Mobile Computing

Mobile Development

Introduction to Activities

Outline

Introduction to activities

The concept of activities

Configuring the manifest

Declare intent filter

Declare permission

The activity life cycle

Managing the activity life cycle

Introduction to Activities

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.

Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

The Concept of Activities

The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically.

For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

The Activity class is designed to facilitate this paradigm. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole.

In this way, the activity serves as the entry point for an app's interaction with the user. You implement an activity as a subclass of the Activity class.

The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically.

For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the *main activity*, which is the first screen to appear when the user launches the app.

Each activity can then start another activity in order to perform different actions.

For example, the main activity in a simple e-mail app may provide the screen that shows an e-mail inbox.

From there, the main activity might launch other activities that provide screens for tasks like writing e-mails and opening individual e-mails.

To use activities in your app, you must register information about them in the app’s manifest, and you must manage activity lifecycles appropriately.

Configuring the manifest

To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element. For example:

```
<manifest ... >
 <application ... >
     <activity android:name=".ExampleActivity" />
     ...
 </application ... >
 ...
</manifest >
```

The only required attribute for this element is android:name, which specifies the class name of the activity.

You can also add attributes that define activity characteristics such as label, icon, or UI theme.

Declare Intent Filters

Intent filters are a very powerful feature of the Android platform. They provide the ability to launch an activity based not only on an *explicit* request, but also an *implicit* one.

For example, an explicit request might tell the system to “Start the Send Email activity in the Gmail app". By contrast, an implicit request tells the system to “Start a Send Email screen in any activity that can do the job."

When the system UI asks a user which app to use in performing a task, that’s an intent filter at work.

You can take advantage of this feature by declaring an <intent-filter> attribute in the <activity> element.

The definition of this element includes an <action> element and, optionally, a <category> element and/or a <data> element.

These elements combine to specify the type of intent to which your activity can respond.

For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```xml
<activity android:name=".ExampleActivity"
android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

In this example, the <action> element specifies that this activity sends data. Declaring the <category> element as DEFAULT enables the activity to receive launch requests.
The <data> element specifies the type of data that this activity can send.

The following code snippet shows how to call the activity described above:

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.setType("text/plain");
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
// Start the activity
startActivity(sendIntent);
```

Declare permissions

You can use the manifest's <activity> tag to control which apps can start a particular activity.

A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest.

If you declare a <uses-permission> element for a parent activity, each child activity must have a matching <uses-permission> element.

For example, if your app wants to use a *hypothetical* app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>
<activity android:name="...."
   android:permission=”com.google.socialapp.permission.SHARE_POST”
/>
```

Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
   <uses-permission
android:name="com.google.socialapp.permission.SHARE_POST" />
</manifest>
```

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 allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

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 allow the user to resume the video from the same spot.

In other words, each callback allows you to perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant.

For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:

Crashing if the user receives a phone call or switches to another app while using your app.

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.

Managing the Activity life cycle

Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states.

These callbacks are given below:

onCreate()

You must implement this callback, which fires when the system creates your activity.

Your implementation should initialize the essential components of your activity:

For example, your app should create views and bind data to lists here. Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.

When onCreate() finishes, the next callback is always onStart().

onStart()

As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user.

This callback contains what amounts to the activity’s final preparations for coming to the foreground and becoming interactive.

onResume()

The system invokes this callback just before the activity starts interacting with the user.

At this point, the activity is at the top of the activity stack, and captures all user input.

Most of an app’s core functionality is implemented in the onResume() method.

The onPause() callback always follows onResume().

onPause()

The system calls onPause() when the activity loses focus and enters a Paused state.
This state occurs when, for example, the user taps the Back or Recents button.
When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

An activity in the Paused state may continue to update the UI if the user is expecting the UI to update. Examples of such an activity include one showing a navigation map screen or a media player playing.
Even if such activities lose focus, the user expects their *User Interface* (UI) to continue updating.
You should **not** use onPause() to save application or user data, make network calls, or execute database transactions.

Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.

onStop()

The system calls onStop() when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity.
In all of these cases, the stopped activity is no longer visible at all.

The next callback that the system calls is either onRestart(), if the activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating.

onRestart()

The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.
This callback is always followed by onStart().

onDestroy()

The system invokes this callback before an activity is destroyed.
This callback is the final one that the activity receives.

onDestroy() is usually implemented to ensure that all of an activity’s resources are released when the activity, or the process containing it, is destroyed.

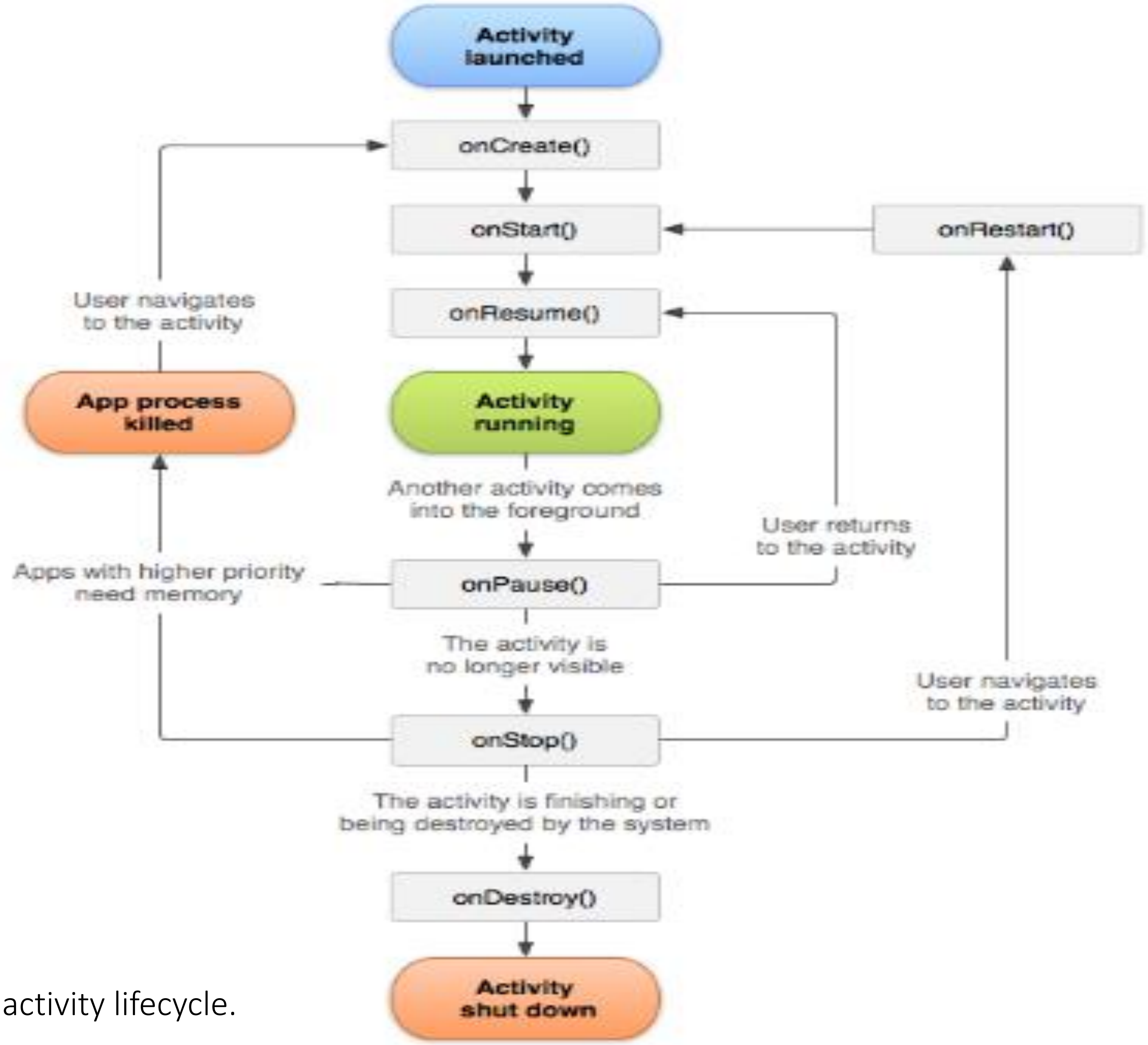

Figure 1. A simplified illustration of the activity lifecycle.

Using Mobile Device for running your app

Using Hardware devices: Android-powered device enable to develop and debug your Android applications just as you would on the emulator.

Declare your application as "debuggable" in your Android Manifest. When using Eclipse, you can skip this step

Enable **USB debugging** on your device.

In Android 4.0 and newer, it's in **Settings > Developer options**.

On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.

In Eclipse, run or debug your application as usual. You will be presented with a **Device Chooser** dialog that lists the available emulator(s) and connected device(s). Select the device upon which you want to install and run the application.

http://developer.android.com/tools/device.html

Thank you