM_ACTION");
registerReceiver(receiver, filter);
```

Responding to Received Broadcasts:
Once your BroadcastReceiver is registered and a matching Intent is received, the onReceive() method gets triggered. You can define the actions your app should perform based on the received broadcast within this method.

Lifecycle Considerations:
Remember to unregister dynamically registered receivers when they are no longer needed to prevent memory leaks. You can do this using unregisterReceiver().

Use Cases:

- System-wide Events: Listening for system events like connectivity changes, battery level updates, etc.
- Custom Interactions: Responding to custom actions initiated by other apps or components.
- Background Tasks: Performing tasks in response to specific triggers even when the app is not actively running.

Advantages:

- Decoupling Components: Allows different parts of your app to communicate without tight coupling.
- Reactivity: Enables your app to respond to various system or custom events in real-time.

9.3.4 Monitoring device state changes:

Monitoring device state changes in Android involves using BroadcastReceivers to listen for system-wide events related to various device states such as connectivity changes, battery status updates, screen state changes, and more. This allows your app to react and adapt to different situations or conditions on the device.

Listening for Connectivity Changes:
You can use a BroadcastReceiver to listen for changes in network connectivity, such as when the device connects to or disconnects from a network.

Example - Listening for Connectivity Changes:

```java
Java

public class ConnectivityReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Check connectivity changes
        // This example checks for network availability
```

Registering Connectivity BroadcastReceiver:
Example - Registering BroadcastReceiver for Connectivity Changes:

```
Java

ConnectivityReceiver receiver = new ConnectivityReceiver();
IntentFilter filter = new
IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
registerReceiver(receiver, filter);
```

Battery Status Updates:
You can also monitor changes in the device's battery status, like when the battery level or charging status changes.

Example - Listening for Battery Status Changes:

```
Java

public class BatteryReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Retrieve battery information
        int batteryLevel = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -
```

Advantages:

- Adaptive Behavior: Allows your app to adapt its behaviour based on different device states.
- Real-time Responsiveness: Enables real-time reactions to changes in device conditions.

9.3.5 Managing manifest Receivers at run time:

Using the Package Manager, you can enable and disable any of your application's manifest Receivers at run time using the setComponentEnabledSetting method. You can use this technique to enable or disable any application component (including Activities and Services), but it is particularly useful for manifest Receivers.

To minimise the footprint of your application, it's good practice to disable manifest Receivers that listen for common system events (such as connectivity changes) when your application doesn't need to respond to those events. This technique also enables you to schedule an action based on a system event — such as downloading a large file when the device is connected to Wi-Fi — without gaining the overhead of having the application launch every time a connectivity change is broadcast.

Below shows how to enable and disable a manifest Receiver at run time:

```
  ComponentName myReceiverName = new ComponentName(this, MyReceiver.class);
  PackageManager pm = getPackageManager();

  // Enable a manifest receiver

pm.setComponentEnabledSetting(myReceiverName,
  PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
  PackageManager.DONT_KILL_APP);

// Disable a manifest receiver
pm.setComponentEnabledSetting(myReceiverName,
  PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
  PackageManager.DONT_KILL_APP);
```

Self-assessment questions:

15. What is the primary purpose of Intent Filters in Android?
A) To register Broadcast Receivers dynamically
B) To specify the type of Intents a component can respond to
C) To handle background tasks in an app
D) To create notifications within an app

16. Which component listens for system-wide broadcast messages in Android?
A) Intent Filter
B) BroadcastReceiver
C) Activity
D) Service

17. What is the purpose of registering a BroadcastReceiver in the manifest file statically?
A) To allow dynamic listening to system events
B) To handle notifications
C) To respond to custom actions
D) To listen for broadcasts even when the app is not running

18. What does using Intent Filters allow an Android app to do?
A) Isolate app functionalities
B) Provide specific services to other apps or components
C) Limit the communication between app components
D) Improve UI responsiveness

19. Which method is used to handle incoming broadcasts in a BroadcastReceiver?

A) handleBroadcast()
B) processIntent()

C) onReceive()
D) handleIntent()

20. ________________ are used to specify the types of Intents a component can respond to in Android.

21. Registering a BroadcastReceiver in the ________________ allows it to listen for broadcasts even when the app is not running.

22.Monitoring device state changes allows an app to adapt its behavior based on different ________________.

23.Using the Package Manager, you can enable or disable manifest Receivers at ________________ using the setComponentEnabledSetting method.

24.Intent Filters allow an app to offer services to other apps or components by declaring the types of Intents they can ________________.

9.5 Summary

Utilising Intents for Component Communication:

- Facilitates Interaction: Intents serve as a versatile mechanism to facilitate communication between various app components, enabling seamless interaction by conveying requests, data, or events.
- Flexible Inter-Component Communication: Allows for both explicit and implicit communication between different components like activities, services, and broadcast receivers, promoting a loosely coupled architecture.

Using Intents to Launch Activities:

- Activity Triggering: Intents serve as the trigger mechanism to start activities within an Android app, enabling navigation between different screens or functionalities.
- Data Exchange: Intents can carry data between activities, facilitating seamless data transfer or passing parameters when launching activities.

Returning Results from Activities:

- Activity Communication: Intents allow activities to send back results to the calling activity using startActivityForResult(), enabling two-way communication between activities.
- Response Handling: onActivityResult() method helps in receiving and handling results returned from child activities, enhancing the flexibility of data exchange between activities.

Native Android Actions:

- Predefined System Actions: Native Android actions encapsulate common functionalities, simplifying app development by providing predefined actions like viewing content, making calls, sharing data, etc.

- Enhanced User Experience: Utilizing native actions ensures a consistent user experience across different apps while leveraging system functionalities.

Introducing Linkify:

- Clickable Text Generation: Linkify simplifies the process of converting text into clickable links for specific patterns like URLs, phone numbers, emails, etc., within TextViews.
- Interactive Text Handling: Enables users to interact directly with recognised patterns, enhancing user engagement and providing ease of navigation within text.

Broadcasting events using Broadcast Intents:

- System-Wide Messaging: Broadcast Intents enable the broadcasting of system-wide messages or events, allowing multiple app components to receive and react to these messages simultaneously.
- Inter-App Communication: Facilitates communication between different apps by sending and receiving broadcast messages, offering a mechanism for apps to react to relevant system events or custom actions.

Using Pending Intents:

- Delayed or Delegated Actions: Pending Intents enable delayed execution of an Intent or delegate the authority to another application to perform an action on behalf of the original application.
- Permission Delegation: Grants permissions to other apps to perform predefined operations on behalf of the app, enhancing flexibility in handling actions asynchronously or at a later time.

Implementing Broadcast Receivers:

- System-Wide Event Handling: Broadcast Receivers listen for system-wide or custom broadcast Intents, allowing apps to react to various system events or notifications.
- Dynamic Event Response: Enables dynamic registration and handling of broadcast events, allowing apps to respond to changes or triggers that occur throughout the Android system.

An Introduction to Intent Filters and Broadcast Receivers:

- Component Capabilities Declaration: Intent Filters declare the capabilities of components in the app, specifying the types of Intents they can respond to, enabling inter-component communication.
- Broadcast Receiver Registration: Intent Filters allow registering Broadcast Receivers to listen for specific actions or events, facilitating the reception and handling of broadcast Intents.

Extending application functionality using Intent Filters:

- Enhanced Component Functionality: Intent Filters extend an app's functionality by allowing components to respond to specific Intents or actions from external sources, increasing interoperability.
- Service Provision: Enables an app to offer services to other apps by declaring the types of Intents it can handle, expanding its usability and interactions with the Android ecosystem.

Listening for Broadcast Intents:

- System-Wide Event Handling: Broadcast Intents enable listening for system-wide or custom events, allowing apps to react to specific events or actions initiated by other apps or system components.
- Dynamic Event Response: Provides the ability to dynamically register Broadcast Receivers to listen for relevant broadcast events, offering real-time reactions to changes or triggers.

Monitoring device state changes:

- State Change Observation: Utilizing Broadcast Receivers, apps can monitor changes in device states like connectivity, battery status, and screen state changes, allowing for adaptive behaviour Adaptive Functionality: Helps apps adapt their behaviour based on different device conditions, providing a better user experience by reacting to varying situations.

Managing manifest Receivers at run time:

- Dynamic Registration Control: Allows for the dynamic registration and unregistration of manifest-declared receivers at runtime, providing more control over when the receiver is active.
- Prevention of Memory Leaks: Ensures proper management of receiver lifecycle, unregistering them when no longer needed to prevent memory leaks and optimise resource usage.

9.6 Terminal Questions

1. What method is used to explicitly start a new Activity using an Intent in Android?
2. How are implicit Intents helpful in launching Activities?
3. How do you receive results from a sub-Activity in the calling Activity in Android?
4. What are some examples of native actions available in Android's Intent class?
5. What's the purpose of Linkify in Android TextViews?
6. How do Broadcast Intents facilitate communication in Android?

7. What's the primary function of Pending Intents in Android?
8. What's the role of Broadcast Receivers in Android?
9. What's the purpose of Intent Filters in Android?
10. How do Intent Filters extend an app's functionality in Android?
11. What's the role of BroadcastReceiver in listening for Broadcast Intents?
12. What advantages do monitoring device state changes offer in Android?
13. How can you dynamically enable or disable manifest Receivers in Android?

9.7 Terminal Answers

1. The startActivity() method is used to launch a specific Activity by passing an Intent with explicit information about the Activity class.
2. Implicit Intents allow actions to be performed without specifying a particular component, enabling dynamic binding at runtime.
3. Results from a sub-activity are received in the calling Activity's onActivityResult() method, identified by a request code.
4. Examples include ACTION_VIEW, ACTION_SEND, ACTION_EDIT, enabling common functionalities like viewing, sharing, and editing.
5. Linkify automatically detects specific patterns in TextViews and converts them into clickable links, enhancing user interaction.
6. Broadcast Intents allow messages to be sent system-wide, enabling various app components to react to specific events or actions.
7. Pending Intents grant permission for other apps or components to perform predefined operations on behalf of an application.
8. Broadcast Receivers listen for broadcast messages and execute specific code when they receive Intents matching their defined Intent Filters.
9. Intent Filters declare the capabilities of components and specify the types of Intents they can respond to in an app.
10. Intent Filters allow an app to offer services to other apps by specifying the types of Intents it can handle.
11. BroadcastReceivers intercept system-wide messages and execute defined code when they receive specific Intents.
12. It allows apps to adapt behavior based on different device conditions, enhancing user experience and system interaction.
13. You can use the setComponentEnabledSetting() method via PackageManager to enable or disable manifest Receivers at runtime.

9.8 Self-assessment answers:

1. B) It starts a sub-Activity and waits for a result from it.
2. C) Linkify.addLinks()

3. B) To communicate between different parts of an app
4. A) Ordered broadcasts
5. A) PendingIntent.getActivity()
6. URLs or specific patterns
7. result
8. Broadcast Intents
9. notifications
10. Activity
11. False (Explanation: Implicit Intents are helpful when you don't want to specify a particular component and want the system to choose the best component dynamically.)
12. False (Explanation: startActivityForResult() expects a result back from the launched sub-Activity).
13. False (Explanation: Broadcast Intents facilitate loose coupling between different parts of an app).
14. True (Explanation: PendingIntents are commonly used in scenarios like notifications, alarms, or deferred actions, which involve user interactions within an app's UI).
15. B) To specify the type of Intents a component can respond to
16. B) BroadcastReceiver
17. D) To listen for broadcasts even when the app is not running
18. B) Provide specific services to other apps or components
19. C) onReceive()
20. Intent Filters
21. manifest file
22. device states
23. run time
24. respond to

INSPIRED BY LIFE

9.9 References

- "Professional Android 4th Edition" by Reto Meier R. (2018). Professional Android 4th Edition. Wrox.

- "Android App Development for Dummies" by Michael Burton and Donn Felker, D. (2015). Android App Development for Dummies for Dummies.

- Reference: BroadcastReceiver - Android Developers

INSPIRED BY LIFE