Mobile Applications (420-P84-AB)

John Abbott College

Activity Lifecycle

Aakash Malhotra

Activity Lifecycle

As a user navigates through, out of, and back to your app, the `Activity` instances in your app transition through different states in their lifecycle. The `Activity` class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot. In other words, each callback allows you to perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant.

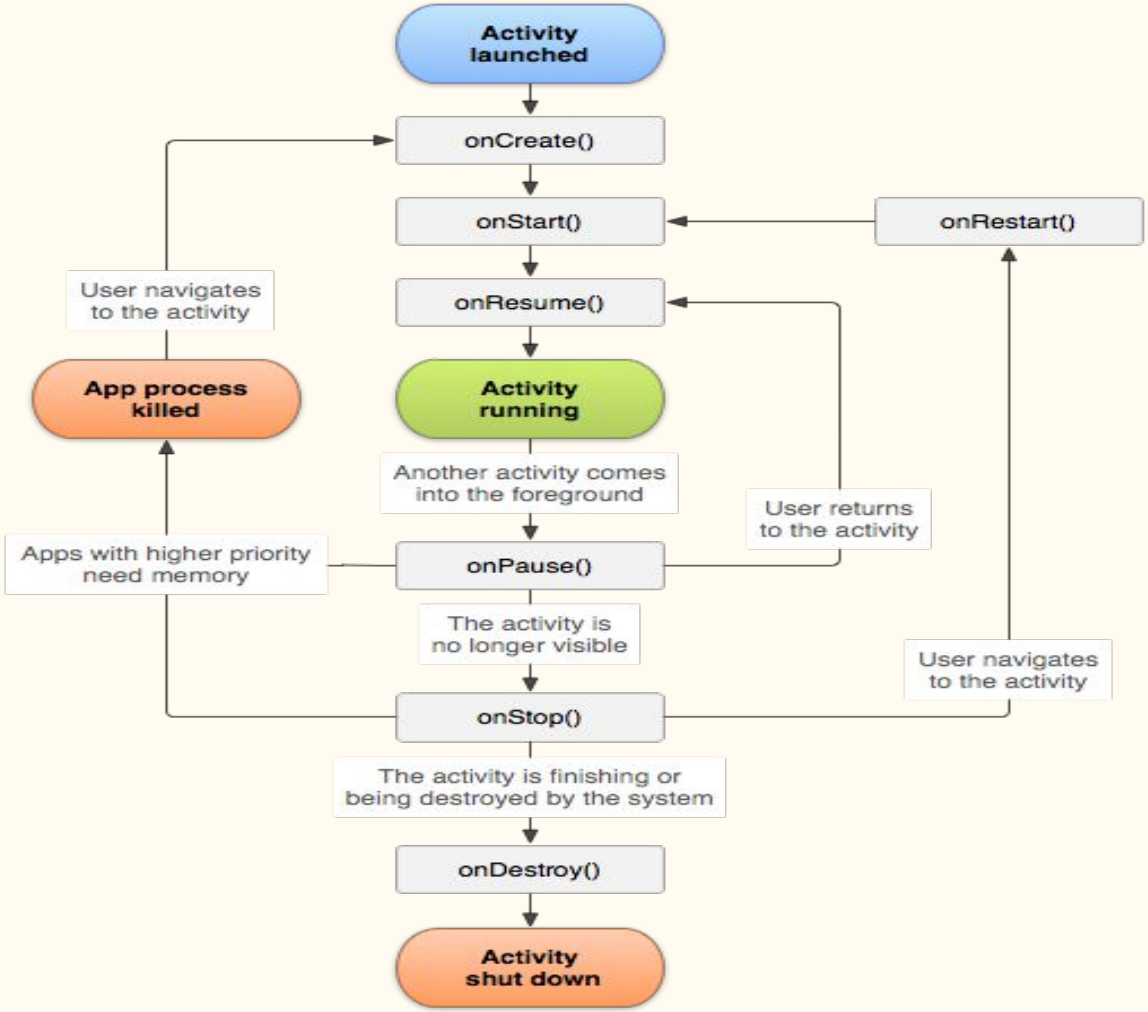
Activity Lifecycle

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.

Activity-Lifecycle Concepts

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`. The system invokes each of these callbacks as an activity enters a new state.


Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Activity Lifecycle Concepts

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.

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.

Lifecycle Callbacks

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()` 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()` might bind data to lists, associate the activity with a `ViewModel`, and instantiate some class-scope variables. This method receives the parameter `savedInstanceState`, which is a `Bundle` object containing the activity's previously saved state. If the activity has never existed before, the value of the `Bundle` object is null.

Lifecycle Callbacks

The following example of the `onCreate()` 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()`.

Lifecycle Callbacks

```
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

    super.onCreate(savedInstanceState);
```

Lifecycle Callbacks

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

    // set the user interface layout for this activity

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

Lifecycle Callbacks

```
// This callback is called only when there is a saved instance that is
previously saved by using
// onSaveInstanceState(). We restore some state in onCreate(), while we can
optionally restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    mTextView.setText(savedInstanceState.getString(TEXT_VIEW_KEY));
}
```

Lifecycle Callbacks

```
// 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);
}
```

Lifecycle Callbacks

As an alternative to defining the XML file and passing it to `setContentView()`, you can create new `View` objects in your activity code and build a view hierarchy by inserting new Views into a `ViewGroup`. You then use that layout by passing the root `ViewGroup` to `setContentView()`. For more information about creating a user interface, see the User Interface documentation.

Your activity does not reside in the Created state. After the `onCreate()` method finishes execution, the activity enters the *Started* state, and the system calls the `onStart()` and `onResume()` methods in quick succession. The next section explains the `onStart()` callback.

Lifecycle Callbacks

onStart()

When the activity enters the Started state, the system invokes this callback. The `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.

When the activity moves to the started state, any lifecycle-aware component tied to the activity's lifecycle will receive the `ON_START` event.

The `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()` method.

Lifecycle Callbacks

onResume()

When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the `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 the activity moves to the resumed state, any lifecycle-aware component tied to the activity's lifecycle will receive the `ON_RESUME` event. This is where the lifecycle components can enable any functionality that needs to run while the component is visible and in the foreground, such as starting a camera preview.

Lifecycle Callbacks

When an interruptive event occurs, the activity enters the *Paused* state, and the system invokes the `onPause()` callback.

If the activity returns to the Resumed state from the Paused state, the system once again calls `onResume()` method. For this reason, you should implement `onResume()` to initialize components that you release during `onPause()`, and perform any other initializations that must occur each time the activity enters the Resumed state.

Here is an example of a lifecycle-aware component that accesses the camera when the component receives the `ON_RESUME` event:

```
public class CameraComponent implements LifecycleObserver {

    ...

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void initializeCamera() {
        if (camera == null) {
            getCamera();
        }
    }

    ...
}
```

Lifecycle Callbacks

The code above initializes the camera once the `LifecycleObserver` receives the ON_RESUME event. In multi-window mode, however, your activity may be fully visible even when it is in the Paused state. For example, when the user is in multi-window mode and taps the other window that does not contain your activity, your activity will move to the Paused state. If you want the camera active only when the app is

Resumed (visible and active in the foreground), then initialize the camera after the ON_RESUME event demonstrated above. If you want to keep the camera active while the activity is Paused but visible (e.g. in multi-window mode) then you should instead initialize the camera after the ON_START event. Note, however, that having the camera active while your activity is Paused may deny access to the camera to another Resumed app in multi-window mode. Sometimes it may be necessary to keep the camera active while your activity is Paused, but it may actually degrade the overall user experience if you do. Think carefully about where in the lifecycle it is more appropriate to take control of shared system resources in the context of multi-window. To learn more about supporting multi-window mode, see Multi-Window Support.

Lifecycle Callbacks

Regardless of which build-up event you choose to perform an initialization operation in, make sure to use the corresponding lifecycle event to release the resource. If you initialize something after the ON_START event, release or terminate it after the ON_STOP event. If you initialize after the ON_RESUME event, release after the ON_PAUSE event.

Note, the code snippet above places camera initialization code in a lifecycle aware component. You can instead put this code directly into the activity lifecycle callbacks such as `onStart()` and `onStop()`but this is not recommended. Adding this logic into an independent, lifecycle-aware component allows you to reuse the component across multiple activities without having to duplicate code.

Lifecycle Callbacks

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); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode). Use the `onPause()` method to pause or adjust operations that should not continue (or should continue in moderation) while the `Activity` is in the Paused state, and that you expect to resume shortly. There are several reasons why an activity may enter this state. For example:

- Some event interrupts app execution, as described in the 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.

Lifecycle Callbacks

- 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.

When the activity moves to the paused state, any lifecycle-aware component tied to the activity's lifecycle will receive the `ON_PAUSE` event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not in the foreground, such as stopping a camera preview.

You can also use the `onPause()` method to release system resources, 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. However, as mentioned above in the onResume() section, a Paused activity may still be fully visible if in multi-window mode. As such, you should consider using onStop() instead of onPause() to fully release or adjust UI-related resources and operations to better support multi-window mode.

The following example of a `LifecycleObserver` reacting to the ON_PAUSE event is the counterpart to the ON_RESUME event example above, releasing the camera that was initialized after the ON_RESUME event was received:

```
public class JavaCameraComponent implements LifecycleObserver {
    ...
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void releaseCamera() {
        if (camera != null) {
            camera.release();
            camera = null;
        }
    }

    ...
}
```

Lifecycle Callbacks

Note, the code snippet above places camera release code after the ON_PAUSE event is received by the LifecycleObserver.

`onPause()` execution is very brief, and does not necessarily afford enough time to perform save operations. For this reason, you should **not** use `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()`. For more information about suitable operations to perform during `onStop()`, see onStop().

Completion of the `onPause()` method does not mean that the activity leaves the Paused state. Rather, the activity remains in this state until either the activity resumes or becomes completely invisible to the user. If the activity resumes, the system once again invokes the `onResume()` callback.

Lifecycle Callbacks

If the activity returns from the Paused state to the Resumed state, the system keeps the `Activity` instance resident in memory, recalling that instance when the system invokes `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()`.

onStop()

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

Lifecycle Callbacks

When the activity moves to the stopped state, any lifecycle-aware component tied to the activity's lifecycle will receive the `ON_STOP` event. This is where the lifecycle components can stop any functionality that does not need to run while the component is not visible on the screen.

In the `onStop()` method, the app should release or adjust resources that are not needed while the app is not visible to the user. For example, your app might pause animations or switch from fine-grained to coarse-grained location updates.

You should also use `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()`. The following example shows an implementation of `onStop()` that saves the contents of a draft note to persistent storage:

```
@Override
protected void onStop() {
    // call the superclass method first
    super.onStop();
    // save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    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.
        );
}
```

Lifecycle Callbacks

Note, the code sample above uses SQLite directly. You should instead use Room, a persistence library that provides an abstraction layer over SQLite. To learn more about the benefits of using Room, and how to implement Room in your app, see the Room Persistence Library guide.

When your activity enters the Stopped state, the `Activity` 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` object in the layout, so if the user entered text into an `EditText` widget, that content is retained so you don't need to save and restore it.

Lifecycle Callbacks

Note, the code sample above uses SQLite directly. You should instead use Room, a persistence library that provides an abstraction layer over SQLite. To learn more about the benefits of using Room, and how to implement Room in your app, see the Room Persistence Library guide.

When your activity enters the Stopped state, the `Activity` 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` object in the layout, so if the user entered text into an `EditText` widget, that content is retained so you don't need to save and restore it.

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()`. If the `Activity` is finished running, the system calls `onDestroy()`.

Lifecycle Callbacks

onDestroy()

`onDestroy()` is called before the activity is destroyed. The system invokes this callback either because:

1. the activity is finishing (due to the user completely dismissing the activity or due to `finish()`being called on the activity), or
2. the system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)

When the activity moves to the destroyed state, any lifecycle-aware component tied to the activity's lifecycle will receive the `ON_DESTROY` event. This is where the lifecycle components can clean up anything it needs to before the Activity is destroyed.

Activity State and Ejection from Memory

The system kills processes when it needs to free up RAM; the likelihood of the system 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. Table below 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

Saving and Restoring transient UI State

A user expects an activity's UI state to remain the same throughout a configuration change, such as rotation or switching into multi-window mode. However, the system destroys the activity by default when such a configuration change occurs, wiping away any UI state stored in the activity instance. Similarly, a user expects UI state to remain the same if they temporarily switch away from your app to a different app and then come back to your app later. However, the system may destroy your application's process while the user is away and your activity is stopped.

When the activity is destroyed due to system constraints, you should preserve the user's transient UI state using a combination of `ViewModel`, `onSaveInstanceState()`, and/or local storage.

Saving and Restoring transient UI State

Instance state

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` object. By default, the system uses the `Bundle` instance state to save information about each `View` object in your activity layout (such as the text value entered into an `EditText` 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.

Bundle

A `Bundle` object isn't appropriate for preserving more than a trivial amount of data because it requires serialization on the main thread and 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()` method, and the `ViewModel` class.

OnSaveInstanceState

As your activity begins to stop, the system calls the `onSaveInstanceState()` method so your activity can save state information to an instance state bundle. 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` widget or the scroll position of a `ListView` widget.

To save additional instance state information for your activity, you must override `onSaveInstanceState()` and add key-value pairs to the `Bundle` object that is saved in the event that your activity is destroyed unexpectedly. If you override onSaveInstanceState(), you must call the superclass implementation if you want the default implementation to save the state of the view hierarchy. For example:

OnSaveInstanceState

```
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);
}
```

Restore UI State

When your activity is recreated after it was previously destroyed, you can recover your saved instance state from the `Bundle` that the system passes to your activity. Both the `onCreate()` and `onRestoreInstanceState()` callback methods receive the same `Bundle` that contains the instance state information.

Because the `onCreate()` method is called whether the system is creating a new instance of your activity or recreating a previous one, you must 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.

Restore UI State

```
@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
    }
    // ...
}
```

onRestoreInstanceState

Instead of restoring the state during `onCreate()` you may choose to implement `onRestoreInstanceState()`, which the system calls after the `onStart()` method. The system calls `onRestoreInstanceState()` only if there is a saved state to restore, so you do not need to check whether the `Bundle` is null.

onRestoreInstanceState

```
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);
}
```

References:

- https://developer.android.com/guide/components/activities/activity-lifecycle