The Activity Life Cycle

A good understanding of the Activity life cycle is vital to ensure that your application provides a seamless user experience and properly manages its resources.

As explained earlier, Android applications do not control their own process lifetimes; the Android run time manages the process of each application, and by extension that of each Activity within it.

While the run time handles the termination and management of an Activity's process, the Activity's state helps determine the priority of its parent application. The application priority, in turn, infl uences the likelihood that the run time will terminate it and the Activities running within it.

Activity Stacks

The state of each Activity is determined by its position on the Activity stack, a last-in–fi rst-out collection of all the currently running Activities. When a new Activity starts, the current foreground screen is moved to the top of the stack. If the user navigates back using the Back button, or the foreground Activity is closed, the next Activity on the stack moves up and becomes active. This process is illustrated in Figure 3-7.

As described previously in this chapter, an application's priority is infl uenced by its highest-priority Activity. The Android memory manager uses this stack to determine the priority of applications based on their Activities when deciding which application to terminate to free resources.

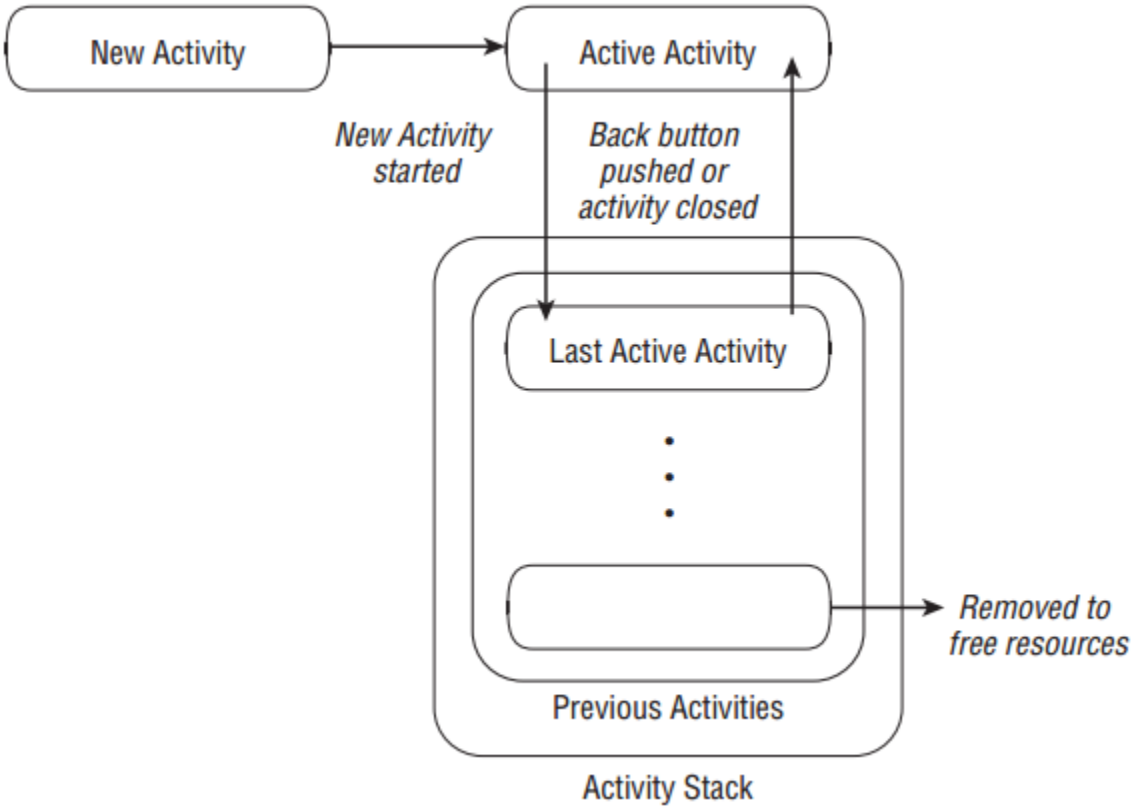

Figure 3-7

Activity States

As activities are created and destroyed, they move in and out of the stack shown in Figure 3-7. As they do so, they transition through four possible states:

❑ **Active** When an Activity is at the top of the stack, it is the visible, focused, foreground activity that is receiving user input. Android will attempt to keep it alive at all costs, killing Activities further down the stack as needed, to ensure that it has the resources it needs. When another Activity becomes active, this one will be paused.

❑ **Paused** In some cases, your Activity will be visible but will not have focus; at this point, it's paused. This state is reached if a transparent or non-full-screen Activity is active in front of it. When paused, an Activity is treated as if it were active; however, it doesn't receive user input events. In extreme cases, Android will kill a paused Activity to recover resources for the active Activity. When an Activity becomes totally obscured, it becomes stopped.

❑ **Stopped** When an Activity isn't visible, it "stops." The Activity will remain in memory retaining all state and member information; however, it is now a prime candidate for execution when the system requires memory elsewhere. When an Activity is stopped, it's important to save data and the current UI state. Once an Activity has exited or closed, it becomes inactive.

❑ **Inactive** After an Activity has been killed, and before it's been launched, it's inactive. Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used.

State transitions are nondeterministic and are handled entirely by the Android memory manager. Android will start by closing applications that contain inactive Activities, followed by those that are stopped, and in extreme cases, it will remove those that are paused.

To ensure a seamless user experience, transitions between these states should be invisible to the user. There should be no difference between an Activity moving from paused, stopped, or killed states back to active, so it's important to save all UI state changes and persist all data when an Activity is paused or stopped. Once an Activity does become active, it should restore those saved values.

Monitoring State Changes

To ensure that Activities can react to state changes, Android provides a series of event handlers that are fired when an Activity transitions through its full, visible, and active lifetimes. Figure 3-8 summarizes these lifetimes in terms of the Activity states described above.

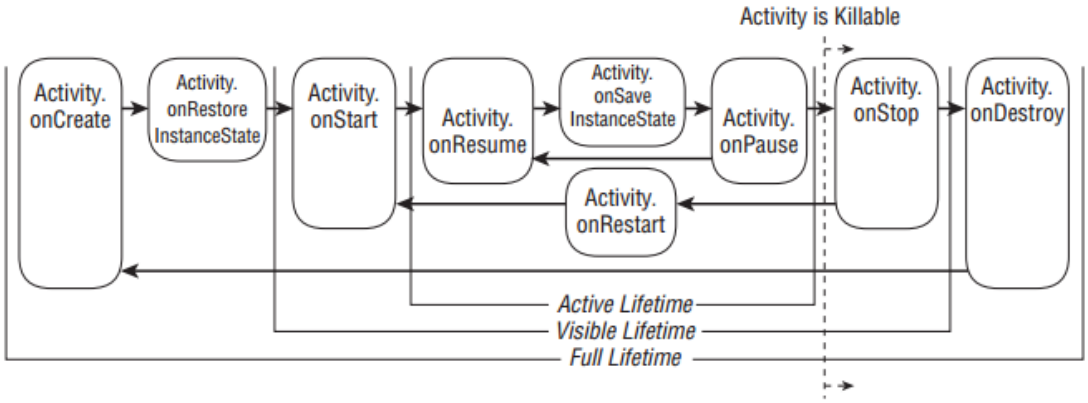

Figure 3-8

The following skeleton code shows the stubs for the state change method handlers available in an Activity. Comments within each stub describe the actions you should consider taking on each state change event.

```
package com.paad.myapplication;
import android.app.Activity;
import android.os.Bundle;
public class MyActivity extends Activity {
// Called at the start of the full lifetime.
@Override
public void onCreate(Bundle icicle) {
super.onCreate(icicle);
// Initialize activity.
}
// Called after onCreate has finished, use to restore UI state
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
super.onRestoreInstanceState(savedInstanceState);
// Restore UI state from the savedInstanceState.
```

```
// This bundle has also been passed to onCreate.
}
// Called before subsequent visible lifetimes
// for an activity process.
@Override
public void onRestart(){
super.onRestart();
// Load changes knowing that the activity has already
// been visible within this process.
}
// Called at the start of the visible lifetime.
@Override
public void onStart(){
super.onStart();
// Apply any required UI change now that the Activity is visible.
}
// Called at the start of the active lifetime.
@Override
public void onResume(){
super.onResume();
// Resume any paused UI updates, threads, or processes required
// by the activity but suspended when it was inactive.
}
// Called to save UI state changes at the
// end of the active lifecycle.
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
// Save UI state changes to the savedInstanceState.

// This bundle will be passed to onCreate if the process is
// killed and restarted.
super.onSaveInstanceState(savedInstanceState);
}
// Called at the end of the active lifetime.
@Override
public void onPause(){
// Suspend UI updates, threads, or CPU intensive processes
// that don't need to be updated when the Activity isn't
// the active foreground activity.
super.onPause();
}
// Called at the end of the visible lifetime.
@Override
public void onStop(){
// Suspend remaining UI updates, threads, or processing
// that aren't required when the Activity isn't visible.
// Persist all edits or state changes
// as after this call the process is likely to be killed.
super.onStop();
}
// Called at the end of the full lifetime.
@Override
public void onDestroy(){
// Clean up any resources including ending threads,
// closing database connections etc.
super.onDestroy();
}
}
```

As shown in the snippet above, you should always call back to the superclass when overriding these event handlers.

Understanding Activity Lifetimes

Within an Activity's full lifetime, between creation and destruction, it will go through one or more iterations of the active and visible lifetimes. Each transition will trigger the method handlers described previously. The following sections provide a closer look at each of these lifetimes and the events that bracket them.

The Full Lifetime

The full lifetime of your Activity occurs between the first call to onCreate and the final call to onDestroy. It's possible, in some cases, for an Activity's process to be terminated *without* the onDestroy method being called.

Use the onCreate method to initialize your Activity: Inflate the user interface, allocate references to class variables, bind data to controls, and create Services and threads. The onCreate method is passed a Bundle object containing the UI state saved in the last call to onSaveInstanceState. You should use this Bundle to restore the user interface to its previous state, either in the onCreate method or by overriding onRestoreInstanceStateMethod.

Override onDestroy to clean up any resources created in onCreate, and ensure that all external connections, such as network or database links, are closed.

As part of Android's guidelines for writing efficient code, it's recommended that you avoid the creation of short-term objects. Rapid creation and destruction of objects forces additional garbage collection, a process that can have a direct impact on the user experience. If your Activity creates the same set of objects regularly, consider creating them in the onCreate method instead, as it's called only once in the Activity's lifetime.

The Visible Lifetime

An Activity's visible lifetimes are bound between calls to onStart and onStop. Between these calls, your Activity will be visible to the user, although it may not have focus and might be partially obscured. Activities are likely to go through several visible lifetimes during their full lifetime, as they move between the foreground and background. While unusual, in extreme cases, the Android run time will kill an Activity during its visible lifetime without a call to onStop.

The onStop method should be used to pause or stop animations, threads, timers, Services, or other processes that are used exclusively to update the user interface. There's little value in consuming resources (such as CPU cycles or network bandwidth) to update the UI when it isn't visible. Use the onStart (or onRestart) methods to resume or restart these processes when the UI is visible again.

The onRestart method is called immediately prior to all but the first call to onStart. Use it to implement special processing that you want done only when the Activity restarts within its full lifetime.

The onStart/onStop methods are also used to register and unregister Broadcast Receivers that are being used exclusively to update the user interface. It will not always be necessary to unregister Receivers when the Activity becomes invisible, particularly if they are used to support actions other than updating the UI. You'll learn more about using Broadcast Receivers in Chapter 5.

The Active Lifetime

The active lifetime starts with a call to onResume and ends with a corresponding call to onPause. An active Activity is in the foreground and is receiving user input events. Your Activity is likely to go through several active lifetimes before it's destroyed, as the active lifetime will end when a new Activity is displayed, the device goes to sleep, or the Activity loses focus. Try to keep code in the onPause and onResume methods relatively fast and lightweight to ensure that your application remains responsive when moving in and out of the foreground.

Immediately before onPause, a call is made to onSaveInstanceState. This method provides an opportunity to save the Activity's UI state in a Bundle that will be passed to the onCreate and onRestoreInstanceState methods. Use onSaveInstanceState to save the UI state (such as check button states, user focus, and entered but uncommitted user input) to ensure that the Activity can present the same UI when it next becomes active. During the active lifetime, you can safely assume that onSaveInstanceState and onPause will be called before the process is terminated.

Most Activity implementations will override at least the onPause method to commit unsaved changes, as it marks the point beyond which an Activity may be killed without warning. Depending on your application architecture, you may also choose to suspend threads, processes, or Broadcast Receivers while your Activity is not in the foreground.

The onResume method can be very lightweight. You will not need to reload the UI state here as this is handled by the onCreate and onRestoreInstanceState methods when required. Use onResume to re-register any Broadcast Receivers or other processes you may have stopped in onPause.

Android Activity Classes

The Android SDK includes a selection of Activity subclasses that wrap up the use of common user interface widgets. Some of the more useful ones are listed below:

- **MapActivity** Encapsulates the resource handling required to support a MapView widget within an Activity. Learn more about MapActivity and MapView in Chapter 7.
- **ListActivity** Wrapper class for Activities that feature a ListView bound to a data source as the primary UI metaphor, and exposing event handlers for list item selection
- **ExpandableListActivity** Similar to the List Activity but supporting an ExpandableListView
- **ActivityGroup** Allows you to embed multiple Activities within a single screen.

Summary

In this chapter, you learned how to design robust applications using loosely coupled application components: Activities, Services, Content Providers, Intents, and Broadcast Receivers bound together using the application manifest.

You were introduced to the Android application life cycle, learning how each application's priority is determined by its process state, which is, in turn, determined by the state of the components within it. To take full advantage of the wide range of device hardware available and the international user base, you learned how to create external resources and how to defi ne alternative values for specifi c locations, languages, and hardware configurations.

Next you discovered more about Activities and their role in the application framework. As well as learning how to create new Activities, you were introduced to the Activity life cycle. In particular, you learned about Activity state transitions and how to monitor these events to ensure a seamless user experience.

Finally, you were introduced to some specialized Android Activity classes. In the next chapter, you'll learn how to create User Interfaces. Chapter 4 will demonstrate how to use layouts to design your UI before introducing some native widgets and showing you how to extend, modify, and group them to create specialized controls. You'll also learn how to create your own unique user interface elements from a blank canvas, before being introduced to the Android menu system.