Mobile Programming

Course 2

Activities and Intents

Qiong LIU

qiong.liu@cyu.fr

CY Tech, CY Cergy Paris University

2024 - 2025

Principles of Android Activities

- **Activities can launch other activities**, creating a sequence of screens.
- Each **activity represents a screen** in the app, managing user interactions.
- At any given time, **only one activity is visible** to the user (managed through the activity stack).
- **Hidden activities are paused** to save memory and battery, and may be destroyed if needed by the system.

The Activity Lifecycle.

The Activity Lifecycle

- Android manages each activity through a well-defined lifecycle.
- The lifecycle consists of multiple callback methods that indicate transitions between activity states.

- Android manages each activity through a well-defined lifecycle.
- The lifecycle consists of multiple callback methods that indicate transitions between activity states.

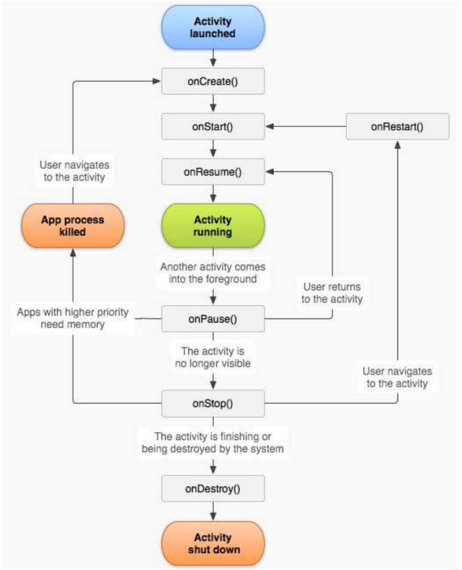
- Key lifecycle methods include:
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - `onPause()`
  - `onStop()`
  - `onDestroy()`

Activity Lifecycle: Key States

An activity can exist in three primary states:

- **Resumed (Running)**
  - The activity is fully in the foreground, and the user can interact with it.
  - This is the active state where the app is running and receiving user input.
- **Paused**
  - The activity is partially covered by another activity that is in the foreground.
  - It does not receive user input and is unable to execute any code.
  - This state typically occurs when the other activity is semi-transparent or doesn't fully cover the screen.
- **Stopped**
  - The activity is completely hidden from view and in the background.
  - The instance and state information (like member variables) are reserved.
  - The system may kill a stopped activity to free up memory for other apps.

Activity Lifecycle


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

Activity Lifecycle

Good implementation of the lifecycle callbacks can help your app **avoid the common problems** as following:

- Crashing if the user receives a phone call or switches to another app while using your app (i.e., `onPause()`, `onStop()`).
- Consuming valuable system resources when the user is not actively using it.
- Losing the user's progress if they leave your app and return to it at a later time.
- Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.
  - Here the activity is recreated when the user rotates the screen between "portrait" and "landscape mode.

Important Lifecycle Callbacks

The two most important callback methods in the activity lifecycle are:

- **onCreate()**
  - **Mandatory method**: You must implement this method.
  - This is where you initialize your Activity and call `setContentView()` to define the layout.
  - Takes one parameter: `Bundle`. It is `null` if the Activity is being created for the first time, or contains the Activity's previous state if it was destroyed by the system (e.g., to free memory).
- **onPause()**
  - Called when the system signals that the user is leaving your Activity.
  - It doesn't always mean the Activity will be destroyed, but it's a good place to save any changes that need to persist, as the user may not return.

Bundle in ‘onCreate()‘

- A Bundle in Android acts as a temporary memory space that saves the current state information of an Activity.
- When an Activity is (re-)created, the ‘onCreate()‘ method receives a ‘Bundle‘.
  - If the ‘Bundle‘ is `null`, it means the Activity is being created for the first time.
  - If the ‘Bundle‘ contains data, the Activity is being recreated (e.g., after rotation), and we can restore the previous state.

Code Example

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (savedInstanceState != null) {
        String data = savedInstanceState.getString("key");
    }
}
```

Instance State

- The Instance State allows an activity to save its current state, such as user input or UI configurations, in a ‘Bundle‘ before the activity is destroyed and recreated.
- The system automatically saves certain View states (e.g., text in ‘EditText‘, scroll positions) to the instance state.

Saving and Restoring Instance State

- The system automatically saves the state of each ‘View‘ in your layout (such as the text entered in an ‘EditText‘) using the ‘Bundle‘. When the activity is destroyed and recreated, the layout state is restored to what it was before.
- To save additional custom data during the activity lifecycle, you can use two key callbacks:

```java
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putString("key", "value");
    // Save any other necessary data
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    String value = savedInstanceState.getString("key");
    // Restore the saved data
}
```

Understanding ‘outState‘ vs ‘savedInstanceState‘

- **‘outState‘** in ‘onSaveInstanceState()‘
  - Used to manually save custom data (like variables or UI state) before the Activity is temporarily destroyed.
  - This data is stored temporarily, but will be available if the Activity is recreated within the same session.
  - It is not a permanent storage mechanism.
- **‘savedInstanceState‘** in ‘onCreate()‘ or ‘onRestoreInstanceState()‘
  - Used to restore the previous state of the Activity when it is recreated (e.g., after screen rotation).
  - This automatically restores the layout and UI state, and can also restore custom data you saved.

Example: Saving Data with ‘onSaveInstanceState‘

Scenario: When the screen rotates, the activity is destroyed and recreated. We need to save the user’s input (e.g., text in an EditText) before the activity is destroyed.

Saving Data Before Destruction:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    // Save the text entered by the user
    outState.putString("user_input",
                            editText.getText().toString());
}
```

- **‘onSaveInstanceState()‘** is called before the activity is temporarily destroyed (e.g., during screen rotation).
- We save the text entered in the EditText into the ‘outState‘ bundle.
- This data will be stored temporarily and can be restored later.

Example: Restoring Data with ‘savedInstanceState‘

Scenario: After the screen rotates, the activity is recreated. We need to restore the text that was entered before the screen rotated. **Restoring Data After Recreation:**

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

    if (savedInstanceState != null) {
        // Restore the saved text
        String savedText = savedInstanceState.getString(
            "user_input");
        editText.setText(savedText);
    }
}
```

- **‘savedInstanceState‘** contains the data we saved before (in ‘onSaveInstanceState()‘).
- We check if there’s any saved data and restore the user input.

Intents.

Intents

- An **Intent** is a messaging object used to request actions like starting an activity, starting services, broadcasting messages, etc.
- Common uses:
  - Starting a new Activity (e.g., navigating to another screen).
  - Sending databetween components.
- Two types of Intents:
  - **Explicit Intent**: Specifies the target component directly (e.g., starting a specific activity).
  - **Implicit Intent**: Declares a general action, allowing the system to find a matching component (e.g., opening a web page).

Explicit Intent

An explicit intent specifies the target component explicitly by providing the component's class name or its fully qualified package name.

```
Intent i = new Intent(this, SecondActivity.class);
startActivity(i);
```

Dont forget to register the second activity in the AndroidManifest.xml:

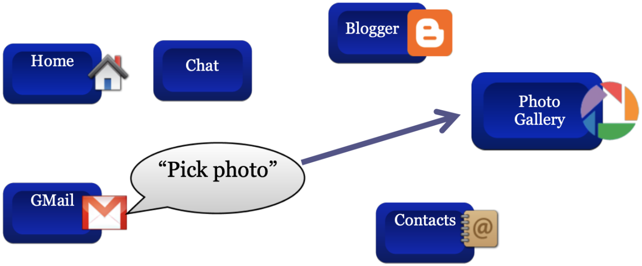
```
<activity
    android:name=".SecondActivity"
    android:label="@string/app_name">
</activity>
```

Example: Explicit Intent

Set a button for navigating to the next Activity

```
// Button for navigating to the next page
    Button buttonJump = findViewById(R.id.mybuttonjump);
    buttonJump.setOnClickListener(new View.OnClickListener()
         {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(MainActivity.this,
                MainActivity2.class);
            startActivity(intent);
        }
    });
```

Implicit Intent Example



- An Intent allows an app to request an action (like "Pick a photo") without specifying the exact component to handle it.
- Android chooses the best app or component to handle the requested action (e.g., opening the photo gallery or a camera app).

Implicit Intents

An implicit intent does not specify the target component by its name; instead, it specifies an action to be performed, and the system determines which component can best handle that action based on the intent's information.

```
// view contact list
Intent i = new Intent(Intent.ACTION_VIEW,
    android.provider.ContactsContract.Contacts.CONTENT_URI);
// make a call
Intent i = new Intent(Intent.ACTION_CALL_BUTTON,null);
// view picture gallery
Intent i = new
    Intent(Intent.ACTION_VIEW,android.provider.MediaStore.Im
    ages.Media.INTERNAL_CONTENT_URI);
startActivity(i);
```

URL: Uniform Resource Locator

Passing Data with Intents: Sending Data

1) **Calling Activity: Sending data**

```
// Create an Intent to start SecondActivity
Intent intent = new Intent(this, SecondActivity.class);

// Attach data to the Intent using putExtra
intent.putExtra("idNumber", "123"); // key-value pair

// Start the activity
startActivity(intent);
```

- Create an Intent: This intent starts the `SecondActivity`. `this` refers to the current activity.
- Attach data: `putExtra("idNumber", 123)` attaches an extra key-value pair to the intent.
- Custom behavior: If the ID number changes, `SecondActivity` can behave differently based on the received value.
- The ‘startActivity()‘ method is called to launch ‘SecondActivity‘ with the attached data.

Passing Data with Intents: Receiving Data

2)textbfCalled Activity: Receiving data

```
// Get the Intent that started this activity
Intent intent = getIntent();

// Retrieve the data attached to the Intent using the key
String value = intent.getStringExtra("idNumber");

// Verify the received value in the log
Log.i(TAG, value); // Should output "123"
```

- The **Called Activity** (‘SecondActivity‘) retrieves the Intent that started it using ‘getIntent()‘.
- The ‘getStringExtra("key")‘ method is used to retrieve the value associated with the key ‘"idNumber"‘.
- The retrieved value (in this case, ‘"123"‘) is logged for verification.

Service.

Services

- A **Service** is an Android component that runs in the background to perform long-running operations.
- Services do not provide a user interface (unlike activities) and are primarily used for background tasks.
- Common use cases include:
  - Playing music in the background
  - Downloading files or fetching data from the network
  - Handling background tasks even if the user switches to another app
- Services run on the **main thread** by default, so heavy operations should be run in a separate thread (e.g., using ‘AsyncTask‘, ‘Thread‘, or ‘Handler‘).

Types of Services

There are two main types of Android services:

- Started Service (unbounded services):
  - A service that is started when a component (like an Activity) calls `startService()`.
  - Runs in the background indefinitely, even if the starting activity ends.
  - The service stops only when it calls `stopSelf()` or another component calls `stopService()`.
  - Example: Playing music in the background. The music keeps playing even if the user navigates to other activities.
- Bound Service:
  - Bound Service is tied to the lifecycle of the activity or component that binds to them uding `bindService()`.
  - They stop automatically when the bound activity is destroyed or the connection is unbound.
  - Example: A service that provides real-time sensor data to an activity. The service stops when the activity is destroyed.

Example SoundService: Unbounded Service

Object: show a background music when you open your app.

Res → New → Android Resource File:

- File name: mysong (**NO** Capital)
- Resource type: raw

Upload your music file to folder "raw".

```
import android.media.MediaPlayer;
public class MainActivity extends AppCompatActivity {
    // declear your member variable
    MediaPlayer music;
    @Override
        protected void onCreate(Bundle savedInstanceState) {

        // MediaPlayer
        music = MediaPlayer.create(MainActivity.this, R.raw.
            junebarcarolle);
        music.start();
        }
    }
```

Service

Declare the service in the AndroidManifest.xml:

```
    <service
        android:name=".MyService"
        android:enabled="true"
        android:exported="true">
    </service>
```

Unbounded Service Example

An unbounded example:

- In the onStart() method, the service is started when the activity becomes visible.
- The onDestroy() method ensures the service is stopped when the activity is destroyed, such as when the user closes the app or navigates away from this activity.

```
@Override
protected void onStart() {
    super.onStart();
    Intent serviceIntent = new Intent(this,
   BackgroundSoundService.class);
    this.startService(serviceIntent);
}
@Override
protected void onDestroy() {
    super.onDestroy();
    Intent serviceIntent = new Intent(this,
   .BackgroundSoundService.class);
    this.stopService(serviceIntent);
```

Bound Service: Pause and Resume (Part 1)

We show how to implement a bound service that can pause and resume audio playback based on user interaction.

```
// onStartCommand to handle PAUSE and RESUME actions
@Override
public int onStartCommand(Intent intent, int flags, int
    startId) {
     // Get the action from the Intent
     String action = intent.getAction();

     if (action != null && action.equals("PAUSE")) {
          if (player.isPlaying()) {
               player.pause();  // Pause the audio
          }
     } else if (action != null && action.equals("RESUME")) {
          player.start();  // Resume audio playback
     } else {
          player.start();  // Default action: start playback
     }
     return START_STICKY;
}
```

Bound Service: Pause and Resume (Part 2)

When jumping to `MainActivity2`, you can send a pause command via Intent to pause the audio playback without stopping the service.

```
// Stop the BackgroundSoundService when MainActivity2 is
    created
Intent serviceIntent = new Intent(this,
    BackgroundSoundService.class);
serviceIntent.setAction("PAUSE");
startService(serviceIntent);
```

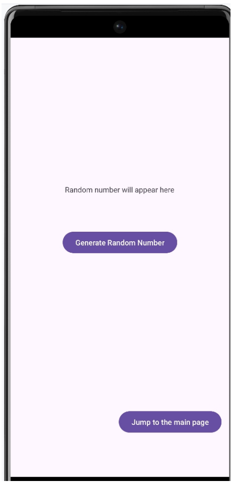
Exercise 1: Click Counter and Random Number Generator

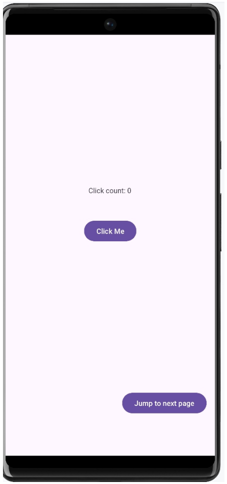
- **MainActivity**:
  - Create a `TextView` to display the number of button clicks.
  - Add a button below to increase the click count and update the `TextView`.
  - Add another button labeled "Jump to Next Page" in the bottom right to navigate to the second page (SecondActivity).
- **SecondActivity**:
  - Create a `TextView` to display a random number less than 6, generated every time a button is clicked.
  - Include a button in the bottom right to return to the first page.

Exercise 2: Adding Background Music with Service

- **Background Music Service**:
  - Implement a `Service` to play background music in MainActivity.
  - Pause the music when navigating to SecondActivity, and resume when returning to MainActivity.

Exercise


Random number will appear here
Generate Random Number
Jump to the main page


Click count: 0
Click Me
Jump to next page