Activity

CS443 — Mobile Applications
Instructor: Bo Sheng

Overview

• An Activity provides a screen with which users can interact.
  – Each activity is given a window to draw its UI.
  – The window may be smaller than the screen and float on top of other windows.

Overview

• An Activity provides a screen with which users can interact.
  – Each activity is given a window to draw its UI.
  – The window may be smaller than the screen and float on top of other windows.

Overview

• An app usually has multiple activities.
  – Each time a new activity starts, the previous activity is stopped, and the system preserves the activity in a stack (the "activity stack").
  – When the user presses the BACK key, it is popped from the stack (destroyed) and the previous activity resumes.

Overview

• An app usually has multiple activities.
  – Each time a new activity starts, the previous activity is stopped, and the system preserves the activity in a stack (the "activity stack").
  – When the user presses the BACK key, it is popped from the stack (destroyed) and the previous activity resumes.

• When the current activity starts another, the new activity is pushed onto the stack and takes focus.
  – The previous activity remains in the stack as stopped.
• If continuously press BACK, then each activity in the stack is popped off to reveal the previous one, until returning to the Home screen.
• When all activities are removed from the stack, the task no longer exists.
Create an Activity — Create a Class

• Create a subclass of Activity (or an existing subclass of it) and implement callback methods that the system calls when the activity transitions between various states of its lifecycle.

```java
public class MyActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

• `onCreate()`
  - This method must be implemented.
  - The system calls this when creating your activity.
  - Initialize the essential components of your activity.
  - `setContentView()`: define the layout for the activity’s user interface.

• `onPause()`
  - The system calls this method when the user is leaving your activity.
  - This is usually where you should commit any changes that should be persisted beyond the current user session.
  - E.g., draft email
  - Release system resources such as camera, GPS, and etc.

• Other calls
  - Stop, resume, destroy, ...

Important Activity States

• Resumed
  - The activity is in the foreground of the screen and has user focus.
  - Also referred to as “running”.

•Paused
  - Another activity is in the foreground and has focus, but this one is still visible.
  - A paused activity is completely alive (the Activity object is in memory), but can be killed by the system in extremely low memory situations.
  - The paused activity does not receive user input and cannot execute any code.

•Stopped
  - The activity is completely hidden (in the "background").
  - A stopped activity is also still alive (the Activity object is in memory, maintains all state and member information, but is not attached to the window manager).
  - It can be killed by the system when memory is needed elsewhere.

Implement the Lifecycle Callbacks

• The lifecycle of activities is managed by implementing callback methods.

• When an activity transitions into and out of the different states, it is notified through various callback methods.

• All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes.

Fundamental Lifecycle Methods

```java
public class MyActivity extends Activity {
    public void onResume() {
        super.onResume();
        // The activity is now "visible".
        // Further actions can be performed.
    }
}
```
Shutdown an Activity

- Shut down an activity by calling its `finish()` method.
- You can also shut down another previously started activity by calling `finishActivity()`.
- In most cases, you should not explicitly finish an activity using these methods.
  - The Android system manages the life of an activity for you, so you do not need to finish your own activities.

Create an Activity — Declare It (I)

- To declare an activity, open the manifest file and add an `<activity>` element as a child of the `<application>` element.
  - The `android:name` attribute is the only required attribute.

Create an Activity — Declare It (II)

- An `<activity>` element can specify various intent filters — using the `<intent-filter>` element—to declare how other app components may activate it.
  - Only one activity should have the “MAIN” action and “LAUNCHER” category.
  - Activities internal to the app should have no intent filters.
    - You start them using explicit intents.

Intents

- There are two main kinds of Intents:
  - **Implicit Intents**
    - Specify “what” they should match, using actions, categories, data, MIME types, and so on.
    - The exact components that they will find are only determined at runtime by the Package Manager matching it against the current applications.
  - **Explicit Intents**
    - Specify “who” they should match, through a ComponentName.
    - It is only associated with the exact manifest package name and class name as given in its ComponentName.

Start an Activity

- Start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity to be started.
  - The intent specifies either the exact activity you want to start (explicit intent)
  - or describes the type of action you want to perform (implicit intent).
  - An intent can also carry small amounts of data to be used by the activity that is started.

Intent Setting

- **Action**
  - `Intent myintent=new Intent(Intent.ACTION_DIAL)`
  - `myintent.setAction(Intent.ACTION_DIAL);`

- **Data associate with an intent (arguments)**
  - Formatted as a Uniform Resource Identifier (URI)
  - `Intent myintent=new Intent(Intent.ACTION_DIAL, Uri.parse("tel:+11234567890"));`
  - `myintent.setData(Uri.parse("tel:+11234567890"));`
Start an Activity

- Start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity to be started.

  - Implicit Intent

  ```java
  Intent intent = new Intent(Intent.ACTION_SEND);
  // Always use string resources for UI text
  String title = getResources().getString(R.string.chooser_title);
  Intent chooser = Intent.createChooser(intent, title);
  // If the activity was able to resolve...
  if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
  }
  ```

- Multiple intent receivers

Using Log Class for Debugging

- Log is a logging class that you can use to print out messages to the LogCat. Common methods include:
  - v(String tag, String msg) (verbose)
  - d(String tag, String msg) (debug)
  - i(String tag, String msg) (information)
  - w(String tag, String msg) (warning)
  - e(String tag, String msg) (error)

- Examples
  - Log.v("MyActivity", "MyClass.getView() — get item number " + position);
  - LogCat will output something like
    "MyActivity(1157): MyClass.getView() — get item number 1"

Outline

- Activity state and task
- Fragments
• onPause(), onStop(), and onDestroy() are "killable," i.e., the system can kill the activity process at any time after the method returns.

• onDestroy() is the last method guaranteed to be called before the process can be killed—if the system must recover memory in an emergency, then onPause() and onStop() might not be called.

• Use onPause() to write crucial persistent data (such as user edits) to storage.
  — Be cautious on what information must be retained during onPause(), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Saving Activity State

• The system uses the Bundle instance state
  — View objects are automatically saved

• You can preserve additional important state information by implementing onSaveInstanceState(), which allows you to restore the state when the system recreates the activity.

• The system calls the method before destroying the activity and passes it a Bundle object which will be passed to both onRestoreInstanceState() and onCreate() methods.

• Key-value data format
  — putString(key, value), putInt(key, value), …
  — getString(key), getInt(key), …

Handle Configuration Changes

• When a device changes its configuration during runtime (e.g., screen orientation), Android will restart the running Activity (onDestroy() is called, followed immediately by onCreate()).

• The restart behavior is to help an app adapt to new configurations by automatically reloading the app with alternative resources that you've provided.

• The best way to handle a configuration change is to preserve the state of your application using onSaveInstanceState() and onRestoreInstanceState() (or onCreate()).
Coordinate Activities

- When an activity starts another, the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

- The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other.

- The order of operations that occur when Activity A starts Activity B is:
  - Activity A's `onPause()` method executes.
  - Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
  - Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

Tasks and Back Stack

- A task is a collection of activities that users interact with when performing a certain job.
  - The activities may be from different applications.
  - The activities are arranged in a stack (the "back stack"), in the order in which each activity is opened.

- Home screen is the starting place for most tasks.
  - When the user touches an launcher icon, that application's task comes to the foreground.
  - If no task exists for the app (the app has not been used recently), then a new task is created and the "main" activity for that app opens as the root activity in the stack.

Multiple Tasks

- When a user leaves a task by pressing the HOME key, the current activity is stopped and its task goes into the background.
- The system retains the state of every activity in the task.
  - You should proactively retain the state of your activities in case the activity is destroyed and must be recreated.
- If the user resumes the task by selecting the launcher icon that began the task, the task comes to the foreground and resumes the activity at the top of the stack.
- Multiple tasks can be held in the background.

Manage Tasks

- <activity> attributes
  - `taskAffinity`<ref reference="taskAffinity">.
  - `launchMode`
  - `allowTaskReparenting`
  - `clearTaskOnLaunch`
  - `alwaysRetainTaskState`
  - `finishOnTaskLaunch`

- Intent flags
  - `FLAG_ACTIVITY_NEW_TASK`
  - `FLAG_ACTIVITY_CLEAR_TOP`
  - `FLAG_ACTIVITY_SINGLE_TOP`

Define Launch Mode

- Launch modes allow you to define how a new instance of an activity is associated with the current task. You can define launch modes:
  - Using the `launchMode` attribute in the `activity` manifest element and with flags in the intent that you pass to `startActivity()`.
  - You can change task management with attributes in the `activity` manifest element and with flags in the intent that you pass to `startActivity()`.
- However, most applications should not interrupt the default behavior for activities and tasks.

In some cases, you want to interrupt the task management by Android:
- You want an activity in your application to begin a new task when it is started (instead of being placed within the current task);
- When you start an activity, you want to bring forward an existing instance of it (instead of creating a new instance on top of the back stack);
- You want your back stack to be cleared of all activities except for the root activity when the user leaves the task.

Some launch modes are not available as flags, and vice versa.

If both activities define how Activity B should associate, then Activity A's request (as defined in the intent) is honored over Activity B's request (as defined in its manifest).
Using the Manifest File

• The `launchMode` attribute specifies how the activity should be launched into a task. Four different launch modes can be assigned:
  - "standard" (the default mode): The system creates a new instance in the task from which it was started and routes the intent to it.
  - "singleTop": If an instance of the activity already exists at the top of the current task, the system routes the intent to that instance rather than creating a new instance.
  - "singleTask": The system creates a new task and instantiates the activity at the root of the new task.
    - However, if an instance already exists in a separate task, the system routes the intent to the existing instance, rather than creating a new instance.
  - "singleInstance": Same as "singleTask", except that the system doesn’t launch any other activities into the task holding the instance.
    - The activity is always the single and only member of its task; any activities started by this one open in a separate task.

An Example of `singleTask`

• If you start an activity from your task (Task A) that specifies the `singleTask` launch mode, that activity might have an instance in the background that belongs to a task with its own back stack (Task B).
  - In this case, when Task B is brought forward to handle a new intent, the BACK key first navigates backward through the activities in Task B before returning to the topmost activity in Task A.

Use Intent Flags

• You can modify the default association of an activity to its task by including flags in the intent that is delivered to `startActivity`.
  - `FLAG_ACTIVITY_NEW_TASK`: produces the same behavior as the "singleTask" `launchMode` value.
  - `FLAG_ACTIVITY_SINGLE_TOP`: produces the same behavior as the "singleTop" `launchMode` value.
  - `FLAG_ACTIVITY_CLEAR_TOP`: If the activity is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it are destroyed and this intent is delivered to the resumed instance of the activity (now on top), through `onNewIntent`.
    - There is no value for the `launchMode` attribute that produces this behavior.

Outline

• Activity state and task
  - Fragments

Create a Fragment Class

• Similar lifecycle with Activity class
  - Need a layout file
  - A few subclasses
    - DialogFragment, ListFragment, PreferenceFragment, and etc.
Create a Fragment Class

- Subclass of Fragment class
  - Use the onCreateView() callback to define the layout

```java
public class ArticleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflated the layout for this fragment
        View view = inflater.inflate(R.layout.article, container, false);
        // Add the fragment to the 'fragment_container' view
        return view;
    }
}
```

- Use the replace() method: allow the user to navigate back
  ```java
  getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container, newFragment).commit();
  ```

Add into an Activity

- Java code
  - Must be associated with a parent FragmentActivity
  - Apply the layout

```java
public class MainActivity extends FragmentActivity {
    // Inflater.onCreate(savedInstanceState, state) (super.onCreate(savedInstanceState, state);)

    // If defined in the layout xml file, the fragment can't be removed at runtime.
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);
    }
}
```

Add a Fragment (runtime)

- Use the FragmentManager to create a FragmentTransaction
  - add, remove, replace,
  - Must have a container View in the layout

```java
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction transaction = fragmentManager.beginTransaction();
transaction.add(R.id.fragment_container, new ArticleFragment());
transaction.commit();
```

Replace a Fragment

- Similar to adding a fragment
  - Use replace() instead of add()
  - addToBackStack() method: allow the user to navigate back

```java
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction transaction = fragmentManager.beginTransaction();
transaction.replace(R.id.fragment_container, new ArticleFragment());
transaction.addToBackStack(null);
transaction.commit();
```

Communication between Fragments

- Communication is through the host Activity
  - No fragment-to-fragment direct communication
  - Define an interface in the Fragment class and implement it within the Activity
  - The Fragment captures the interface implementation during its onAttach() lifecycle method and can then call the interface methods in order to communicate with the Activity.
Communication between Fragments

- Fragment.zip sample

Activity A

Fragment B  Fragment C

Class Activity A implements MyInterface
public void myMethod(int)

MyInterface mCallback;

public interface MyInterface
{
    public void myMethod(int);
}

Class Fragment B

MyInterface mCallback;

public interface MyInterface
{
    public void myMethod(int);
}

public class HeadlineFragment extends ListFragment {
    OnHeadlineSelectedListener mCallback;
    // Container Activity must implement this interface
    public interface OnHeadlineSelectedListener {
        public void onArticleSelected(int);
    }
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // This makes sure that the container activity has implemented
        // the callback interface. If not, it throws an exception
        if (activity instanceof OnHeadlineSelectedListener) {
            mCallback = (OnHeadlineSelectedListener) activity;
        } else {
            throw new ClassCastException(activity.toString() +
                    " must implement OnHeadlineSelectedListener");
        }
    }
    ...
}

@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    // Send the event to the host activity
    mCallback.onArticleSelected(position);
}

Communication between Fragments

- Then the fragment uses the callback to deliver the event to the parent activity
Communication between Fragments

- Deliver a message from the Activity to a Fragment
  - Capture the Fragment instance with findFragmentById()
  - Then call the fragment's public methods

```
ArticleFragment articFrag = (ArticleFragment)
  getSupportFragmentManager().findFragmentById(R.id.article_fragment);
articFrag.updateArticleView(position);
newFragment.setArguments(args);
FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
// Replace whatever is in the fragment_container view with this fragment,
// and add the transaction to the back stack so the user can navigate back
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);
// Commit the transaction
transaction.commit();
```

Project Proposal

- Project statement
  - Motivation and objectives
    - What can your app do (provide some examples if needed)?
    - Why do you want to develop such an application?
    - What kind of users will benefit from your application?
    - Is there any special requirement for using the app?
  - Itemize the features/user interface
    - Like a product manual
    - High-level design of your application (figures or diagrams are helpful)
    - Some details of each component or user interface (as much detailed as possible)