Android Activity

Android Activity

- Android Activity is an essential component of Android app development that represents a single screen with a user interface.
- Activities are the building blocks of Android applications and facilitate the interaction between users and the app.
- Each activity has its lifecycle, allowing developers to manage various states and events efficiently.
- Understanding the activity lifecycle is crucial for proper resource management and maintaining a smooth user experience.

Android Activity

XML Layouts

- Activities use XML layout files to define the user interface components.
- XML layout files are located in the res/layout folder and can be edited using the Android Studio Layout Editor.

Activity Class

- To create an activity, a Java or Kotlin class needs to be defined that extends AppCompatActivity
- This class is responsible for handling the logic and behaviour of the activity.

```java
public class MainActivity extends AppCompatActivity {

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

        // You can add additional code here if needed.
    }
}
```

Android Manifest

- The AndroidManifest.xml file contains essential information about the app and its components.
- All activities need to be declared in the manifest file for the system to recognise them.
- The <intent-filter> with android.intent.action.MAIN and android.intent.category.LAUNCHER inside the activity declaration specifies that this activity should be treated as the main entry point (Launcher) for the app.

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name"
    android:theme="@style/AppTheme.NoActionBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

The activity lifecycle

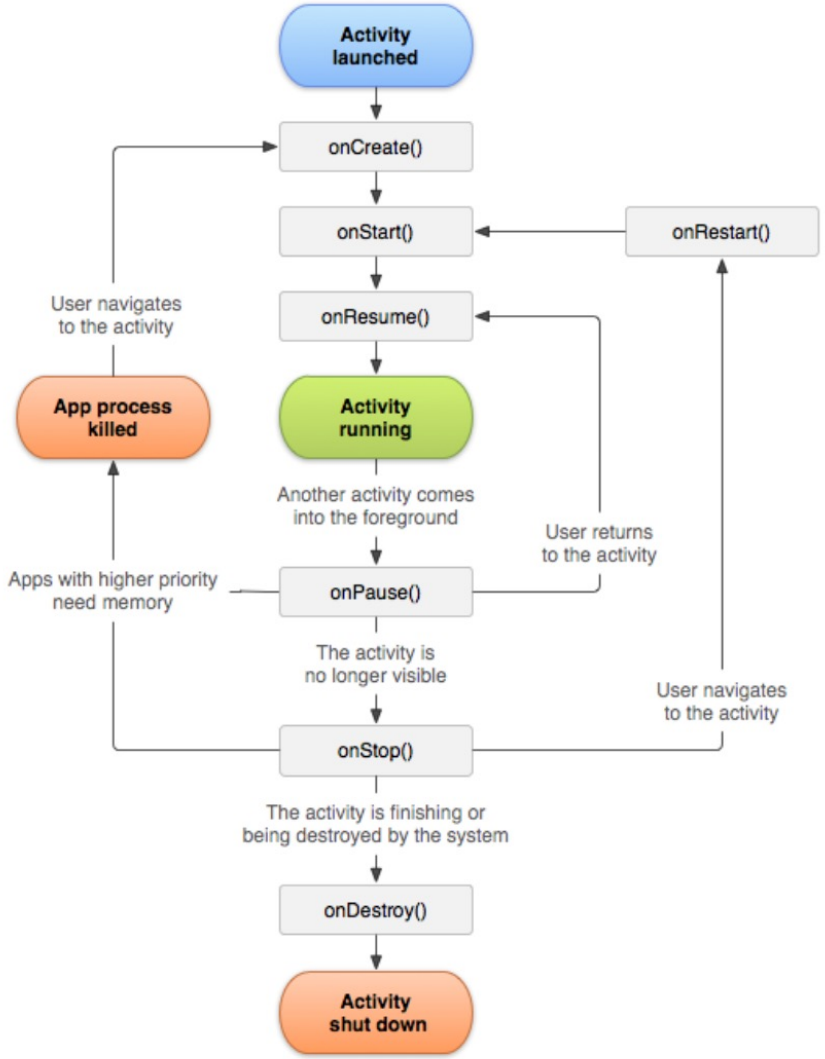


Figure 1. A simplified illustration of the activity lifecycle.

- As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle.
- The Activity class provides a number of callbacks that let the activity know when a state changes or that the system is creating, stopping, or resuming an activity or destroying the process the activity resides in.
- Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.
- For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and let the user resume the video from the same spot.

https://developer.android.com/guide/components/activities/activity-lifecycle#java

onCreate() call back

```
public class MainActivity extends AppCompatActivity {

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

        // You can add additional code here if needed.
    }
}
```

- You must implement this callback, which fires when the system first creates the activity. On activity creation, the activity enters the Created state. In the onCreate() method, perform basic application startup logic that happens only once for the entire life of the activity.
- This method receives the parameter savedInstanceState, which is a Bundle object containing the activity's previously saved state. If the activity has never existed before, the value of the Bundle object is null.

Bundle Object

A Bundle object in Android is a key-value store designed to hold and pass data between components, especially when working with activities and fragments.

```
Bundle bundle = new Bundle();
bundle.putString("key_string", "Hello, Bundle!");
bundle.putInt("key_int", 42);
bundle.putBoolean("key_bool", true);
```

A Bundle is essentially a collection of key-value pairs, where the keys are strings and the values can be various data types, such as strings, integers, booleans, Parcelable objects, and even other Bundle objects.

onSaveInstanceState() call back

- The onSaveInstanceState() method is a crucial part of the Android Activity lifecycle that allows you to save temporary UI state data in case the activity is destroyed and then recreated due to configuration changes (like screen orientation changes) or system constraints (like low memory situations).
- It's typically used to preserve data that might be lost when the activity is recreated.

onSaveInstanceState() – why?

Rotating the screen - example


```java
int i = 0;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button button = findViewById(R.id.button);
    TextView textView = findViewById(R.id.textView);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            i = i+1;
            textView.setText(String.valueOf(i));
        }
    });
}
```

onSaveInstanceState() – saving the data and retrieve

Override the method in your activity

```
public class MainActivity extends AppCompatActivity {
    2 usages
    Button button;
    4 usages
    TextView textView;
    3 usages
    int i;
    2 usages
    private static final String COUNTER ="com.example.counter";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = findViewById(R.id.button);
        textView = findViewById(R.id.textView);

        if(savedInstanceState!=null) {
            String value = savedInstanceState.getString(COUNTER);
            textView.setText(value);
        }
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                i = i+1;
                textView.setText(String.valueOf(i));
            }
        });
    }

    @Override
    public void onSaveInstanceState(@NonNull Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putString(COUNTER,textView.getText().toString());
    }
}
```

Intent

- An Intent is a fundamental component that facilitates communication between different components of an application or between different applications.
- It's a message object that is used to request an action, launch activities, and pass data between components.
- An **explicit intent** is used to specify the target component (usually an activity) that should be launched. You explicitly mention the component's class name or its action; optionally, you can include data.

```
Intent explicitIntent = new Intent(context, TargetActivity.class);
explicitIntent.putExtra("key", "value"); // Optional data
startActivity(explicitIntent);
```

Context

Context is a fundamental class that provides information about the current state and environment of an application.

The Context class itself is an abstract class and serves as the base class for several concrete implementations, such as Activity, Service, Application, and ApplicationContext.

When starting an activity, you generally use an explicit Intent that specifies the target activity class. This ensures that you're directly launching the intended activity.

Starting an activity from within an activity

- We use explicit intent

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button button = findViewById(R.id.menuButton);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            startSavingStateActivity();
        }
    });
}
1 usage
private void startSavingStateActivity(){
    Intent intent = new Intent( packageContext: this, SavingStateActivity.class);
    startActivity(intent);
}
```

Activity to activity Communication

- Sending data using Intent
- To pass data to the new activity, add some “extras” to the intent object.

Sending basic types

```
Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
intent.putExtra("key_string", "Hello, world!");
intent.putExtra("key_int", 42);
intent.putExtra("key_boolean", true);
startActivity(intent);
```

Sending Bundle using Intent

```
Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
Bundle bundle = new Bundle();
bundle.putString("key_string", "Hello, world!");
bundle.putInt("key_int", 42);
intent.putExtra("key_bundle", bundle);
startActivity(intent);
```

Activity to activity Communication (Cont..)

- Extracting the data sent by other activity. In the target component (activity, service, receiver), you retrieve the data using the appropriate method based on the data type you sent

Retrieving basic types

```
String stringValue = getIntent().getStringExtra("key_string");
int intValue = getIntent().getIntExtra("key_int", defaultValue);
boolean boolValue = getIntent().getBooleanExtra("key_boolean", defaultValue);
```

Retrieving bundle type

```
Bundle receivedBundle = getIntent().getBundleExtra("key_bundle");
String stringValue = receivedBundle.getString("key_string");
int intValue = receivedBundle.getInt("key_int");
```

Activity to activity Communication- good practice

Sharing data keys between activities is not good!!

You may make a spelling mistake on the keys, but the compiler could not detect it!!

Solution – Java Static Final - you create a variable that is global to the class and impossible to change.

Good practice example

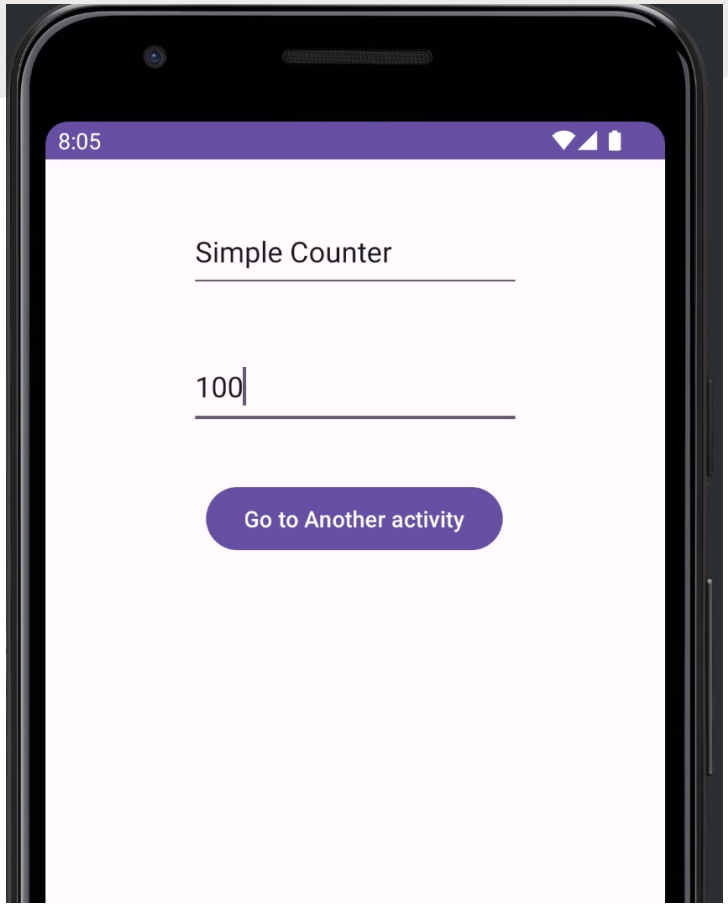
The target activity defines the Intent function and Keys that will be used by the calling activity.

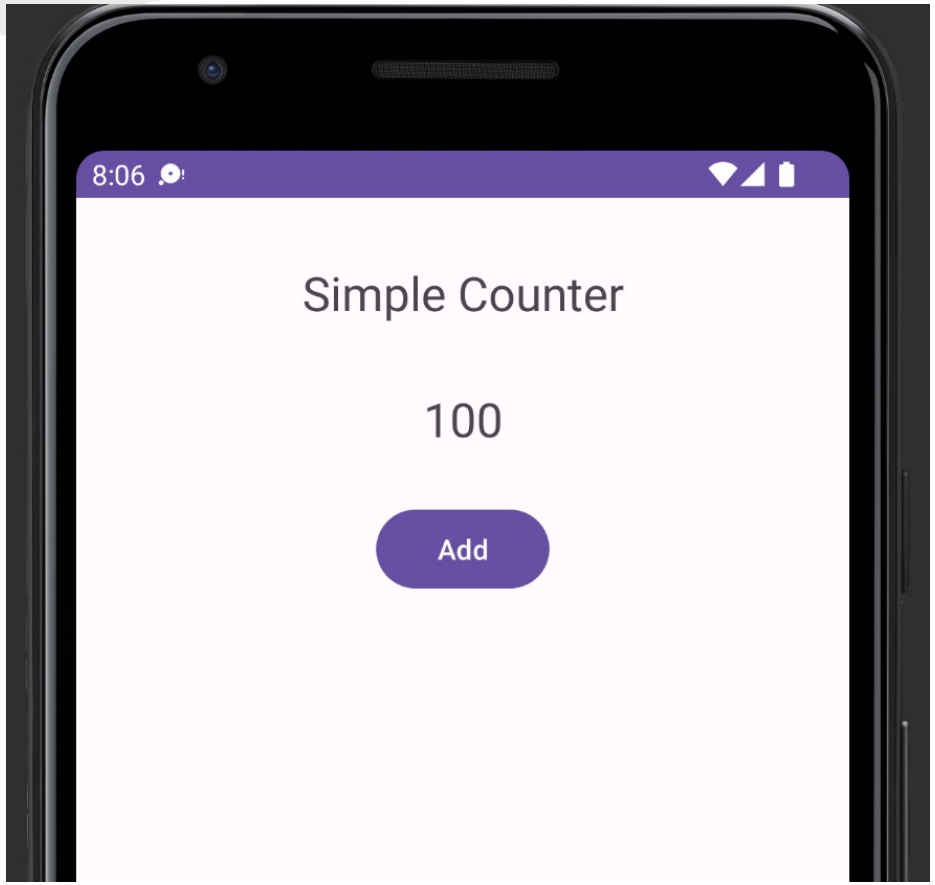
```
private static final String COUNTER ="com.example.counter";
4 usages
private static final String TITLE = "com.example.title";
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_saving_state);
    button = findViewById(R.id.button);
    counterView = findViewById(R.id.textView);
    titleView = findViewById(R.id.title);
    updateView();
    if(savedInstanceState!=null) {...}
    button.setOnClickListener(new View.OnClickListener() {...});
}
1 usage
private void updateView(){
    Intent intent = getIntent();
    String title = intent.getStringExtra(TITLE);
    if(title!=null){
        titleView.setText(title);
    }
    i = intent.getIntExtra(COUNTER, defaultValue: 0);
    counterView.setText(String.valueOf(i));
}
1 usage
public static Intent getSavingStateActivityIntent(Context c, String title, int counter){
    Intent intent = new Intent(c, SavingStateActivity.class);
    intent.putExtra(TITLE,title);
    intent.putExtra(COUNTER,counter);
    return intent;
}
```

Good practice example – Calling activity

```
public class MainActivity extends AppCompatActivity {
    Button button;
    2 usages
    EditText title;
    2 usages
    EditText counter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = findViewById(R.id.menuButton);
        title = findViewById(R.id.title);
        counter = findViewById(R.id.startingValue);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                startSavingStateActivity();
            }
        });
    }
    1 usage
    private void startSavingStateActivity(){
        String inputTitle = title.getText().toString();
        int inputCounter = Integer.parseInt(counter.getText().toString());
        Intent intent = SavingStateActivity.getSavingStateActivityIntent( c: this,inputTitle, inputCounter);
        startActivity(intent);
    }
}
```

How the demo looks





Passing data between activities - Singleton

- The Singleton pattern ensures that only one class instance exists, allowing you to store and retrieve data consistently across different activities.
- Create a Singleton class to hold the data you want to pass between activities. This class will have a private constructor, a private instance of itself, and a method to access that instance.

```
public class MyDataManager {
    3 usages
    private static MyDataManager instance;
    2 usages
    private String title;
    2 usages
    private int counter;
    public static MyDataManager getInstance(){
        if(instance==null){
            instance = new MyDataManager();
        }
        return instance;
    }
    public String getTitle(){
        return title;
    }
    public void setTitle(String title){
        this.title = title;
    }
    public int getCounter(){
        return counter;
    }
    public void setCounter(int counter){
        this.counter = counter;
    }
}
```

Passing data between activities – Singleton (cont..)

- Setting data in the calling activity

```
singletonButton.setOnClickListener(new View.OnClickListener(){

    @Override
    public void onClick(View view) {
        startActivityUsingSingleTon();
    }
});
```

```
1 usage
private void startActivityUsingSingleTon(){
    MyDataManager myDataManager = MyDataManager.getInstance();
    myDataManager.setTitle(title.getText().toString());
    myDataManager.setCounter(Integer.parseInt(counter.getText().toString()));
    Intent intent = new Intent( packageContext: this, SavingStateActivity.class);
    startActivity(intent);
}
```

Passing data between activities – Singleton (cont..)

Getting data in the target activity – calling the function in the onCreate() method.

```
private void updateViewWithSingleTon(){
    MyDataManager myDataManager = MyDataManager.getInstance();
    titleView.setText(myDataManager.getTitle());
    i = myDataManager.getCounter();
    counterView.setText(String.valueOf(i));
}
```

Caution: Singleton for passing data between activities can lead to potential memory leaks if not managed properly. You should clean up data when the data is no longer needed.

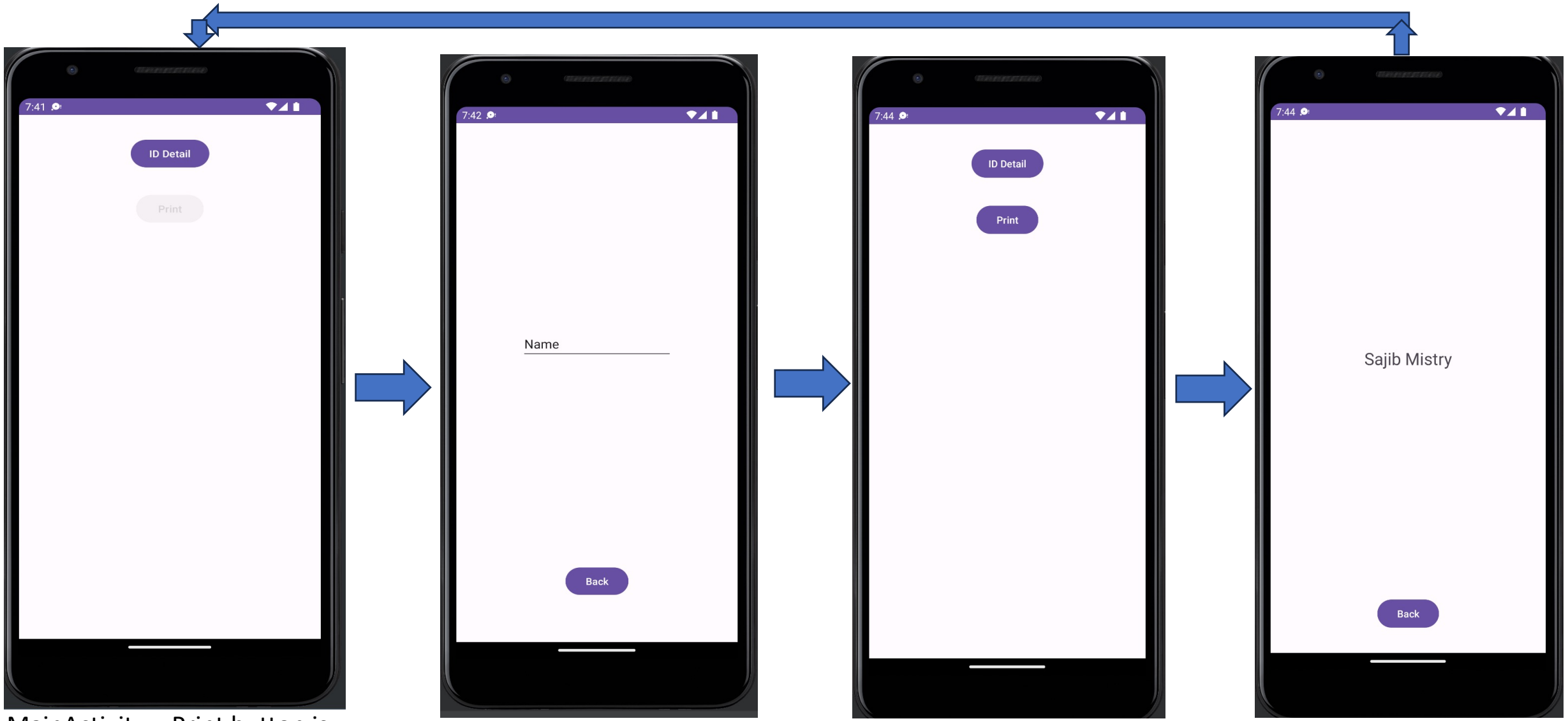
Sending Back Results

- Often a new activity must return a result to its caller.
- For example, when performing authentication or asking for permissions in a separate activity, the result can indicate whether the authentication was successful or whether the requested permissions were granted.
- Another example you may want to get the images taken from the camera app.
- The result has two parts:

1. A “result code”, is generally one of the integer constants RESULT_OK or RESULT_CANCELLED.

2. An Intent object, which can contain arbitrary “extras” data.

Sending back results – example use case



MainActivity – Print button is disabled

DetailActivity

MainActivity – Print button is enabled

PrintActivity-Restarting MainActivity

ActivityResultLauncher – key concepts

- **ActivityResultLauncher** is used for managing the process of starting an activity for a result and receiving the result data once the activity finishes.
- **registerForActivityResult()** takes an ActivityResultContract and an ActivityResultCallback and returns an ActivityResultLauncher which you'll use to launch the other activity.
- **ActivityResultContract** provides a set of pre-defined contract classes that define the input and output types for various common operations involving starting activities for results, requesting permissions, taking pictures, and more.

Predefined contracts in ActivityResultContracts

StartActivityForResult: This contract is used for starting activities and receiving results. It takes an Intent as input and provides an ActivityResult as output.

RequestPermission: This contract is used to request permissions from the user. It takes a permission name as input and provides a Boolean (indicating whether the permission was granted or not) as output.

TakePicture: This contract is used for taking pictures using the device's camera. It takes no input and provides a Uri representing the captured image as output.

GetContent: This contract is used for picking a piece of content, such as an image or a document, from external sources like the gallery or file picker. It takes a content type as input and provides a Uri representing the selected content as output.

Start Another activity for the result

Step 1: Define an ActivityResultLauncher with a callback to handle the result

```
String name="";
1 usage
ActivityResultLauncher<Intent> detailActivityLauncher = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        result -> {
            if(result.getResultCode()==RESULT_OK){
                Intent intent = result.getData();
                name = intent.getStringExtra( name: "NAME");
                detailActivityChecked =1;
                printButton.setAlpha(1.0f);
                printButton.setEnabled(true);
            }
        }
);
```

Start Another activity for the result (cont..)

Step 2: Instead of calling the target activity using startAcitivity(), use the ActivityResultLauncher.

```
detailButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent( packageContext: MainActivity.this, DetailActi
        detailActivityLauncher.launch(intent);
    }
});
```

Sending back Result from the target activity

- The newly started activity performs its operations, and when it's done, it can call setResult() to send back a result to the calling activity.

```
backButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent =  new Intent();
        intent.putExtra( name: "NAME",nameEditText.getText().toString());
        setResult(RESULT_OK,intent);
        finish();
    }
});
```

- Target activity can also finish without result.

```
public void finishWithoutResult() {
    setResult(RESULT_CANCELED);
    finish();
}
```

Check the source attached to this lecture.

Next week – Fragments.