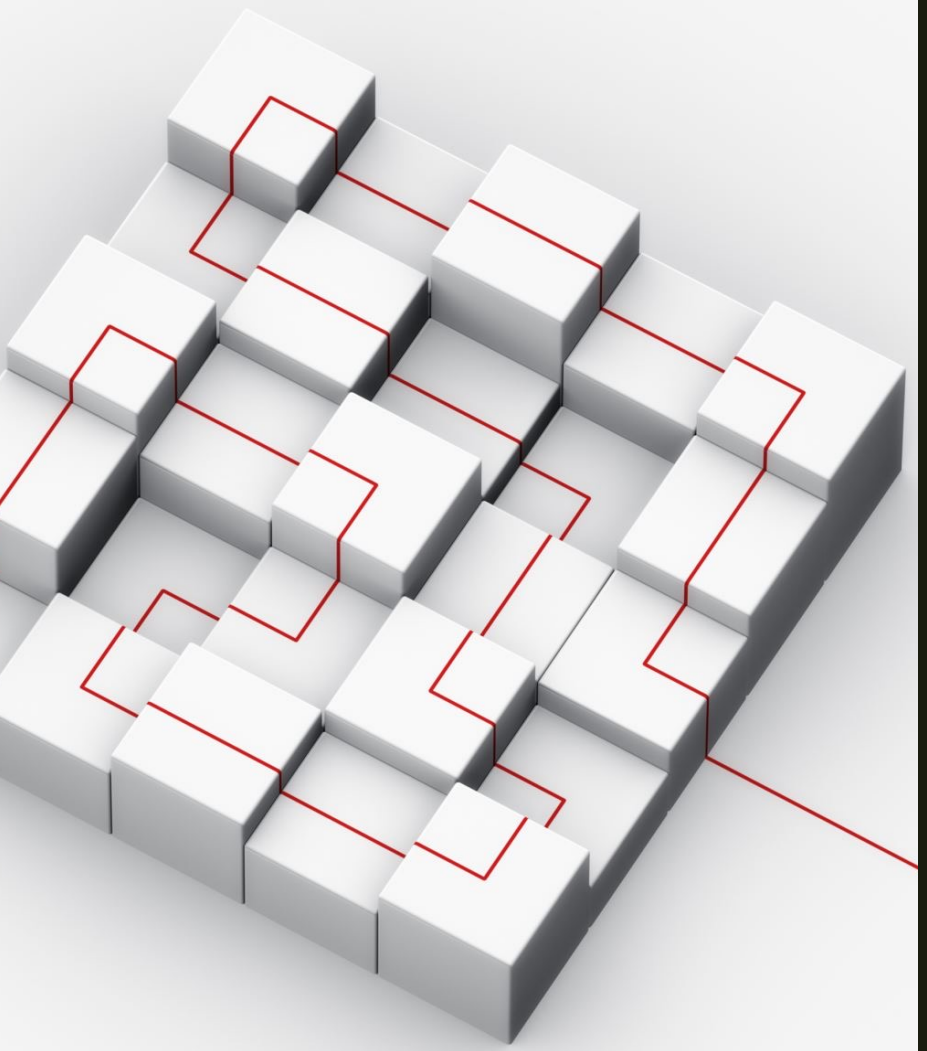
ACTIVITIES, INTENT, NOTIFICATIONS

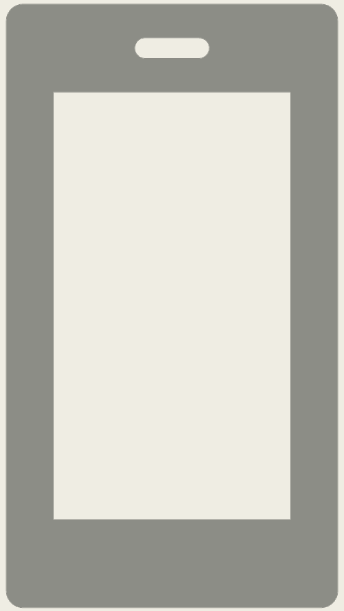
Mobile App Development using Android Studio

Anu Arora, AP, CSE



Activities

- An Activity is a fundamental component of an Android application that represents a single screen with a user interface.
- Activities serve as building blocks for Android apps, and each screen or UI component typically corresponds to an Activity.
- Activities have a lifecycle that includes methods like onCreate, onStart, onResume, onPause, onStop, and onDestroy, which allow you to manage the behavior of your app as it transitions between different states.
- They are used to create the user interface, handle user interactions, and control the flow of the application.



Intents

Intents are a messaging mechanism that enables communication between different components of an Android application or between different Android apps.

There are two types of Intents: Explicit Intents and Implicit Intents.

- Explicit Intents are used to specify a particular component (e.g., an Activity) within the same app that should be started.
- Implicit Intents allow you to request an action to be performed by another app or system component, such as opening a web page, sending an email, or capturing a photo.

Intents can also carry data in the form of key-value pairs (Extras), allowing you to send information between components.

Notifications

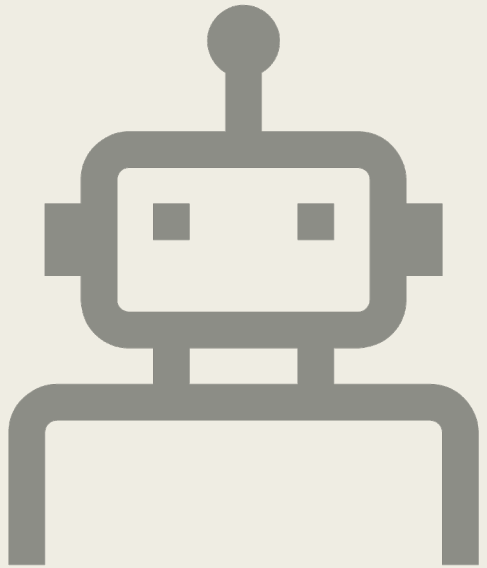
- Notifications are a way for Android apps to provide timely information and alerts to users, even when the app is not in the foreground.
- They appear in the device's notification shade, often with an icon and a message, and can include actions for the user to take.
- Notifications are created and managed using the NotificationCompat.Builder class, allowing developers to customize their appearance and behavior.
- Notifications can be used for various purposes, including informing users of new messages, updates, events, or reminders.
- Notification channels were introduced in Android to categorize and prioritize notifications, giving users control over which types of notifications they want to receive.

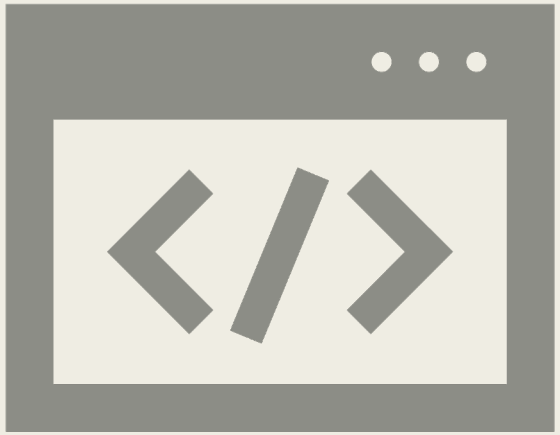
Overview of Android App Components

Android applications are composed of several components, each with its specific purpose and lifecycle. These components work together to create a cohesive user experience.

The four primary Android app components are:

- Activities
- Services
- Broadcast receivers
- Content provider





Activities

- Activities represent the user interface (UI) and serve as the entry point for interactions within the app.
- They are typically associated with a single screen or a user-facing interface element.
- Activities have a well-defined lifecycle that includes methods like `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, and `onDestroy`. These methods allow developers to manage the behavior of an Activity as it transitions between different states.
- Activities can start other Activities, either within the same app (Explicit Intents) or in different apps (Implicit Intents).
- They are responsible for handling user input, displaying content, and managing the user flow within the app.

Services

- Services are background components that perform long-running operations without a user interface.
- They are used for tasks such as playing music, handling network requests, or periodically updating data.
- Services run independently of the UI and can continue to execute even if the user switches to another app or the device goes to sleep.
- Examples of services include media playback services and background location tracking.

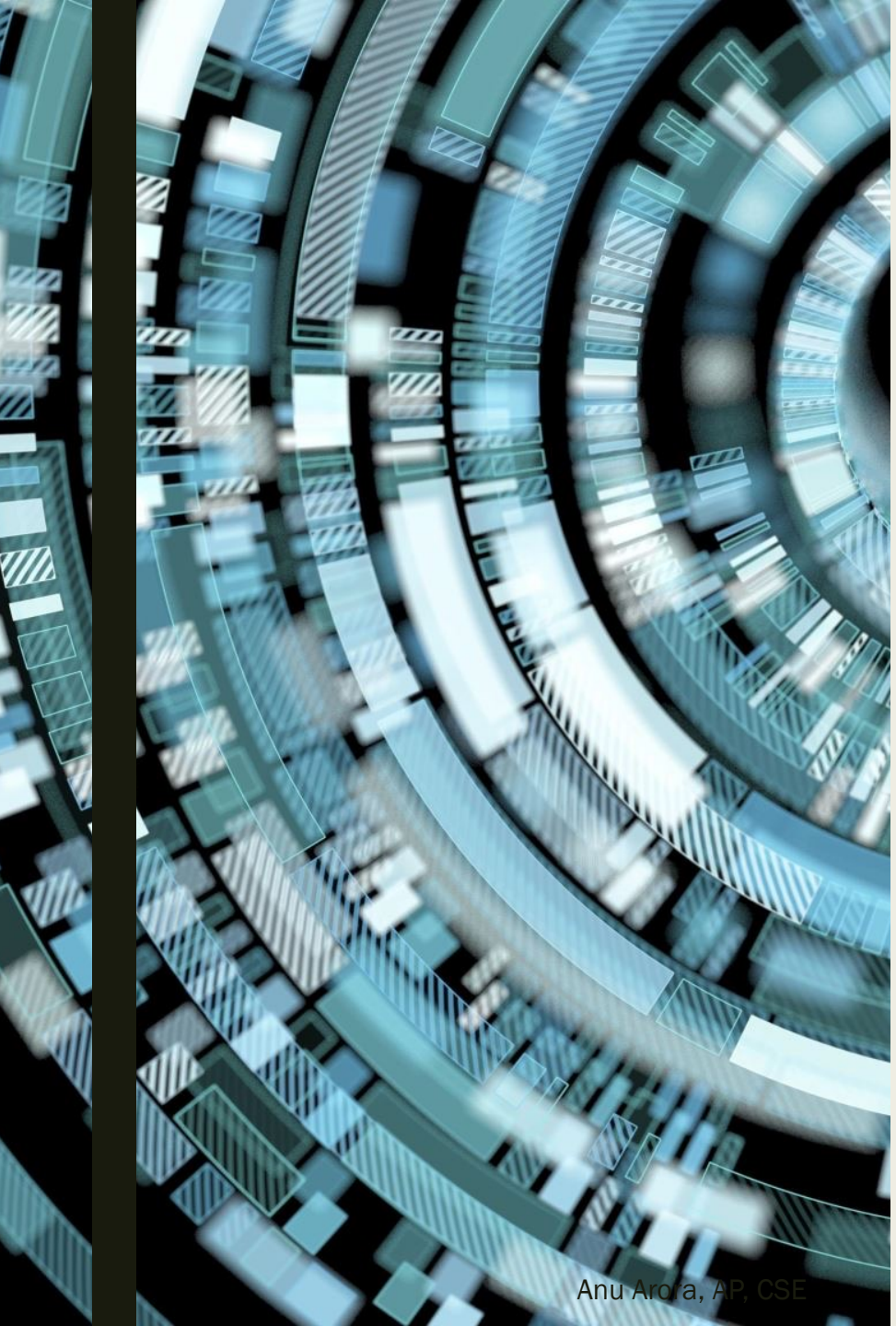
Broadcast Receivers

- Broadcast Receivers are components that listen for and respond to system-wide broadcast messages or custom events.
- They are used to trigger actions or start other components based on specific events or conditions.
- Broadcast Receivers can be registered to respond to system events like battery level changes, network connectivity changes, or incoming SMS messages.
- They are typically lightweight and are not designed for long-running tasks.

Content Providers

- Content Providers allow apps to share data with other apps, facilitating data access and sharing between apps.
- They provide a standardized interface for querying and managing data, typically in databases or content stores.
- Other apps can access data from a Content Provider using a Content Resolver, which abstracts the underlying data source and enforces data access permissions.
- Content Providers are commonly used for managing app data like contacts, calendar events, or media files.

ACTIVITY IN ANDROID



Definition of an Activity

An Activity in Android is a fundamental component of an application that represents a single screen with a user interface.

It serves as a user-facing window that enables users to interact with the app's content, perform actions, and navigate to other parts of the application.

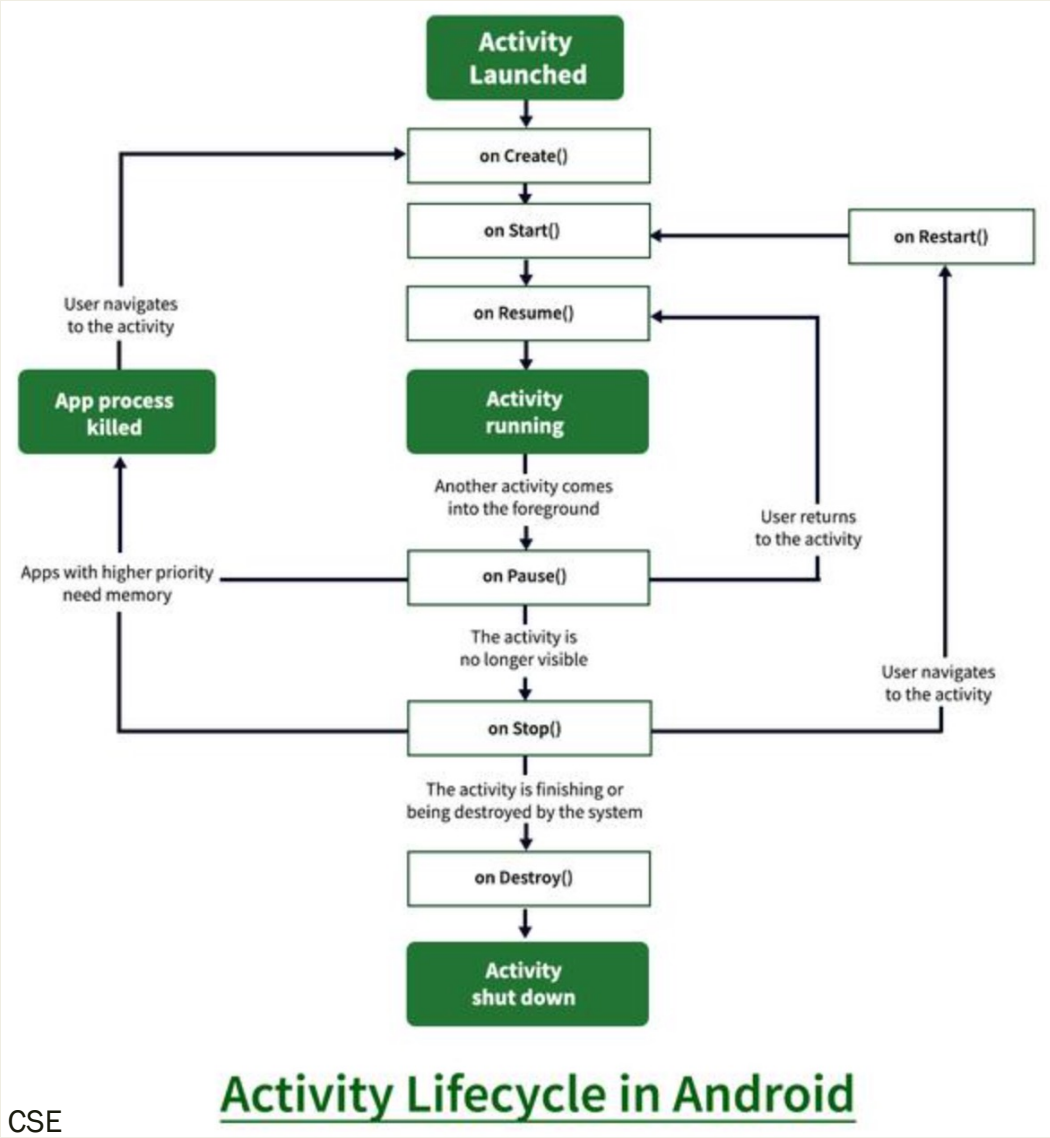
Activities are the primary units of user interaction within an Android app and are responsible for presenting the graphical user interface (GUI) to the user.

Activities as the Building Blocks of Android Apps

Activities are often referred to as the building blocks of Android apps because they encapsulate distinct user interfaces or screens within an application. Here's how Activities function as the foundation for Android app development:

- **User Interaction:** Each Activity corresponds to a specific screen or UI element in the app, such as a login screen, a settings page, or a messaging interface. Users interact with the app by navigating through these Activities.
- **Modularity:** Android encourages modularity and compartmentalization of user interfaces and functionality. By organizing different parts of the app into separate Activities, developers can create a clean and structured app architecture.
- **Navigation:** Activities are linked together to create the app's navigation flow. Transitioning from one Activity to another allows users to move through different sections or features of the app.
- **Lifecycle Management:** Activities have a well-defined lifecycle with methods like `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, and `onDestroy`. Developers can use these lifecycle methods to manage the behavior of each Activity as it transitions between various states.
- **Reusability:** Activities can be reused within the app or even across different Android applications. This reusability promotes efficient development and the creation of consistent user experiences.
- **Parallel Execution:** Multiple Activities can exist simultaneously within an app, allowing users to multitask or switch between different screens seamlessly.
- **Intent-Based Communication:** Activities can start other Activities using Intents, which is a mechanism for inter-component communication. This enables the app to move between different screens and pass data between them.

The Activity lifecycle



The Activity lifecycle

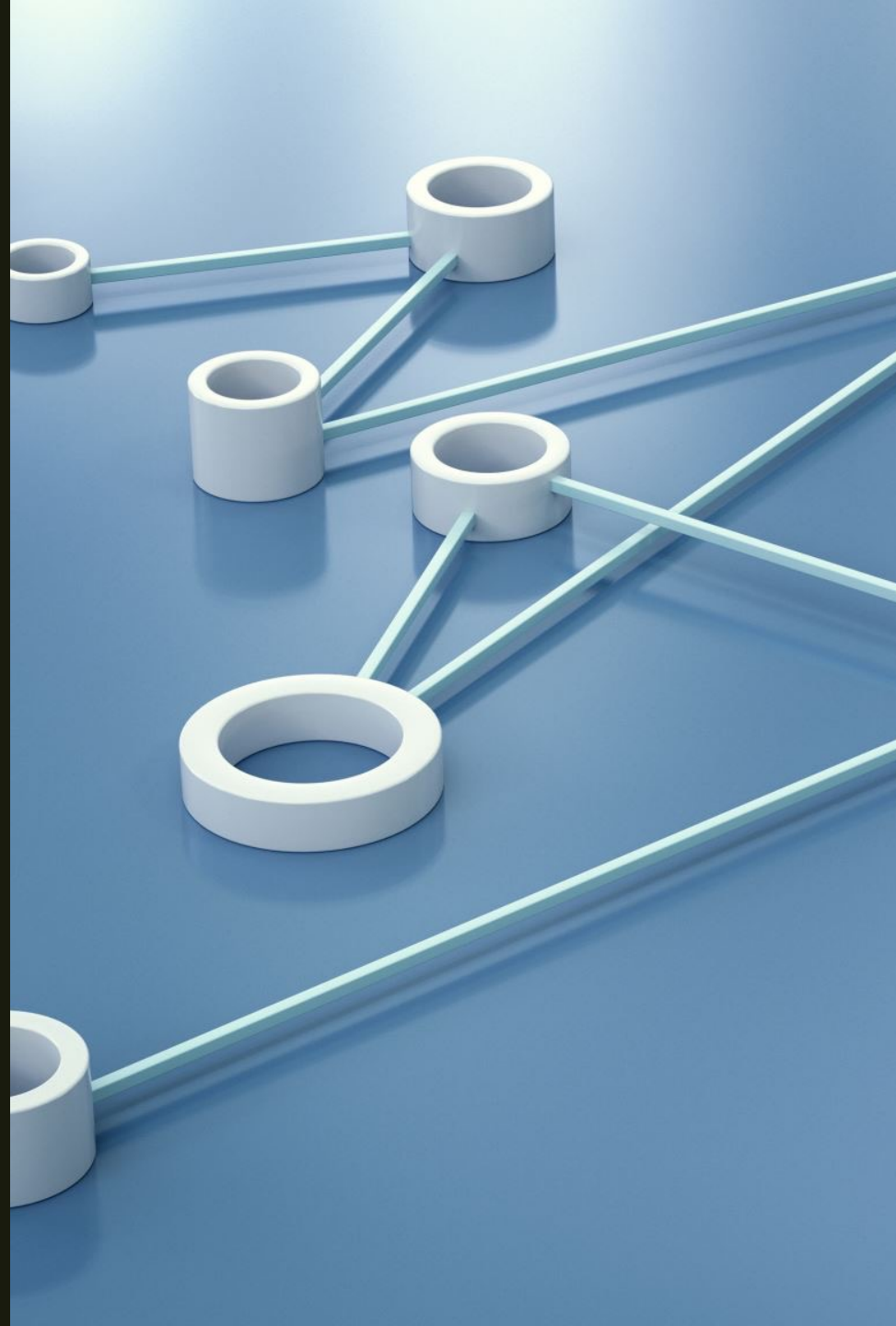
The Activity lifecycle in Android consists of a series of states and methods that allow you to manage the behavior of an Activity as it transitions between these states. Understanding the Activity lifecycle is crucial for controlling the flow of your Android app and ensuring that your app responds appropriately to user interactions and system events. Here's an explanation of the key lifecycle states and methods:

onCreate()

- This is the first method called when an Activity is created.
- It is where you typically perform one-time initialization, such as setting up the user interface (UI) using `setContentView()` and initializing variables.
- You should also restore any saved instance state using the `savedInstanceState` parameter if the Activity is being recreated after being destroyed (e.g., due to a configuration change).

onStart()

- `onStart()` is called when the Activity becomes visible to the user, but it is not yet interactive.
- This is a good place to perform any setup that should occur when the Activity is about to become visible.
- Note that at this point, the Activity is not yet in the foreground and may be partially obscured by other Activities or system UI elements.



The Activity lifecycle

onResume()

- `onResume()` is called when the Activity comes into the foreground and becomes interactive.
- This is where you should perform tasks like starting animations, playing audio or video, and acquiring resources that should only be held while the Activity is in the foreground.
- User interactions, such as button clicks, are typically handled in this state.

onPause()

- `onPause()` is called when the Activity is about to lose focus and move into the background.
- You should use this method to pause or release resources that are no longer needed while the Activity is not visible to the user.
- Long-running operations or operations that require continuous updates (e.g., location updates) should be stopped in this method.

onStop()

- - `onStop()` is called when the Activity is no longer visible to the user.
- - This is a good place to release resources that are not needed when the Activity is not in the foreground.
- - The Activity may still be in memory, but it is not currently being displayed.

onDestroy()

- `onDestroy()` is called when the Activity is being destroyed, either because the user has navigated away from it or because the system is reclaiming resources.
- This is where you should perform cleanup tasks, release resources, and unregister any registered listeners or receivers.
- After this method completes, the Activity object is no longer valid.

Step 1: Create a New Activity

1. Open your Android Studio project.

2. In the Project Explorer or Android view, right-click on the `java` or `kotlin` directory corresponding to your app's package and choose "New" -> "Java Class" or "Kotlin Class."

3. Give your new Activity a meaningful name, such as "NewActivity." Make sure it extends the `AppCompatActivity` class for modern Android development.

4. Android Studio will generate the code for your new Activity. It should look something like this:

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
public class NewActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_new);
    }}
```

5. You can customize the `onCreate` method to set the content view and perform any other initialization required for this Activity.

Step 2: Define the Activity in the Manifest

6. Open the `AndroidManifest.xml` file in your project.

7. Add the new Activity to the manifest file by including an `<activity>` element inside the `<application>` element. Specify the name of the Activity using its fully-qualified class name

(e.g., `"com.example.myapp.NewActivity"`).

```
<activity android:name=".NewActivity">
    <!-- Optionally, you can set the label and other attributes here -->
</activity>
```

Step 3: Launch the New Activity using an Intent

8. In your existing Activity (the one from which you want to launch the new Activity), create an Intent to start the new Activity. You can do this in response to a button click, menu item selection, or any user-triggered event.

For example:

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

Here, `this` refers to the current Activity, and `NewActivity.class` is the class name of the Activity you want to launch

9. If you need to pass data from the current Activity to the new Activity, you can add extras to the Intent:

```
Intent intent = new Intent(this, NewActivity.class);
intent.putExtra("key", "value");
startActivity(intent);
```

Step 4: Handling Data in the New Activity

10. In the `onCreate` method of your new Activity (i.e., `NewActivity` in this example), you can retrieve any data passed from the previous Activity using the `getIntent()` method:

```
Intent intent = getIntent();
String value = intent.getStringExtra("key");
```

Now, when you run your app and trigger the event that launches the new Activity, it should open the `NewActivity` you created.

INTRODUCTION TO INTENTS

Introduction to Intents

In Android app development, Intents are a powerful and essential concept that facilitate communication and interaction between different components of an Android application or even between different Android applications.

Intents serve as a messaging mechanism to initiate various actions, such as starting activities, broadcasting messages, and binding services, making them a fundamental part of the Android framework.

Intents are crucial for enabling flexibility and extensibility in Android apps.

They allow developers to design apps that can seamlessly interact with other apps, system services, and components, enhancing the overall user experience.

Explicit and Implicit Intents

There are two main types of Intents in Android: Explicit and Implicit Intents.

1. Explicit Intents

- Explicit Intents are used for specifying a particular component within the same Android app that should be started or executed.
- They explicitly define the target component (e.g., Activity, Service) by providing the component's fully-qualified class name.
- Explicit Intents are commonly used for navigating within an app, launching specific activities, and invoking services that are part of the same application.

Example of creating and using an explicit Intent:

```
Intent explicitIntent = new Intent(this, TargetActivity.class);
startActivity(explicitIntent);
```

2. Implicit Intents

- Implicit Intents are more flexible and do not specify the exact component that should handle the intent. Instead, they declare the desired action and optionally the data type (e.g., text, image) or URI.
- Android's system then determines which component can handle the Intent based on its declared action and data type. Multiple apps may respond to the same Implicit Intent.
- Implicit Intents are commonly used for actions that can be performed by different apps, such as sharing content, opening a web page, sending emails, or capturing photos.

Example of creating and using an implicit Intent to open a web page:

```
Intent implicitIntent = new Intent(Intent.ACTION_VIEW,
Uri.parse("https://www.example.com"));
startActivity(implicitIntent);
```

Using Explicit Intents to Start Specific Activities Within Your App

Explicit Intents are used when you want to start a specific component, typically an Activity, within your own Android app. They explicitly specify the target component by providing its fully-qualified class name.

Here's how to use Explicit Intents:

```
// Create an Explicit Intent to start a specific Activity within your app
Intent explicitIntent = new Intent(CurrentActivity.this, TargetActivity.class);
// Optionally, you can pass data to the TargetActivity
explicitIntent.putExtra("key", "value");
// Start the TargetActivity
startActivity(explicitIntent);
```

`CurrentActivity.this`: The current context or Activity from which you are creating the Intent.

`TargetActivity.class`: The class name of the Activity you want to start.

Using Implicit Intents to Open Components Outside Your App

Implicit Intents are used when you want to trigger an action that can be handled by various components, including those outside your app. Implicit Intents declare the desired action and, optionally, the data type or URI. Android's system then determines which component can handle the Intent. Here's how to use Implicit Intents:

```
// Create an Implicit Intent to open a web page
Intent implicitIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("https://www.example.com"));

// Start an Activity that can handle this Intent
startActivity(implicitIntent);
```

`Intent.ACTION_VIEW`: Specifies the action to view the data (in this case, a web page).

`Uri.parse("https://www.example.com")`: Specifies the data or URI to be viewed.

Examples of Common Implicit Intents

1. Opening a Web Page:

```
Intent webIntent = new Intent(Intent.ACTION_VIEW,
Uri.parse("https://www.example.com"));

startActivity(webIntent);
```

2. Sending an Email:

```
Intent emailIntent = new Intent(Intent.ACTION_SENDTO,
Uri.parse("mailto:recipient@example.com"));

emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Subject");

emailIntent.putExtra(Intent.EXTRA_TEXT, "Message body");

startActivity(emailIntent);
```

3. Dialing a Phone Number:

```
Intent dialIntent = new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:+1234567890"));

startActivity(dialIntent);
```

4. Sharing Content:

```
Intent shareIntent = new
Intent(Intent.ACTION_SEND);

shareIntent.setType("text/plain");

shareIntent.putExtra(Intent.EXTRA_TEXT,
"Share this content.");

startActivity(Intent.createChooser(shareIntent,
"Share via"));
```

5. Taking a Photo (Camera App):

```
Intent cameraIntent = new
Intent(MediaStore.ACTION_IMAGE_CAPTURE);

startActivityForResult(cameraIntent,
CAMERA_REQUEST_CODE);
```

PASSING DATA FROM ONE ACTIVITY TO ANOTHER

Passing Data from One Activity to Another

Suppose you have two Activities: `ActivityA` and `ActivityB`. You want to pass a string value from `ActivityA` to `ActivityB`.

In ActivityA:

```
// Define the data to be passed
String message = "Hello, ActivityB!";
// Create an Intent to start ActivityB
Intent intent = new Intent(ActivityA.this, ActivityB.class);
// Add the data to the Intent as an extra
intent.putExtra("message_key", message);
// Start ActivityB
startActivity(intent);
```

Passing Data from One Activity to Another

In ActivityB:

// Retrieve the data from the Intent

Intent intent = getIntent();

String message = intent.getStringExtra("message_key");

// Now, you can use the 'message' variable in ActivityB as needed

In this example:

- In `ActivityA`, we create an Intent and add an extra with the key `"message_key"` and the string value `"Hello, ActivityB!"`.
- We then start `ActivityB` with this Intent.
- In `ActivityB`, we retrieve the Intent using `getIntent()`, and we extract the extra data using `getStringExtra("message_key")`.

Passing Data Back from ActivityB to ActivityA:

If you want to pass data back from `ActivityB` to `ActivityA`, you can use the `setResult()` method and send the result back through the Intent.

In ActivityB (when data is ready to be sent back):

```
// Define the data to be sent back
String response = "Data from ActivityB to ActivityA";
// Create an Intent to hold the response data
Intent resultIntent = new Intent();
resultIntent.putExtra("response_key", response);
// Set the result code and attach the result Intent
setResult(RESULT_OK, resultIntent);
// Finish ActivityB to return to ActivityA
finish();
```

Passing Data Back from ActivityB to ActivityA:

In ActivityA (handling the result from ActivityB):

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == REQUEST_CODE_ACTIVITYB) {
        if (resultCode == RESULT_OK) {
            // Retrieve the response data from ActivityB
            String response = data.getStringExtra("response_key");
            // Now, you can use the 'response' variable in ActivityA as needed
        } } }
```

In this example:

- In `ActivityB`, we create an Intent (`resultIntent`) to hold the response data and use `setResult()` to set the result code to `RESULT_OK`. We then attach the result Intent and finish `ActivityB`.
- In `ActivityA`, we override `onActivityResult()` to handle the result from `ActivityB`. We check for the `requestCode` (which should match the one used when starting `ActivityB`) and the `resultCode` to ensure it's `RESULT_OK`. If everything is as expected, we retrieve the response data from the received Intent.

This mechanism allows you to pass data between Activities in both directions, enhancing the interactivity and functionality of your Android app.

FRAGMENTS IN ANDROID

Introduction to Fragments

In the realm of Android app development, Fragments are indispensable components that offer a flexible and modular approach to building user interfaces.

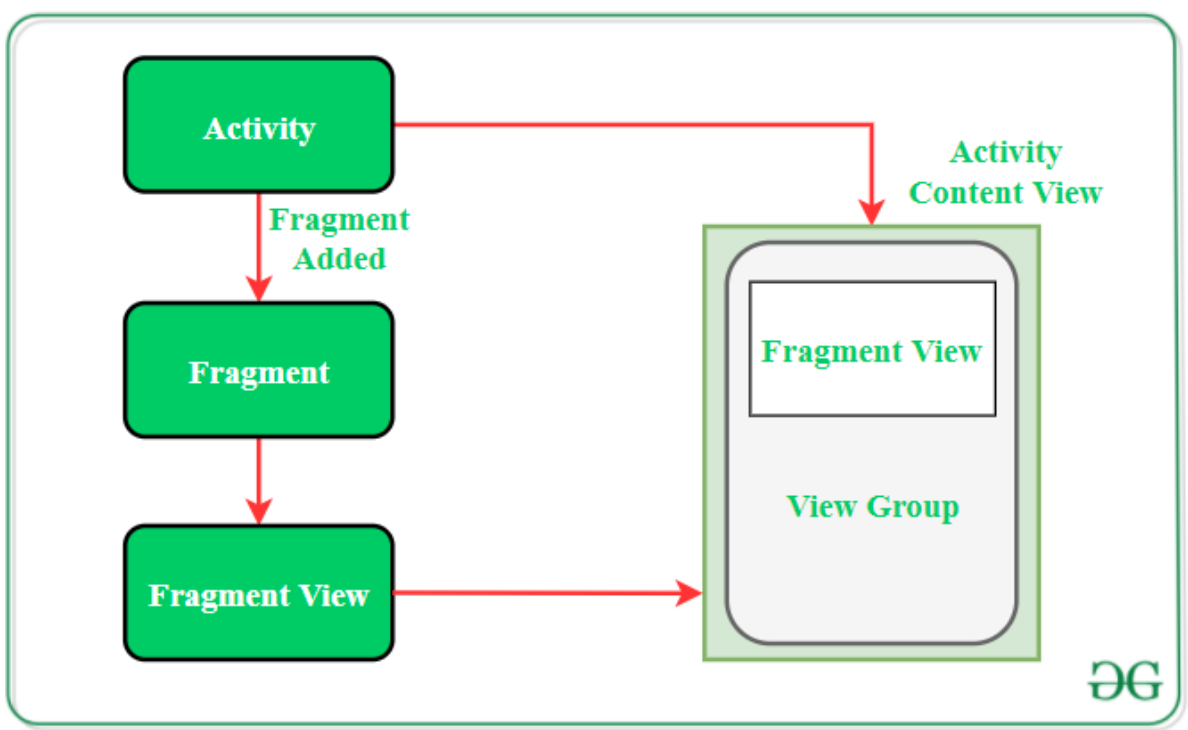
Fragments are essentially "sub-activities" or self-contained UI components that can be combined within a single activity to create dynamic and responsive user interfaces.

They provide a way to design apps that can adapt gracefully to various screen sizes and orientations, offering a more versatile user experience.

Fragments in Android App Development

There are several compelling reasons to incorporate Fragments into your Android app development:

1. **Reusability:** Fragments can be reused across multiple activities, promoting code modularity and reducing redundancy. This reusability ensures a consistent look and behavior throughout your app.
2. **Responsive Design:** Fragments enable the creation of responsive layouts that can adjust to different screen sizes and orientations. This is particularly valuable in a world with diverse Android devices.
3. **Multi-Pane Interfaces:** Fragments are ideal for creating multi-pane interfaces, such as tablet layouts with two or more panels displayed side by side. They allow you to present more content and functionality on larger screens.
4. **Fragment Transactions:** You can dynamically add, replace, or remove fragments within an activity. This makes it easy to adapt your app's UI based on user interactions or screen size changes.
5. **Parallel Execution:** Multiple fragments within the same activity can run concurrently, enhancing multitasking and providing a more immersive user experience.
6. **Code Separation:** Fragments encourage a separation of concerns by allowing you to encapsulate UI and logic within individual fragments. This improves code maintainability and makes it easier to collaborate with other developers.



THE FRAGMENT LIFECYCLE

The Fragment Lifecycle

Fragments have their own lifecycle, which consists of various methods that get called at different stages of the fragment's existence. Understanding the fragment lifecycle is crucial for managing their behavior effectively. Here are some key lifecycle methods:

- **onAttach():**This method is called when the fragment is attached to its parent activity. It provides a reference to the hosting activity and is an appropriate place to initialize variables related to the activity.
- **onCreate()**: This method is used for basic initialization of the fragment. You can initialize essential components and perform setup tasks here.
- **onCreateView():** This method is responsible for creating the fragment's view hierarchy. It returns the root view of the fragment's layout, which can be inflated from an XML layout file or dynamically created in code.
- **onViewCreated():** Called after `onCreateView()`, this method is a good place to perform any additional setup that involves the created view, such as setting up user interface elements.
- **onStart():** This is where the fragment becomes visible to the user but is not yet interactive. You can start animations or load data in this phase.
- **onResume():** In this state, the fragment is active and interactive. User interactions and UI updates typically occur here.

The Fragment Lifecycle

- **onPause():** When the fragment is losing focus, you should pause ongoing operations, save data, or perform cleanup in this method.
- **onStop():** The fragment is no longer visible to the user in this state. It's a suitable place to release resources that are no longer needed.
- **onDestroyView():** This method is called when the fragment's view hierarchy is being destroyed. Clean up any resources tied to the UI here.
- **onDestroy():** Perform final cleanup and resource release in this method, as the fragment is about to be removed.
- **onDetach():** The final step in the lifecycle, this method is called when the fragment is detached from the activity. Release any references to the activity or other objects here.

Adding a Fragment

To add a Fragment to your Activity, you typically do this within your Activity's `onCreate` method or in response to a user action. Here's an example of adding a Fragment to an Activity:

```
// Create a Fragment instance

MyFragment myFragment = new MyFragment();

// Get the FragmentManager

FragmentManager fragmentManager = getSupportFragmentManager();

// Begin a Fragment transaction

FragmentTransaction transaction = fragmentManager.beginTransaction();

// Add the Fragment to the container (R.id.fragment_container in this case)

transaction.add(R.id.fragment_container, myFragment);

// Commit the transaction

transaction.commit();
```

In this example:

- `MyFragment` is the Fragment you want to add.
- `getSupportFragmentManager()` retrieves the FragmentManager for the Activity.
- `beginTransaction()` starts a new transaction.
- `add()` adds the Fragment to the specified container (identified by its ID).
- `commit()` commits the transaction.

Replacing a Fragment

Replacing a Fragment allows you to switch one Fragment with another in the same container. Here's an example of replacing a Fragment:

```
// Create a new Fragment instance
NewFragment newFragment = new NewFragment();
// Get the FragmentManager
FragmentManager fragmentManager = getSupportFragmentManager();
// Begin a Fragment transaction
FragmentTransaction transaction = fragmentManager.beginTransaction();
// Replace the current Fragment with the new one
transaction.replace(R.id.fragment_container, newFragment);
// Commit the transaction
transaction.commit();
```

In this example, we use `replace()` instead of `add()` to replace the current Fragment with `NewFragment`.

Removing a Fragment

Removing a Fragment is as simple as calling `remove()` in a Fragment transaction. Here's an example:

```
// Get the FragmentManager
FragmentManager fragmentManager = getSupportFragmentManager();
// Begin a Fragment transaction
FragmentTransaction transaction = fragmentManager.beginTransaction();
// Find the Fragment by its tag or ID and remove it
Fragment fragmentToRemove = fragmentManager.findFragmentByTag("tag_name");
if (fragmentToRemove != null) {
    transaction.remove(fragmentToRemove);
}
// Commit the transaction
transaction.commit();
```

In this example, we first find the Fragment you want to remove using `findFragmentByTag()` or another suitable method. If the Fragment exists, we remove it using `remove()` in the transaction.

NOTIFICATION IN ANDROID

Introduction to Notifications in Android

Notifications in Android are a vital communication mechanism between apps and users.

They provide a way for apps to deliver important information, updates, and alerts to users, even when the app is not actively in use.

Notifications typically appear in the device's notification shade and can include text, icons, and actions that allow users to interact with the app without launching it.

They play a crucial role in keeping users informed and engaged with an app.

Importance of Notifications for User Engagement

Notifications are essential for several reasons:

1. **Timely Information:** Notifications allow apps to provide real-time updates, news, messages, and alerts to users, ensuring that they stay informed about important events or changes.
2. **User Engagement:** Notifications can encourage users to interact with the app by reminding them of its existence, prompting them to take specific actions, or inviting them to explore new content.
3. **Personalization:** Notifications can be customized to deliver content that is relevant to individual users, increasing their engagement with the app.
4. **Task Completion:** Users can perform actions directly from notifications, such as replying to messages, marking emails as read, or snoozing alarms, without navigating to the app.

How to Create and Customize Notifications

To create and customize notifications in Android, you typically use the `NotificationCompat.Builder` class, which provides a builder pattern to construct notifications. Additionally, you can use notification channels to categorize and manage notifications.

Here's a brief overview of the process:

1. Create a Notification Builder:

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(context, channelId)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("Notification Title")
    .setContentText("Notification Content")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT);
```

2. Create a Notification Channel (Android 8.0+):

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        CharSequence name = "Channel Name";
        String description = "Channel Description";
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new NotificationChannel(channelId, name, importance);
        channel.setDescription(description);
        NotificationManager notificationManager =
context.getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
```

3. Customize Actions (Optional):

```
Intent intent = new Intent(context, YourActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(context, 0, intent, 0);
builder.setContentIntent(pendingIntent);
```

4. Show the Notification:

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

Code Examples for Displaying Notifications:

Here's a simplified example of creating and showing a notification in Android:

```
// Create a NotificationCompat.Builder
NotificationCompat.Builder builder = new NotificationCompat.Builder(context, channelId)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("New Message")
    .setContentText("You have a new message from John.")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT);
// Create a NotificationManagerCompat
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
// Show the notification
int notificationId = 1; // A unique ID for the notification
notificationManager.notify(notificationId, builder.build());
```

IMPORTANCE OF ACTIVITIES, INTENTS, AND NOTIFICATIONS

Activities

User Interface and Interaction: Activities are the building blocks of Android user interfaces. They represent individual screens or UI components, allowing users to interact with your app. Activities are responsible for displaying content, receiving user input, and managing the user interface.

Navigation: Activities define the navigation flow within your app. Users move between different Activities to access various features and content. Properly managing Activities ensures a seamless and intuitive user experience.

Lifecycle Management: Activities have a well-defined lifecycle with methods for initialization, interaction, and cleanup. Understanding the Activity lifecycle is crucial for managing resources, saving and restoring state, and delivering a responsive user interface.

Multitasking: Android allows multiple Activities to run concurrently, enabling users to switch between different screens or tasks effortlessly. This multitasking capability enhances user productivity and flexibility.

Intents

- **Inter-Component Communication:** Intents serve as a versatile communication mechanism between different app components, including Activities, Services, Broadcast Receivers, and even external applications. They facilitate data exchange and action invocation.
- **Launching Components:** Intents enable you to start Activities and initiate various actions, both within your app (Explicit Intents) and in other apps (Implicit Intents). This flexibility allows for deep linking and seamless integration with other apps.
- **Data Passing:** Intents allow you to pass data between components, making it possible to share information and context between different parts of your app. This is essential for creating feature-rich and interconnected apps.

Notifications

- **User Engagement:** Notifications are a direct channel to engage users with your app, even when it's not actively in use. They provide timely updates, alerts, and personalized content, keeping users informed and connected.
- **Interaction Without Launching:** Users can perform actions directly from notifications, such as responding to messages, dismissing alerts, or even completing tasks like email management or calendar events without opening the app. This enhances user convenience and productivity.
- **Personalization:** Notifications can be tailored to individual user preferences and behaviors, delivering content that matters most to them. Personalized notifications increase user retention and engagement.
- **Re-engagement:** Notifications remind users about your app's presence and encourage them to return. Whether it's a news update, a social media mention, or a special offer, notifications bring users back to your app.