Activities

Quickview

- An activity provides a user interface for a single screen in your application
- Activities can move into the background and then be resumed with their state restored

In this document

Key classes

`Activity`

See also

Hello World Tutorial
Tasks and Back Stack

An `Activity` is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" queue mechanism, so, when the user is done with the current activity and presses the BACK key, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the Tasks and Back Stack document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

Creating an Activity

To create an activity, you must create a subclass of `Activity` (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

`onCreate()`
: You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call `setContentView()` to define the layout for the activity's user interface.

`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). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interuptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about Managing the Activity Lifecycle.

Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the `View` class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from `ViewGroup` that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the `View` and `ViewGroup` classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with `setContentView()`, passing the resource ID for the layout. However, you can also create new `View`s in your activity code and build a view hierarchy by inserting new `View`s into a `ViewGroup`, then use that layout by passing the root `ViewGroup` to `setContentView()`.

For information about creating a user interface, see the User Interface documentation.

Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To decalare 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 >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. See the `<activity>` element reference for more information about available attributes.

Using intent filters

An `<activity>` element can also specify various intent filters—using the `<intent-filter>` element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you

don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an `<intent-filter>` that includes an `<action>` element and, optionally, a `<category>` element and/or a `<data>` element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the Intents and Intent Filters document.

Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent 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 can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

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

However, 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, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
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.

Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider
URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}
```

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST) {
        // Perform a query to the contact's content provider for the contact's
name
        Cursor cursor = getContentResolver().query(data.getData(),
        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}
```

This example shows the basic logic you should use in your `onActivityResult()` method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be `RESULT_OK`—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with `startActivityForResult()`. From there, the code handles the activity result by querying the data returned in an `Intent` (the `data` parameter).

What happens is, a `ContentResolver` performs a query against a content provider, which returns a `Cursor` that allows the queried data to be read. For more information, see the Content Providers document.

For more information about using intents, see the Intents and Intent Filters document.

Shutting Down an Activity

You can shut down an activity by calling its `finish()` method. You can also shut down a separate activity that you previously started by calling `finishActivity()`.

> **Note:** In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

Resumed
: The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

Paused
: Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the `Activity` object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

Stopped
: The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the `Activity` object is retained in memory, it maintains all state and member information, but is *not*

attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its `finish()` method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, 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. The following skeleton activity includes each of the fundamental lifecycle methods:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be
"paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

> **Note:** Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.
- The **visible lifetime** of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to

monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.

- The **foreground lifetime** of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.

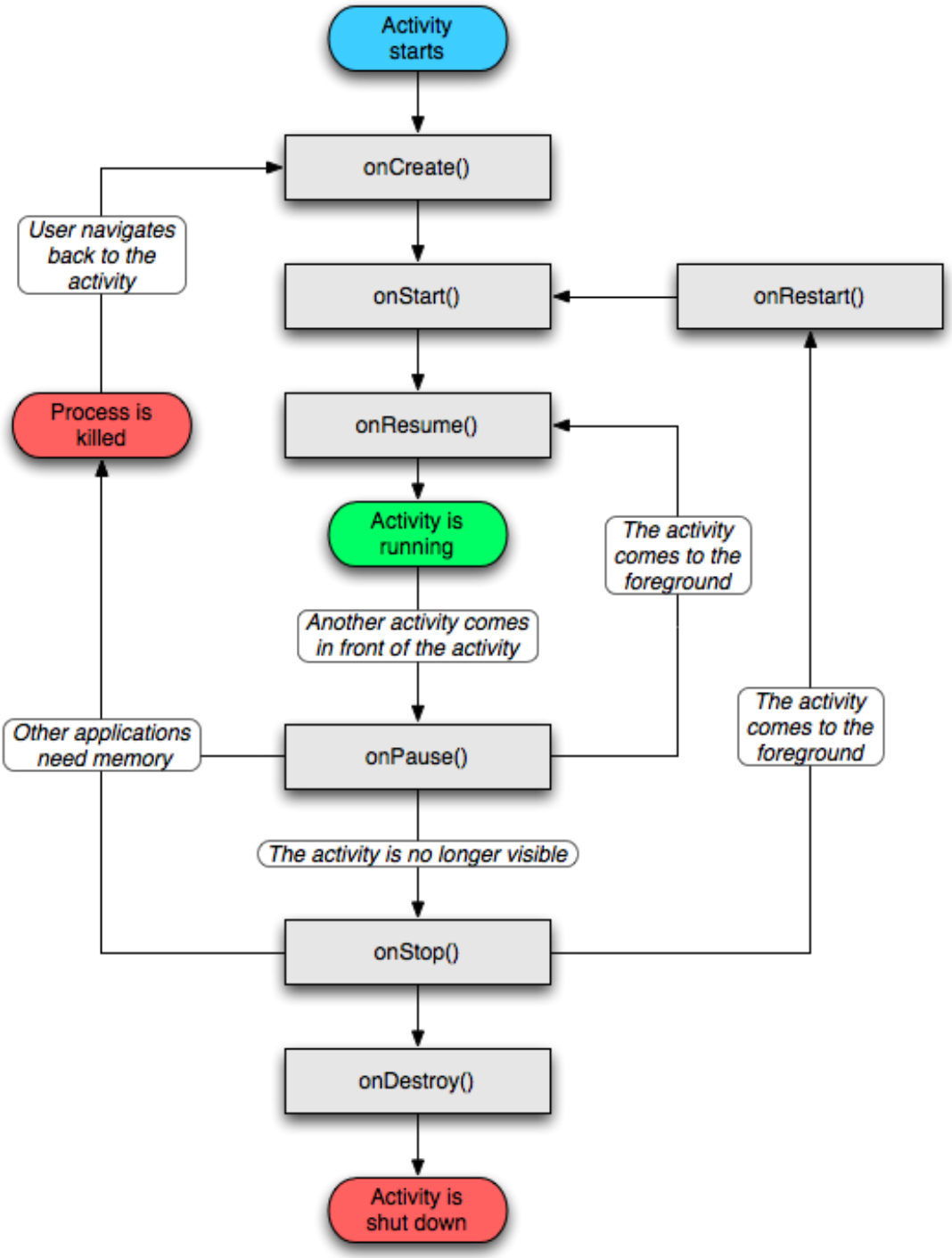

Figure 1. The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

Table 1. A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
`onCreate()`	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see Saving Activity State, later). Always followed by `onStart()`.	No	`onStart()`
`onRestart()`	Called after the activity has been stopped, just prior to it being started again. Always followed by `onStart()`	No	`onStart()`
`onStart()`	Called just before the activity becomes visible to the user. Followed by `onResume()` if the activity comes to the foreground, or `onStop()` if it becomes hidden.	No	`onResume()` or `onStop()`
`onResume()`	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it. Always followed by `onPause()`.	No	`onPause()`
`onPause()`	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns. Followed either by `onResume()` if the activity returns back to the front, or by `onStop()` if it becomes invisible to the user.	**Yes**	`onResume()` or `onStop()`
`onStop()`	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it. Followed either by `onRestart()` if the activity is coming back to interact with the user, or by `onDestroy()` if this activity is going away.	**Yes**	`onRestart()` or `onDestroy()`

onDestroy()	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called finish() on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method.	**Yes**	*nothing*

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": (onPause(), onStop(), and onDestroy()). Because onPause() is the first of the three, once the activity is created, onPause() is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then onStop() and onDestroy() might not be called. Therefore, you should use onPause() to write crucial persistent data (such as user edits) to storage. However, you should be selective about 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.

Methods that are marked "No" in the **Killable** column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time onPause() returns to the time onResume() is called. It will not again be killable until onPause() is again called and returns.

> **Note:** An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the Processes and Threading document.

Saving activity state

The introduction to Managing the Activity Lifecycle briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the Activity object is still held in memory when it is paused or stopped —all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained in memory, so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the Activity object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the Activity object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity and then restore it when the the system recreates the activity.

The callback method in which you can save information about the current state of your activity is onSaveInstanceState(). The system calls this method before making the activity vulnerable to being destroyed and passes it a Bundle object. The Bundle is where you can store state information about the activity as name-value pairs, using methods such as putString(). Then, if the system kills your activity's process and the user navigates back to your activity, the system passes the Bundle to onCreate() so you can restore the activity state you saved during onSaveInstanceState(). If there is no state information to restore, then the Bundle passed to onCreate() is null.

> **Note:** There's no guarantee that onSaveInstanceState() will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the BACK key, because the user is explicitly closing the activity). If the method is called, it is always called before onStop() and possibly before onPause().

However, even if you do nothing and do not implement onSaveInstanceState(), some of the activity state is restored by the Activity class's default implementation of onSaveInstanceState(). Specifically, the default implementation calls onSaveInstanceState() for every View in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as

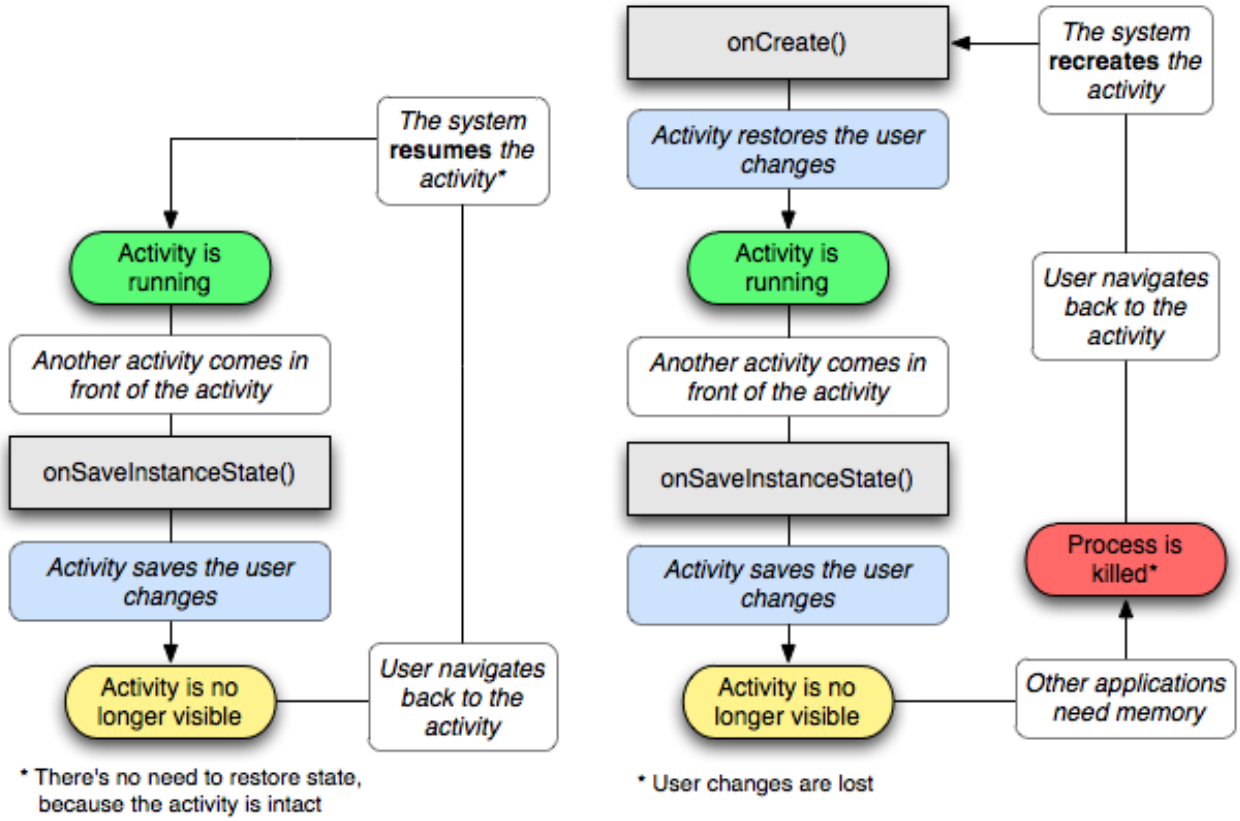

Figure 2. The two ways in which an activity returns to user focus with its state intact: either the activity is stopped, then resumed and the activity state remains intact (left), or the activity is destroyed, then recreated and the activity must restore the previous activity state (right).

appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the `EditText` widget saves any text entered by the user and the `CheckBox` widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then it cannot save its state.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

You can also explicitly stop a view in your layout from saving its state by setting the `android:saveEnabled` attribute to `"false"` or by calling the `setSaveEnabled()` method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work.

> **Note:** Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new orientation. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android restarts the running Activity (`onDestroy()` is called, followed immediately by `onCreate()`). The restart behavior is designed to help your application adapt to new configurations by automatically

reloading your application with alternative resources that you've provided. If you design your activity to properly handle this event, it will be more resilient to unexpected events in the activity lifecycle.

The best way to handle a configuration change, such as a change in the screen orientation, is to simply preserve the state of your application using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For a detailed discussion about configuration changes that happen at runtime and how you should handle them, read Handling Runtime Changes.

Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), 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 and one is starting the other. Here's the order of operations that occur when Activity A starts Acivity B:

1. Activity A's `onPause()` method executes.
2. Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.

↑ Go to top