Tasks and Back Stack

Quickview

- All activities belong to a task
- A task contains a collection of activities in the order in which the user interacts with them
- Tasks can move to the background and retain the state of each activity in order for the user to perform other tasks without loosing their work

In this document

Articles

Multitasking the Android Way

See also

Application Lifecycle video
`<activity>` manifest element

An application usually contains multiple activities. Each activity should be designed around a specific kind of action the user can perform and can start other activities. For example, an email application might have one activity to show a list of new email. When the user selects an email, a new activity opens to view that email.

An activity can even start activities that exist in other applications on the device. For example, if your application wants to send an email, you can define an intent to perform a "send" action and include some data, such as an email address and a message. An activity from another application that declares itself to handle this kind of intent then opens. In this case, the intent is to send an email, so an email application's "compose" activity starts (if multiple activities support the same intent, then the system lets the user select which one to use). When the email is sent, your activity resumes and it seems as if the email activity was part of your application. Even though the activities may be from different applications, Android maintains this seamless user experience by keeping both activities in the same *task*.

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

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

When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface. When the user presses the BACK key, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored). Activities in the stack are never rearranged, only pushed and popped from the stack—pushed onto the stack when started by the current activity and popped off when the user leaves it using the BACK key. As such, the back stack operates as a "last in, first out" object structure. Figure 1 visualizes this behavior with a timeline showing the progress between activities along with the current back stack at each point in time.

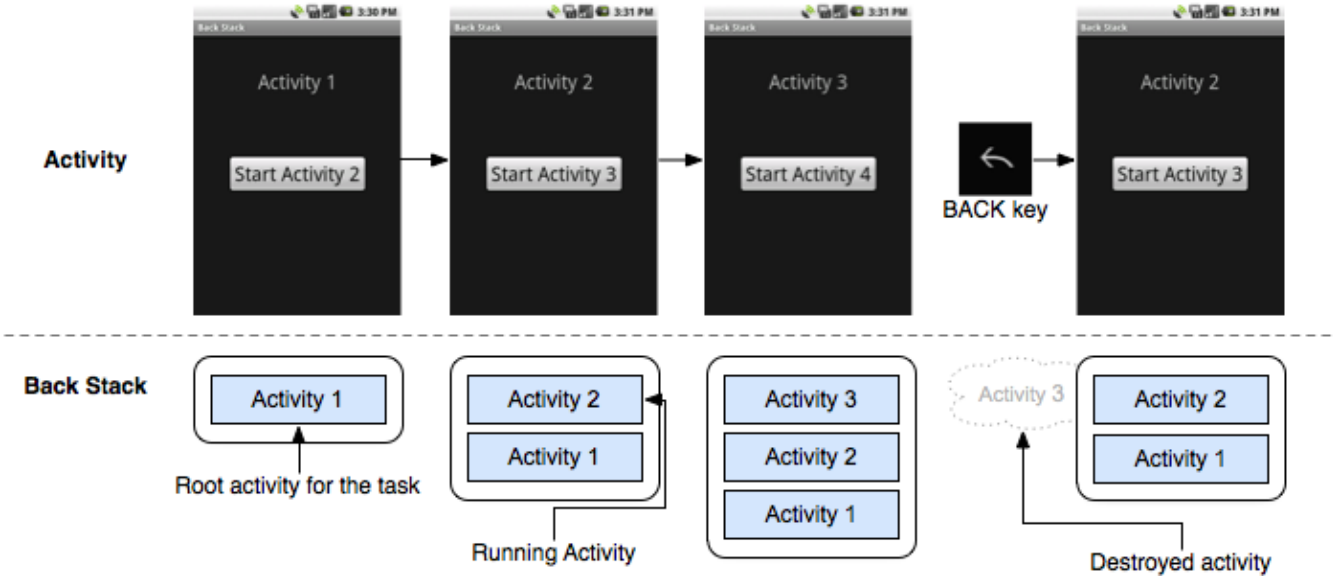

Figure 1. A representation of how each new activity in a task adds an item to the back stack. When the user presses the BACK key, the current activity is destroyed and the previous activity resumes.

If the user continues to press BACK, then each activity in the stack is popped off to reveal the previous one, until the user returns

to the Home screen (or to whichever activity was running when the task began). When all activities are removed from the stack, the task no longer exists.

A task is a cohesive unit that can move to the "background" when users begin a new task or go to the Home screen, via the HOME key. While in the background, all the activities in the task are stopped, but the back stack for the task remains intact—the task has simply lost focus while another task takes place, as shown in figure 2. A task can then return to the "foreground" so users can pick up where they left off. Suppose, for example, that the current task (Task A) has three activities in its stack—two under the current activity. The user presses the HOME key, then starts a new application from the application launcher. When the Home screen appears, Task A goes into the background. When the new application starts, the system starts a task for that application (Task B) with its own stack of activities. After interacting with that application, the user returns Home again and selects the application that originally started Task A. Now, Task A comes to the foreground—all three activities in its stack are intact and the activity at the top of the stack resumes. At this point, the user can also switch back to Task B by going Home and selecting the application icon that started that task (or by touching and holding the HOME key to reveal recent tasks and selecting one). This is an example of multitasking on Android.

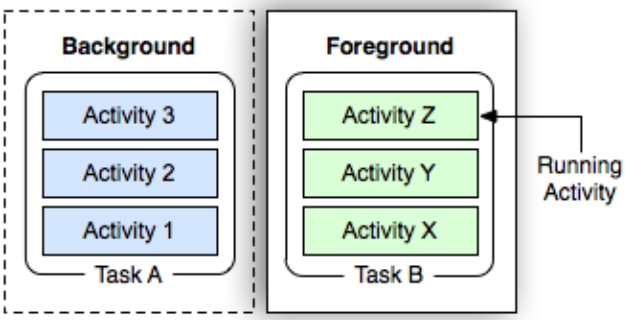

Figure 2. Two tasks: Task A is in the background, waiting to be resumed, while Task B receives user interaction in the foreground.

> **Note:** Multiple tasks can be held in the background at once. However, if the user is running many background tasks at the same time, the system might begin destroying background activities in order to recover memory, causing the activity states to be lost. See the following section about Activity state.

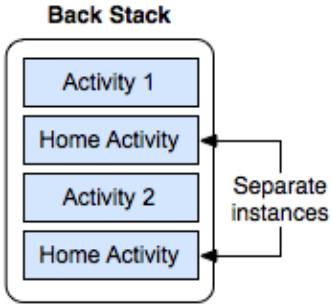

Figure 3. A single activity is instantiated multiple times.

Because the activities in the back stack are never rearranged, if your application allows users to start a particular activity from more than one activity, a new instance of that activity is created and popped onto the stack (rather than bringing any previous instance of the activity to the top). As such, one activity in your application might be instantiated multiple times (even from different tasks), as shown in figure 3. As such, if the user navigates backward using the BACK key, each instance of the activity is revealed in the order they were opened (each with their own UI state). However, you can modify this behavior if you do not want an activity to be instantiated more than once. How to do so is discussed in the later section about Managing Tasks.

To summarize the default behavior for activities and tasks:

- When Activity A starts Activity B, Activity A is stopped, but the system retains its state (such as scroll position and text entered into forms). If the user presses the BACK key while in Activity B, Activity A resumes with its state restored.
- When the user leaves a task by pressing the HOME key, the current activity is stopped and its task goes into the background. The system retains the state of every activity in the task. If the user later resumes the task by selecting the launcher icon that began the task, the task comes to the foreground and resumes the activity at the top of the stack.
- If the user presses the BACK key, the current activity is popped from the stack and destroyed. The previous activity in the stack is resumed. When an activity is destroyed, the system *does not* retain the activity's state.
- Activities can be instantiated multiple times, even from other tasks.

Saving Activity State

As discussed above, the system's default behavior preserves the state of an activity when it is stopped. This way, when users navigate back to a previous activity, its user interface appears the way they left it. However, you can—and **should**—proactively retain the state of your activities using callback methods, in case the activity is destroyed and must be recreated.

When the system stops one of your activities (such as when a new activity starts or the task moves to the background), the system might destroy that activity completely if it needs to recover system memory. When this happens, information about the activity state is lost. If this happens, the system still knows that the activity has a place in the back stack, but when the activity is brought to the top of the stack the system must recreate it (rather than resume it). In order to avoid loosing the user's work, you should proactively retain it by implementing the `onSaveInstanceState()` callback methods in your activity.

For more information about how to save your activity state, see the Activities document.

Managing Tasks

The way Android manages tasks and the back stack, as described above—by placing all activities started in succession in the same task and in a "last in, first out" stack—works great for most applications and you shouldn't have to worry about how your activities are associated with tasks or how they exist in the back stack. However, you might decide that you want to interrupt the normal behavior. Perhaps you want an activity in your application to begin a new task when it is started (instead of being placed within the current task); or, when you start an activity, you want to bring forward an existing instance of it (instead of creating a new instance on top of the back stack); or, you want your back stack to be cleared of all activitiesstart an activity except for the root activity when the user leaves the task.

You can do these things and more, with attributes in the `<activity>` manifest element and with flags in the intent that you pass to `startActivity()`.

In this regard, the the principal `<activity>` attributes you can use are:

- `taskAffinity`
- `launchMode`
- `allowTaskReparenting`
- `clearTaskOnLaunch`
- `alwaysRetainTaskState`
- `finishOnTaskLaunch`

And the principal intent flags you can use are:

- `FLAG_ACTIVITY_NEW_TASK`
- `FLAG_ACTIVITY_CLEAR_TOP`
- `FLAG_ACTIVITY_SINGLE_TOP`

In the following sections, you'll see how you can use these manifest attributes and intent flags to define how activities are associated with tasks and how the behave in the back stack.

> **Caution:** Most applications should not interrupt the default behavior for activities and tasks. If you determine that it's necessary for your activity to modify the default behaviors, use caution and be sure to test the usability of the activity during launch and when navigating back to it from other activities and tasks with the BACK key. Be sure to test for navigation behaviors that might conflict with the user's expected behavior.

Defining launch modes

Launch modes allow you to define how a new instance of an activity is associated with the current task. You can define different launch modes in two ways:

Using the manifest file

When you declare an activity in your manifest file, you can specify how the activity should associate with tasks when it starts.

Using Intent flags

When you call `startActivity()`, you can include a flag in the `Intent` that declares how (or whether) the new activity should associate with the current task.

As such, if Activity A starts Activity B, Activity B can define in its manifest how it should associate with the current task (if at all) and Activity A can also request how Activity B should associate with current task. If both activities define how Activity B should associate with a task, then Activity A's request (as defined in the intent) is honored over Activity B's request (as defined in its manifest).

> **Note:** Some the launch modes available in the manifest are not available as flags for an intent and, likewise, some launch modes available as flags for an intent cannot be defined in the manifest.

Using the manifest file

When declaring an activity in your manifest file, you can specify how the activity should associate with a task using the `<activity>` element's `launchMode` attribute.

The `launchMode` attribute specifies an instruction on how the activity should be launched into a task. There are four different launch modes you can assign to the `launchMode` attribute:

`"standard"` (the default mode)
: Default. The system creates a new instance of the activity in the task from which it was started and routes the intent to it. The activity can be instantiated multiple times, each instance can belong to different tasks, and one task can have multiple instances.

`"singleTop"`
If an instance of the activity already exists at the top of the current task, the system routes the intent to that instance through a call to its `onNewIntent()` method, rather than creating a new instance of the activity. The activity can be instantiated multiple times, each instance can belong to different tasks, and one task can have multiple instances (but only if the the activity at the top of the back stack is *not* an existing instance of the activity).

For example, suppose a task's back stack consists of root activity A with activities B, C, and D on top (the stack is A-B-C-D; D is on top). An intent arrives for an activity of type D. If D has the default `"standard"` launch mode, a new instance of the class is launched and the stack becomes A-B-C-D-D. However, if D's launch mode is `"singleTop"`, the existing instance of D is deliverd the intent through `onNewIntent()`, because it's at the top of the stack—the stack remains A-B-C-D. However, if an intent arrives for an activity of type B, then a new instance of B is added to the stack, even if its launch mode is `"singleTop"`.

> **Note:** When a new instance of an activity is created, the user can press the BACK key to return to the previous activity. But when an existing instance of an activity handles a new intent, the user cannot press the BACK key to return to the state of the activity before the new intent arrived in `onNewIntent()`.

`"singleTask"`
The system creates a new task and instantiates the activity at the root of the new task. However, if an instance of the activity already exists in a separate task, the system routes the intent to the existing instance through a call to its `onNewIntent()` method, rather than creating a new instance. Only one instance of the activity can exist at a time.

> **Note:** Although the activity starts in a new task, the BACK key still returns the user to the previous activity.

`"singleInstance"`.
Same as `"singleTask"`, except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task; any activities started by this one open in a separate task.

As another example, the Android Browser application declares that the web browser activity should always open in its own task—by specifying the `singleTask` launch mode in the `<activity>` element. This means that if your application issues an intent to open the Android Browser, its activity is *not* placed in the same task as your application. Instead, either a new task starts for the Browser or, if the Browser already has a task running in the background, that task is brought forward to handle the new intent.

Regardless of whether an activity starts in a new task or in the same task as the activity that started it, the BACK key always takes the user to the previous activity. However, if you start an activity from your task (Task A) that specifies the `singleTask` launch mode, then that activity might have an instance in the background that belongs to a task with its own back stack (Task B). In this case, when Task B is brought forward to handle a new intent, the BACK key first navigates backward through the activities in Task B before returning to the top-most activity in Task A. Figure 4 visualizes this type of scenario.

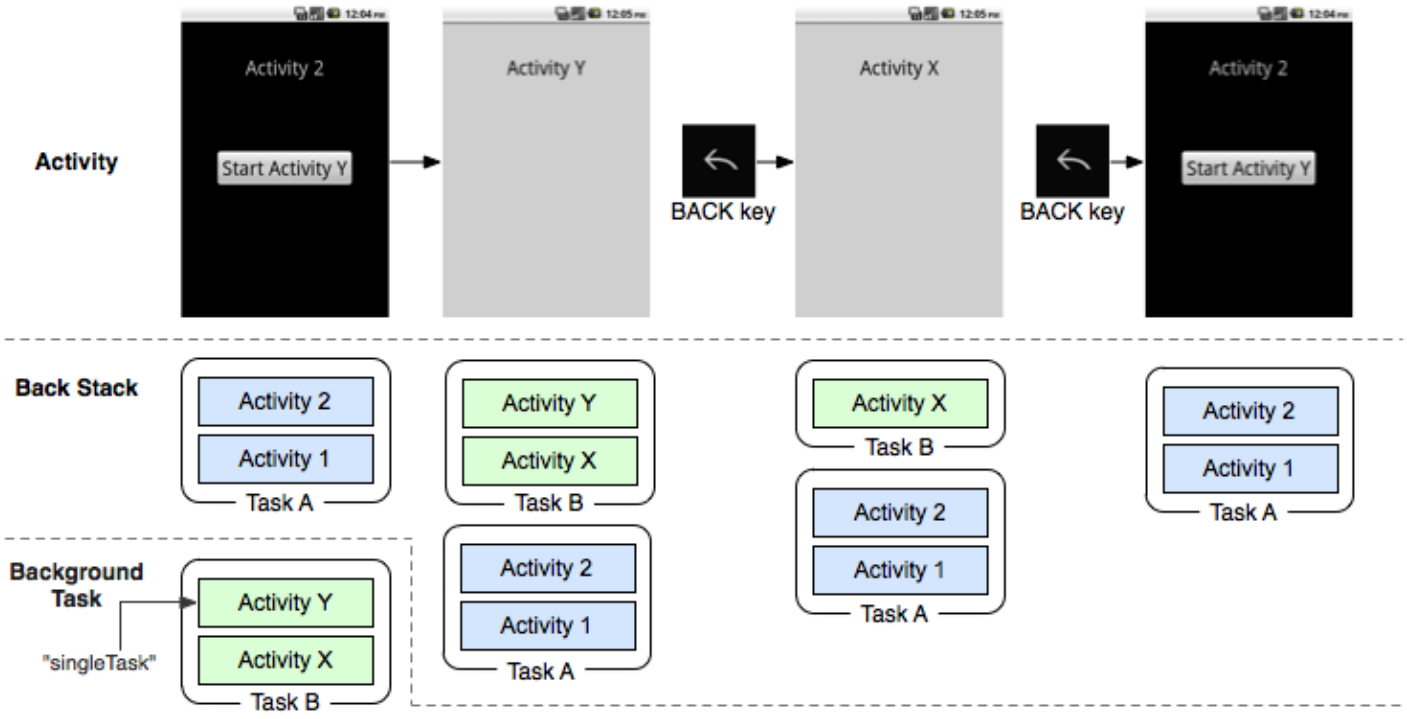

Figure 4. A representation of how an activity with launch mode "singleTask" is added to the back stack. If the activity is already a part of a background task with its own back stack (Task B), then the entire back stack also comes forward, on top of the current task (Task A).

For more information about using launch modes in the manifest file, see the `<activity>` element documentation, where the `launchMode` attribute and the accepted values are discussed more.

> **Note:** The behaviors that you specify for your activity with the `launchMode` attribute can be overriden by flags included with the intent that start your activity, as discussed in the next section.

Using Intent flags

When starting an activity, you can modify the default association of an activity to its task by including flags in the intent that you deliver to `startActivity()`. The flags you can use to modify the default behavior are:

`FLAG_ACTIVITY_NEW_TASK`
: Start the activity in a new task. If a task is already running for the activity you are now starting, that task is brought to the foreground with its last state restored and the activity receives the new intent in `onNewIntent()`.

  This produces the same behavior as the `"singleTask"` `launchMode` value, discussed in the previous section.

`FLAG_ACTIVITY_SINGLE_TOP`
: If the activity being started is the current activity (at the top of the back stack), then the existing instance receives a call to `onNewIntent()`, instead of creating a new instance of the activity.

  This produces the same behavior as the `"singleTop"` `launchMode` value, discussed in the previous section.

`FLAG_ACTIVITY_CLEAR_TOP`
: If the activity being started is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it are destroyed and this intent is delivered to the resumed instance of the activity (now on top), through `onNewIntent()`).

  There is no value for the `launchMode` attribute that produces this behavior.

  `FLAG_ACTIVITY_CLEAR_TOP` is most often used in conjunction with `FLAG_ACTIVITY_NEW_TASK`. When used together, these flags are a way of locating an existing activity in another task and putting it in a position where it can respond to the intent.

  > **Note:** If the launch mode of the designated activity is `"standard"`, it too is removed from the stack and a new instance is launched in its place to handle the incoming intent. That's because a new instance is always created for a new intent when the launch mode is `"standard"`.

Handling affinities

The *affinity* indicates which task an activity prefers to belong to. By default, all the activities from the same application have an affinity for each other. So, by default, all activities in the same application prefer to be in the same task. However, you can modify the default affinity for an activity. Activities defined in different applications can share an affinity, or activities defined in the same application can be assigned different task affinities.

You can modify the affinity for any given activity with the `taskAffinity` attribute of the `<activity>` element.

The `taskAffinity` attribute takes a string value, which must be unique from the default package name declared in the `<manifest>` element, because the system uses that name to identify the default task affinity for the application.

The affinity comes into play in two circumstances:

- When the intent that launches an activity contains the `FLAG_ACTIVITY_NEW_TASK` flag.

  A new activity is, by default, launched into the task of the activity that called `startActivity()`. It's pushed onto the same back stack as the caller. However, if the intent passed to `startActivity()` contains the `FLAG_ACTIVITY_NEW_TASK` flag, the system looks for a different task to house the new activity. Often, it's a new task. However, it doesn't have to be. If there's already an existing task with the same affinity as the new activity, the activity is launched into that task. If not, it begins a new task.

  If this flag causes an activity to begin a new task and the user presses the HOME key to leave it, there must be some way for the user to navigate back to the task. Some entities (such as the notification manager) always start activities in an external task, never as part of their own, so they always put `FLAG_ACTIVITY_NEW_TASK` in the intents they pass to `startActivity()`. If you have an activity that can be invoked by an external entity that might use this flag, take care that the user has a independent way to get back to the task that's started, such as with a launcher icon (the root activity of the task has a `CATEGORY_LAUNCHER` intent filter; see the Starting a task section below).

- When an activity has its `allowTaskReparenting` attribute set to `"true"`.

  In this case, the activity can move from the task it starts to the task it has an affinity for, when that task comes to the foreground.

  For example, suppose that an activity that reports weather conditions in selected cities is defined as part of a travel application. It has the same affinity as other activities in the same application (the default application affinity) and it allows re-parenting with this attribute. When one of your activities starts the weather reporter activity, it initially belongs to the same task as your activity. However, when the travel application's task comes to the foreground, the weather reporter activity is

reassigned to that task and displayed within it.

> **Tip:** If an `.apk` file contains more than one "application" from the user's point of view, you probably want to use the `taskAffinity` attribute to assign different affinities to the activities associated with each "application".

Clearing the back stack

If the user leaves a task for a long time, the system clears the task of all activities except the root activity. When the user returns to the task again, only the root activity is restored. The system behaves this way, because, after an extended amount of time, users likely have abandoned what they were doing before and are returning to the task to begin something new.

There are some activity attributes that you can use to modify this behavior:

`alwaysRetainTaskState`
: If this attribute is set to `"true"` in the root activity of a task, the default behavior just described does not happen. The task retains all activities in its stack even after a long period.

`clearTaskOnLaunch`
: If this attribute is set to `"true"` in the root activity of a task, the stack is cleared down to the root activity whenever the user leaves the task and returns to it. In other words, it's the opposite of `alwaysRetainTaskState`. The user always returns to the task in its initial state, even after a leaving the task for only a moment.

`finishOnTaskLaunch`
: This attribute is like `clearTaskOnLaunch`, but it operates on a single activity, not an entire task. It can also cause any activity to go away, including the root activity. When it's set to `"true"`, the activity remains part of the task only for the current session. If the user leaves and then returns to the task, it is no longer present.

Starting a task

You can set up an activity as the entry point for a task by giving it an intent filter with `"android.intent.action.MAIN"` as the specified action and `"android.intent.category.LAUNCHER"` as the specified category. For example:

```
<activity ... >
    <intent-filter ... >
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    ...
</activity>
```

An intent filter of this kind causes an icon and label for the activity to be displayed in the application launcher, giving users a way to launch the activity and to return to the task that it creates any time after it has been launched.

This second ability is important: Users must be able to leave a task and then come back to it later using this activity launcher. For this reason, the two launch modes that mark activities as always initiating a task, `"singleTask"` and `"singleInstance"`, should be used only when the activity has an `ACTION_MAIN` and a `CATEGORY_LAUNCHER` filter. Imagine, for example, what could happen if the filter is missing: An intent launches a `"singleTask"` activity, initiating a new task, and the user spends some time working in that task. The user then presses the HOME key. The task is now sent to the background and not visible. Because it is not represented in the application launcher, the user has no way to return to the task.

For those cases where you don't want the user to be able to return to an activity, set the `<activity>` element's `finishOnTaskLaunch` to `"true"` (see Clearing the stack).

← Back to Activities ↑ Go to top