Android Developers

The Activity Lifecycle

In this document

See also

As a user navigates through, out of, and back to your app, the `Activity` (https://developer.android.com/reference/android/app/Activity.html) instances in your app transition through different states in their lifecycle. The `Activity` (https://developer.android.com/reference/android/app/Activity.html) 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.

This document explains the activity lifecycle in detail. The document begins by describing the lifecycle paradigm. Next, it explains each of the callbacks: what happens internally while they execute, and what you should implement during them. It then briefly introduces the relationship between activity state and a process's vulnerability to being killed by the system. Last, it discusses several topics related to transitions between activity states.

For information about handling lifecycles, including guidance about best practices, see Handling Lifecycles (https://developer.android.com/topic/libraries/architecture/lifecycle.html).

Activity-lifecycle concepts

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)), `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()), `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()), `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()), `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()), and `onDestroy()` (https://developer.android.com/reference/android/app/Activity.html#onDestroy()). The system invokes each of these callbacks as an activity enters a new state.

Figure 1 presents a visual representation of this paradigm.

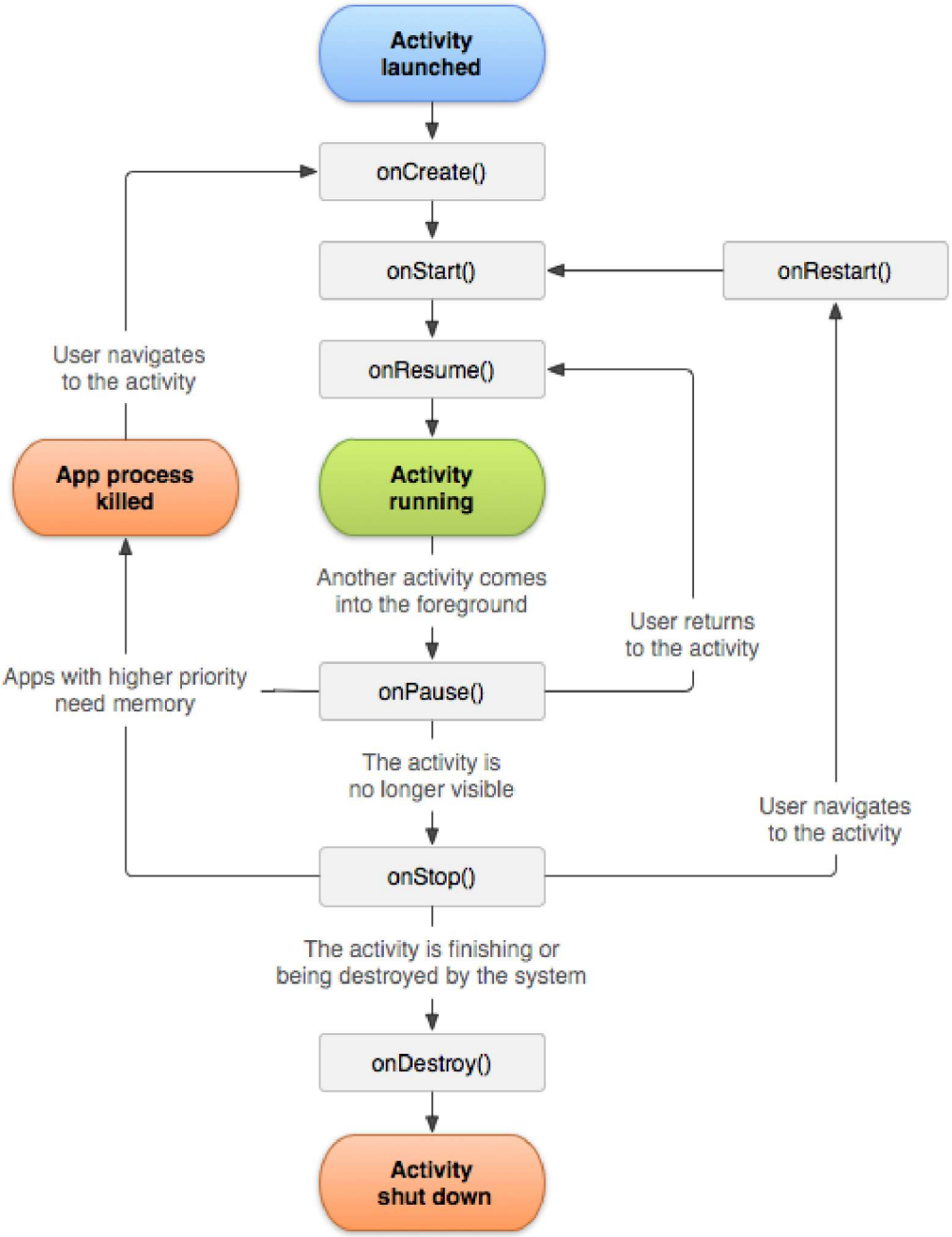

Figure 1. A simplified illustration of the activity lifecycle.

As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, this dismantlement is only partial; the activity still resides in memory (such as when the user switches to another app), and can still come back to the foreground. If the user returns to that activity, the activity resumes from where the user left off. The system's likelihood of killing a given process—along with the activities in it—depends on the state of the activity at the time. Activity state and ejection from memory (#asem) provides more information on the relationship between state and vulnerability to ejection.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

The next section of this document provides detail on the callbacks that you use to handle transitions between states.

Lifecycle callbacks

This section provides conceptual and implementation information about the callback methods used during the activity lifecycle.

onCreate()

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()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method, you perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) might bind data to lists, initialize background threads, and instantiate some class-scope variables. This method receives the parameter `savedInstanceState`, which is a `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) object containing the activity's previously saved state. If the activity has never existed before, the value of the `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) object is null.

The following example of the `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method shows fundamental setup for the activity, such as declaring the user interface (defined in an XML layout file), defining member variables, and configuring some of the UI. In this example, the XML layout file is specified by passing file's resource ID `R.layout.main_activity` to `setContentView()` (https://developer.android.com/reference/android/app/Activity.html#setContentView(android.view.View)).

```
TextView mTextView;

// some transient state for the activity instance
String mGameState;

@Override
public void onCreate(Bundle savedInstanceState) {
    // call the super class onCreate to complete the creation of activity like
    // the view hierarchy
    super.onCreate(savedInstanceState);

    // recovering the instance state
    if (savedInstanceState != null) {
        mGameState = savedInstanceState.getString(GAME_STATE_KEY);
    }
```

```
    // set the user interface layout for this Activity
    // the layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_view);
}

// This callback is called only when there is a saved instance previously saved using
// onSaveInstanceState(). We restore some state in onCreate() while we can optionally restore
// other state here, possibly usable after onStart() has completed
```



```
}

// invoked when the activity may be temporarily destroyed, save the instance state here
@Override
public void onSaveInstanceState(Bundle outState) {
    outState.putString(GAME_STATE_KEY, mGameState);
    outState.putString(TEXT_VIEW_KEY, mTextView.getText());

    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState);
}
```

As an alternative to defining the XML file and passing it to `setContentView()` (https://developer.android.com/reference/android/app/Activity.html#setContentView(android.view.View)), you can create new `View` (https://developer.android.com/reference/android/view/View.html) objects in your activity code and build a view hierarchy by inserting new `View` (https://developer.android.com/reference/android/view/View.html)s into a `ViewGroup` (https://developer.android.com/reference/android/view/ViewGroup.html). You then use that layout by passing the root `ViewGroup` (https://developer.android.com/reference/android/view/ViewGroup.html) to `setContentView()` (https://developer.android.com/reference/android/app/Activity.html#setContentView(android.view.View)). For more information about creating a user interface, see the User Interface (https://developer.android.com/guide/topics/ui/index.html) documentation.

Your activity does not reside in the Created state. After the `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method finishes execution, the activity enters the *Started* state, and the system calls the `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) and `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) methods in quick succession. The next section explains the `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) callback.

onStart()

When the activity enters the Started state, the system invokes this callback. The `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) call makes the activity visible to the user, as the

app prepares for the activity to enter the foreground and become interactive. For example, this method is where the app initializes the code that maintains the UI. It might also register a `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) that monitors changes that are reflected in the UI.

The `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state. Once this callback finishes, the activity enters the *Resumed* state, and the system invokes the `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) method.

When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) callback. This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.

When an interruptive event occurs, the activity enters the *Paused* state, and the system invokes the `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) callback.

If the activity returns to the Resumed state from the Paused state, the system once again calls `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) method. For this reason, you should implement `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) to initialize components that you release during `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()). For example, you may initialize the camera as follows:

```
@Override
public void onResume() {
    super.onResume();  // Always call the superclass method first


    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

Be aware that the system calls this method every time your activity comes into the foreground, including when it's created for the first time. As such, you should implement `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) to initialize components that you release during `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()), and perform any other

initializations that must occur each time the activity enters the Resumed state. For example, you should begin animations and initialize components that the activity only uses when it has user focus.

onPause()

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). Use the `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) method to pause operations such animations

- Some event interrupts app execution, as described in the onResume() (#onresume) section. This is the most common case.
- In Android 7.0 (API level 24) or higher, multiple apps run in multi-window mode. Because only one of the apps (windows) has focus at any time, the system pauses all of the other apps.
- A new, semi-transparent activity (such as a dialog) opens. As long as the activity is still partially visible but not in focus, it remains paused.

You can use the `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) method to release system resources, such as broadcast receivers, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.

For example, if your application uses the Camera, the `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) method is a good place to release it. The following example of `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) is the counterpart to the `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) example above, releasing the camera that the `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) example initialized.

```
@Override
public void onPause() {
    super.onPause();  // Always call the superclass method first


    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

`onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) execution is very brief, and does not necessarily afford enough time to perform save operations. For this reason, you should **not** use `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) to save application or user data, make network calls, or execute database transactions; such work may not complete before the method completes. Instead, you should perform heavy-load shutdown operations during `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()). For more information about suitable operations to perform during `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()), see onStop() (#onstop). For more information about saving data, see Saving and restoring activity state (#saras).

or becomes completely invisible to the user. If the activity resumes, the system once again invokes the `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) callback. If the activity returns from the Paused state to the Resumed state, the system keeps the `Activity` (https://developer.android.com/reference/android/app/Activity.html) instance resident in memory, recalling that instance when it the system invokes `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()). In this scenario, you don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. If the activity becomes completely invisible, the system calls `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()). The next section discusses the `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) callback.

onStop()

When your activity is no longer visible to the user, it has entered the *Stopped* state, and the system invokes the `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) callback. This may occur, for example, when a newly launched activity covers the entire screen. The system may also call `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) when the activity has finished running, and is about to be terminated.

In the `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) method, the app should release almost all resources that aren't needed while the user is not using it. For example, if you registered a `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) in `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) to listen for changes that might affect your UI, you can unregister the broadcast receiver in `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()), as the user can no longer see the UI. It is also important that you use `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) to release resources that might leak memory, because it is possible for the system to kill the process hosting your activity without calling the activity's final `onDestroy()` (https://developer.android.com/reference/android/app/Activity.html#onDestroy()) callback.

You should also use `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) to perform relatively CPU-intensive shutdown operations. For example, if you can't find a more opportune time to save information

to a database, you might do so during `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()). The following example shows an implementation of `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) that saves the contents of a draft note to persistent storage:

```
@Override
protected void onStop() {
    // call the superclass method first
    super.onStop();
```



```
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    // do this update in background on an AsyncQueryHandler or equivalent
    mAsyncQueryHandler.startUpdate (
            mToken,  // int token to correlate calls
            null,    // cookie, not used here
            mUri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
    );
}
```

When your activity enters the Stopped state, the `Activity` (https://developer.android.com/reference/android/app/Activity.html) object is kept resident in memory: It maintains all state and member information, but is not attached to the window manager. When the activity resumes, the activity recalls this information. You don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. The system also keeps track of the current state for each `View` (https://developer.android.com/reference/android/view/View.html) object in the layout, so if the user entered text into an `EditText` (https://developer.android.com/reference/android/widget/EditText.html) widget, that content is retained so you don't need to save and restore it.

> **Note:** Once your activity is stopped, the system might destroy the process that contains the activity if the system needs to recovery memory. Even if the system destroys the process while the activity is stopped, the system still retains the state of the `View` (https://developer.android.com/reference/android/view/View.html) objects (such as text in an `EditText` (https://developer.android.com/reference/android/widget/EditText.html) widget) in a `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) (a blob of key-value pairs) and restores them if the user navigates back to the activity. For more information about restoring an activity to which a user returns, see Saving and restoring activity state (#saras).

From the Stopped state, the activity either comes back to interact with the user, or the activity is finished running and goes away. If the activity comes back, the system invokes `onRestart()`

(https://developer.android.com/reference/android/app/Activity.html#onRestart()). If the Activity (https://developer.android.com/reference/android/app/Activity.html) is finished running, the system calls onDestroy() (https://developer.android.com/reference/android/app/Activity.html#onDestroy()). The next section explains the onDestroy() (https://developer.android.com/reference/android/app/Activity.html#onDestroy()) callback.

onDestroy()

Called before the activity is destroyed. This is the final call that the activity receives. The system either invokes this

isFinishing() (https://developer.android.com/reference/android/app/Activity.html#isFinishing()) method. The system may also call this method when an orientation change occurs, and then immediately call onCreate() (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) to recreate the process (and the components that it contains) in the new orientation.

The onDestroy() (https://developer.android.com/reference/android/app/Activity.html#onDestroy()) callback releases all resources that have not yet been released by earlier callbacks such as onStop() (https://developer.android.com/reference/android/app/Activity.html#onStop()).

Activity state and ejection from memory

The system never kills an activity directly. Instead, it kills the process in which the activity runs, destroying not only the activity but everything else running in the process, as well.

The system kills processes when it needs to free up RAM; the likelihood of its killing a given process depends on the state of the process at the time. Process state, in turn, depends on the state of the activity running in the process.

A user can also kill a process by using the Application Manager under Settings to kill the corresponding app.

Table 1 shows the correlation among process state, activity state, and likelihood of the system's killing the process.

Likelihood of being killed	Process state	Activity state
Least	Foreground (having or about to get focus)	Created Started Resumed
More	Background (lost focus)	Paused
Most	Background (not visible)	Stopped
	Empty	Destroyed

Table 1. Relationship between process lifecycle and activity state

For more information about processes in general, see Processes and Threads (https://developer.android.com/guide/components/processes-and-threads.html). For more information about how the lifecycle of a process is tied to the states of the activities in it, see the Process Lifecycle (https://developer.android.com/guide/components/processes-and-threads.html#Lifecycle) section of that page.

tap the device's Back button, or the activity may need to launch a different activity. This section covers topics you need to know to implement successful activity transitions. These topics include starting an activity from another activity, saving activity state, and restoring activity state.

Starting one activity from another

An activity often needs to start another activity at some point. This need arises, for instance, when an app needs to move from the current screen to a new one.

Depending on whether your activity wants a result back from the new activity it's about to start, you start the new activity using either the `startActivity()` (https://developer.android.com/reference/android/app/Activity.html#startActivity(android.content.Intent, android.os.Bundle)) or the `startActivityForResult()` (https://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int)) method. In either case, you pass in an `Intent` (https://developer.android.com/reference/android/content/Intent.html) object.

The `Intent` (https://developer.android.com/reference/android/content/Intent.html) object specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An `Intent` (https://developer.android.com/reference/android/content/Intent.html) object can also carry small amounts of data to be used by the activity that is started. For more information about the `Intent` (https://developer.android.com/reference/android/content/Intent.html) class, see Intents and Intent Filters (https://developer.android.com/guide/components/intents-filters.html).

startActivity()

If the newly started activity does not need to return a result, the current activity can start it by calling the `startActivity()` (https://developer.android.com/reference/android/app/Activity.html#startActivity(android.content.Intent, android.os.Bundle)) method.

When working within your own application, you often need to simply launch a known activity. For example, the following code snippet shows how to launch an activity called `SignInActivity`.

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

Your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable: You can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the


```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

startActivityForResult()

Sometimes you want to get a result back from an activity when it ends. For example, you may start an activity that lets the user pick a person in a list of contacts; when it ends, it returns the person that was selected. To do this, you call the `startActivityForResult(Intent, int)` (https://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int)) method, where the integer parameter identifies the call. This identifier is meant to disambiguate between multiple calls to `startActivityForResult(Intent, int)` (https://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int)) from the same activity. It's not global identifier and is not at risk of conflicting with other apps or activities.The result comes back through your `onActivityResult(int, int, Intent)` (https://developer.android.com/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent)) method.

When a child activity exits, it can call `setResult(int)` to return data to its parent. The child activity must always supply a result code, which can be the standard results `RESULT_CANCELED`, `RESULT_OK`, or any custom values starting at `RESULT_FIRST_USER`. In addition, the child activity can optionally return an `Intent` (https://developer.android.com/reference/android/content/Intent.html) object containing any additional data it wants. The parent activity uses the `onActivityResult(int, int, Intent)` (https://developer.android.com/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent)) method, along with the integer identifier the parent activity originally supplied, to receive the information.

If a child activity fails for any reason, such as crashing, the parent activity receives a result with the code `RESULT_CANCELED`.

```
public class MyActivity extends Activity {
    ...

    static final int PICK_CONTACT_REQUEST = 0;

    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
            // When the user center presses, let them pick a contact.
            startActivityForResult(
                new Intent(Intent.ACTION_PICK,
```



```
            return false;
        }

        protected void onActivityResult(int requestCode, int resultCode,
                Intent data) {
            if (requestCode == PICK_CONTACT_REQUEST) {
                if (resultCode == RESULT_OK) {
                    // A contact was picked.  Here we will just display it
                    // to the user.
                    startActivity(new Intent(Intent.ACTION_VIEW, data));
                }
            }
        }
    }
```

Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity stops operating and enters the Paused or Stopped state, while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that 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 (app) and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) method executes.
2. Activity B's `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)), `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()), and `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another.

Saving and restoring activity state

There are a few scenarios in which your activity is destroyed due to normal app behavior, such as when the user presses the Back button or your activity signals its own destruction by calling the `finish()` (https://developer.android.com/reference/android/app/Activity.html#finish()) method. The system may also destroy the

that `Activity` (https://developer.android.com/reference/android/app/Activity.html) instance is gone forever because the behavior indicates the activity is no longer needed. However, if the system destroys the activity due to system constraints (rather than normal app behavior), then although the actual `Activity` (https://developer.android.com/reference/android/app/Activity.html) instance is gone, the system remembers that it existed such that if the user navigates back to it, the system creates a new instance of the activity using a set of saved data that describes the state of the activity when it was destroyed. The saved data that the system uses to restore the previous state is called the *instance state* and is a collection of key-value pairs stored in a `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) object.

By default, the system uses the `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) instance state to save information about each `View` (https://developer.android.com/reference/android/view/View.html) object in your activity layout (such as the text value entered into an `EditText` (https://developer.android.com/reference/android/widget/EditText.html) widget). So, if your activity instance is destroyed and recreated, the state of the layout is restored to its previous state with no code required by you. However, your activity might have more state information that you'd like to restore, such as member variables that track the user's progress in the activity.

A `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) object isn't appropriate for preserving more than a trivial amount of data because it consumes system-process memory. To preserve more than a very small amount of data, you should take a combined approach to preserving data, using persistent local storage, the `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) method, and the `ViewModel` (https://developer.android.com/reference/android/arch/lifecycle/ViewModel.html) class. For more information about preserving complex data structures, see Saving UI States (https://developer.android.com/topic/libraries/architecture/saving-states.html).

For cases where a `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) is appropriate, you may use the `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) method. The next section provides detail about how to use this method.

Save your activity state

As your activity begins to stop, the system calls the `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) method so your activity can save state information with a collection of key-value pairs. The default implementation of this method saves transient information about the state of the activity's view hierarchy, such as the text in an `EditText` (https://developer.android.com/reference/android/widget/EditText.html) widget or the scroll position of a `ListView` (https://developer.android.com/reference/android/widget/ListView.html) widget. Your app should implement the

(https://developer.android.com/reference/android/app/Activity.html#onStop()). Do not implement this callback in onPause() (https://developer.android.com/reference/android/app/Activity.html#onPause()).

> **Caution:** You must always call the superclass implementation of `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) so the default implementation can save the state of the view hierarchy.

To save additional state information for your activity, you must override `onSaveInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) and add key-value pairs to the `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) object that is saved in the event that your activity is destroyed unexpectedly. For example:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...


@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);


    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

> **Note:** In order for the Android system to restore the state of the views in your activity, each view must have a unique ID, supplied by the `android:id` attribute.

To save persistent data, such as user preferences or data for a database, you should take appropriate opportunities when your activity is in the foreground. If no such opportunity arises, you should save such data during the `onStop()` (https://developer.android.com/reference/android/app/Activity.html#onStop()) method.

Restore your activity state

When your activity is recreated after it was previously destroyed, you can recover your saved state from the `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) that the system passes to your activity. Both the `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) and `onRestoreInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onRestoreInstanceState(android.os.Bundle, android.os.PersistableBundle)) callback methods receive the same `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) that contains the instance state information.

check whether the state Bundle is null before you attempt to read it. If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

For example, the following code snippet shows how you can restore some state data in `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)):

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first


    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Instead of restoring the state during `onCreate()` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) you may choose to implement `onRestoreInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onRestoreInstanceState(android.os.Bundle, android.os.PersistableBundle)), which the system calls after the `onStart()` (https://developer.android.com/reference/android/app/Activity.html#onStart()) method. The system calls `onRestoreInstanceState()` (https://developer.android.com/reference/android/app/Activity.html#onRestoreInstanceState(android.os.Bundle, android.os.PersistableBundle)) only if there is a saved state to restore, so you do not need to check whether the `Bundle` (https://developer.android.com/reference/android/os/Bundle.html) is null:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);


    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```


android.os.PersistableBundle)) so the default implementation can restore the state of the view hierarchy.