Android Developers

# Handling Lifecycles with Lifecycle-Aware Components

Lifecycle-aware components perform actions in response to a change in the lifecycle status of another component, such as activities and fragments. These components help you produce better-organized, and often lighter-weight code, that is easier to maintain.

A common pattern is to implement the actions of the dependent components in the lifecycle methods of activities and fragments. However, this pattern leads to a poor organization of the code and to the proliferation of errors. By using lifecycle-aware components, you can move the code of dependent components out of the lifecycle methods and into the components themselves.

The `android.arch.lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/package-summary.html) package provides classes and interfaces that let you build *lifecycle-aware* components—which are components that can automatically adjust their behavior based on the current lifecycle state of an activity or fragment.

> **Note:** To import `android.arch.lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/package-summary.html) into your Android project, see adding components to your project (https://developer.android.google.cn/topic/libraries/architecture/adding-components.html).

Most of the app components that are defined in the Android Framework have lifecycles attached to them. Lifecycles are managed by the operating system or the framework code running in your process. They are core to how Android works and your application must respect them. Not doing so may trigger memory leaks or even application crashes.

Imagine we have an activity that shows the device location on the screen. A common implementation might be like the following:

```
class MyLocationListener {
    public MyLocationListener(Context context, Callback callback) {
        // ...
    }

    void start() {
        // connect to system location service
```

Issue Tracker

(https://issuetracker.google.com/issues/new?component=197448&template=878802)

Report issues so we can fix bugs.

G+ Community

(https://plus.google.com/b/108967384991768947849/communities/105153134372062985968/stream/1bd006e4-f2d6-41cc-93c6-38b4afcdc36f)

Provide feedback and discuss ideas with other developers.

Codelab

(https://codelabs.developers.google.com/codelabs/android-lifecycles/#0)

Sample App

(https://github.com/googlesamples/android-architecture-components)

In this document

```
    }

    void stop() {
        // disconnect from system location service
    }
}

class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    @Override
    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, (location) -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        myLocationListener.start();
        // manage other components that need to respond
        // to the activity lifecycle
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
        // manage other components that need to respond
        // to the activity lifecycle
    }
}
```

Even though this sample looks fine, in a real app, you end up having too many calls that manage the UI and other components in response to the current state of the lifecycle. Managing multiple components places a considerable amount of code in lifecycle methods, such as `onStart()` `(https://developer.android.google.cn/reference/android/app/Activity.html#onStart())` and `onStop()` `(https://developer.android.google.cn/reference/android/app/Activity.html#onStop())`, which makes them difficult to maintain.

Moreover, there's no guarantee that the component starts before the activity or fragment is stopped. This is especially true if we need to perform a long-running operation, such as some configuration check in `onStart()` `(https://developer.android.google.cn/reference/android/app/Activity.html#onStart())`. This can cause a race condition where the `onStop()` `(https://developer.android.google.cn/reference/android/app/Activity.html#onStop())` method finishes before the `onStart()` `(https://developer.android.google.cn/reference/android/app/Activity.html#onStart())`, keeping the component alive longer than it's needed.

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;
    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, location -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        Util.checkUserStatus(result -> {
            // what if this callback is invoked AFTER activity is stopped?
            if (result) {
                myLocationListener.start();
            }
        });
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
    }
}
```

The `android.arch.lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/package-summary.html) package provides classes and interfaces that help you tackle these problems in a resilient and isolated way.

## Lifecycle

`Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.

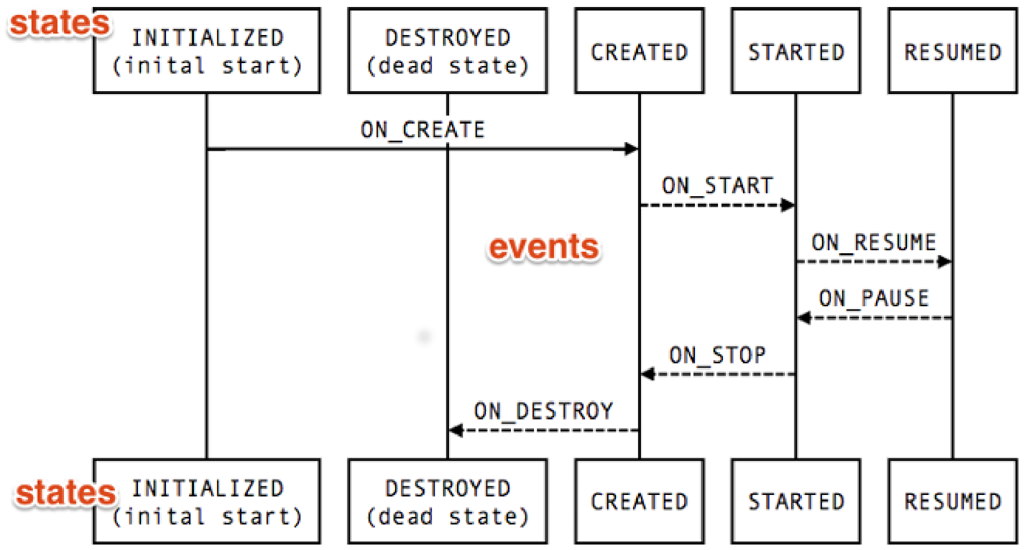
`Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) uses two main enumerations to track the lifecycle status for its associated component:

Event

> The lifecycle events that are dispatched from the framework and the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) class. These events map to the callback events in activities and fragments.

State

The current state of the component tracked by the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) object.



Think of the states as nodes of a graph and events as the edges between these nodes.

A class can monitor the component's lifecycle status by adding annotations to its methods. Then you can add an observer by calling the `addObserver()` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html#addObserver(android.arch.lifecycle.LifecycleObserver)) method of the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) class and passing an instance of your observer, as shown in the following example:

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void connectListener() {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void disconnectListener() {
        ...
    }
}

myLifecycleOwner.getLifecycle().addObserver(new MyObserver());
```

In the example above, the `myLifecycleOwner` object implements the `LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html) interface, which is explained in the following section.

## LifecycleOwner

`LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html) is a single method interface that denotes that the class has a `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html). It has one method, `getLifecycle()` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html#getLifecycle()), which must be implemented by the class. If you're trying to manage the lifecycle of a whole application process instead, see `ProcessLifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/ProcessLifecycleOwner.html).

This interface abstracts the ownership of a `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) from individual classes, such as `Fragment (https://developer.android.google.cn/reference/android/support/v4/app/Fragment.html)` and `AppCompatActivity (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html)`, and allows writing components that work with them. Any custom application class can implement the `LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html) interface.

Components that implement `LifecycleObserver` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleObserver.html) work seamlessly with components that implement `LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html) because an owner can provide a lifecycle, which an observer can register to watch.

For the location tracking example, we can make the `MyLocationListener` class implement `LifecycleObserver` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleObserver.html) and then initialize it with the activity's `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) in the `onCreate() (https://developer.android.google.cn/reference/android/app/Activity.html#onCreate(android.os.Bundle))` method. This allows the `MyLocationListener` class to be self-sufficient, meaning that the logic to react to changes in lifecycle status is declared in `MyLocationListener` instead of the activity. Having the individual components store their own logic makes the activities and fragments logic easier to manage.

```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, getLifecycle(), location -> {
            // update UI
        });
        Util.checkUserStatus(result -> {
            if (result) {
                myLocationListener.enable();
            }
        });
```

```
  }
}
```

A common use case is to avoid invoking certain callbacks if the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) isn't in a good state right now. For example, if the callback runs a fragment transaction after the activity state is saved, it would trigger a crash, so we would never want to invoke that callback.

To make this use case easy, the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) class allows other objects to query the current state.

```
class MyLocationListener implements LifecycleObserver {
    private boolean enabled = false;
    public MyLocationListener(Context context, Lifecycle lifecycle, Callback callback) {
       ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    void start() {
        if (enabled) {
           // connect
        }
    }

    public void enable() {
        enabled = true;
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {
            // connect if not connected
        }
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    void stop() {
        // disconnect if connected
    }
}
```

With this implementation, our `LocationListener` class is completely lifecycle-aware. If we need to use our `LocationListener` from another activity or fragment, we just need to initialize it. All of the setup and teardown operations are managed by the class itself.

If a library provides classes that need to work with the Android lifecycle, we recommend that you use lifecycle-aware components. Your library clients can easily integrate those components without manual lifecycle management on the client side.

## Implementing a custom LifecycleOwner

Fragments and Activities in Support Library 26.1.0 and later already implement the `LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html) interface.

If you have a custom class that you would like to make a `LifecycleOwner` (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleOwner.html), you can use the LifecycleRegistry (https://developer.android.google.cn/reference/android/arch/lifecycle/LifecycleRegistry.html) class, but you need to forward events into that class, as shown in the following code example:

```
public class MyActivity extends Activity implements LifecycleOwner {
    private LifecycleRegistry mLifecycleRegistry;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mLifecycleRegistry = new LifecycleRegistry(this);
        mLifecycleRegistry.markState(Lifecycle.State.CREATED);
    }

    @Override
    public void onStart() {
        super.onStart();
        mLifecycleRegistry.markState(Lifecycle.State.STARTED);
    }

    @NonNull
    @Override
    public Lifecycle getLifecycle() {
        return mLifecycleRegistry;
    }
}
```

## Best practices for lifecycle-aware components

- Keep your UI controllers (activities and fragments) as lean as possible. They should not try to acquire their own data; instead, use a `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html) to do that, and observe a `LiveData` (https://developer.android.google.cn/reference/android/arch/lifecycle/LiveData.html) object to reflect the changes back to the views.
- Try to write data-driven UIs where your UI controller's responsibility is to update the views as data changes, or notify user actions back to the `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html).
- Put your data logic in your `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html) class. `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html) should serve as the connector between your UI controller and the rest of your app. Be careful though, it isn't `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html)'s responsibility to fetch data (for example, from a network). Instead, `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html) should call the appropriate component to fetch the data, then provide the result back to the UI controller.

- Use Data Binding (https://developer.android.google.cn/topic/libraries/data-binding/index.html) to maintain a clean interface between your views and the UI controller. This allows you to make your views more declarative and minimize the update code you need to write in your activities and fragments. If you prefer to do this in the Java programming language, use a library like Butter Knife (http://jakewharton.github.io/butterknife/) to avoid boilerplate code and have a better abstraction.
- If your UI is complex, consider creating a presenter (http://www.gwtproject.org/articles/mvp-architecture.html#presenter) class to handle UI modifications. This might be a laborious task, but it can make your UI components easier to test.
- Avoid referencing a `View (https://developer.android.google.cn/reference/android/view/View.html)` or `Activity (https://developer.android.google.cn/reference/android/app/Activity.html)` context in your `ViewModel` (https://developer.android.google.cn/reference/android/arch/lifecycle/ViewModel.html). If the `ViewModel` outlives the activity (in case of configuration changes), your activity leaks and isn't properly disposed by the garbage collector.

## Use cases for lifecycle-aware components

Lifecycle-aware components can make it much easier for you to manage lifecycles in a variety of cases. A few examples are:

- Switching between coarse and fine-grained location updates. Use lifecycle-aware components to enable fine-grained location updates while your location app is visible and switch to coarse-grained updates when the app is in the background. `LiveData` (https://developer.android.google.cn/reference/android/arch/lifecycle/LiveData.html), a lifecycle-aware component, allows your app to automatically update the UI when your use changes locations.
- Stopping and starting video buffering. Use lifecycle-aware components to start video buffering as soon as possible, but defer playback until app is fully started. You can also use lifecycle-aware components to terminate buffering when your app is destroyed.
- Starting and stopping network connectivity. Use lifecycle-aware components to enable live updating (streaming) of network data while an app is in the foreground and also to automatically pause when the app goes into the background.
- Pausing and resuming animated drawables. Use lifecycle-aware components to handle pausing animated drawables when while app is in the background and resume drawables after the app is in the foreground.

## Handling on stop events

When a `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) belongs to an `AppCompatActivity (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html)` or `Fragment (https://developer.android.google.cn/reference/android/support/v4/app/Fragment.html)`, the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html)'s state changes to `CREATED` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#CREATED) and the `ON_STOP` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.Event.html#ON_STOP) event is dispatched when the `AppCompatActivity` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html) or `Fragment` (https://developer.android.google.cn/reference/android/support/v4/app/Fragment.html)'s `onSaveInstanceState()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onSaveInstanceState(android.os.Bundle)) is called.

When a `Fragment` (https://developer.android.google.cn/reference/android/support/v4/app/Fragment.html) or `AppCompatActivity` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html)'s state is saved via `onSaveInstanceState()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onSaveInstanceState(android.os.Bundle)), it's UI is considered immutable until `ON_START` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.Event.html#ON_START) is called. Trying to modify the UI after the state is saved is likely to cause inconsistencies in the navigation state of your application which is why `FragmentManager` (https://developer.android.google.cn/reference/android/support/v4/app/FragmentManager.html) throws an exception if the app runs a `FragmentTransaction` (https://developer.android.google.cn/reference/android/support/v4/app/FragmentTransaction.html) after state is saved. See `commit()` (https://developer.android.google.cn/reference/android/support/v4/app/FragmentTransaction.html#commit()) for details.

`LiveData` (https://developer.android.google.cn/reference/android/arch/lifecycle/LiveData.html) prevents this edge case out of the box by refraining from calling its observer if the observer's associated `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) isn't at least `STARTED` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#STARTED). Behind the scenes, it calls `isAtLeast()` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#isAtLeast(android.arch.lifecycle.Lifecycle.State)) before deciding to invoke its observer.

Unfortunately, `AppCompatActivity` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html)'s `onStop()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onStop()) method is called *after* `onSaveInstanceState()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onSaveInstanceState(android.os.Bundle)), which leaves a gap where UI state changes are not allowed but the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) has not yet been moved to the `CREATED` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#CREATED) state.

To prevent this issue, the `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) class in version `beta2` and lower mark the state as `CREATED` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#CREATED) without dispatching the event so that any code that checks the current state gets the real value even though the event isn't dispatched until `onStop()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onStop()) is called by the system.

Unfortunately, this solution has two major problems:

- On API level 23 and lower, the Android system actually saves the state of an activity even if it is *partially* covered by another activity. In other words, the Android system calls `onSaveInstanceState()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onSaveInstanceState(android.os.Bun dle)) but it doesn't necessarily call `onStop()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onStop()). This creates a potentially long interval where the observer still thinks that the lifecycle is active even though its UI state can't be modified.

- Any class that wants to expose a similar behavior to the `LiveData` (https://developer.android.google.cn/reference/android/arch/lifecycle/LiveData.html) class has to implement the workaround provided by `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) version `beta 2` and lower.

> **Note:** To make this flow simpler and provide better compatibility with older versions, starting at version `1.0.0-rc1`, `Lifecycle` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.html) objects are marked as `CREATED` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.State.html#CREATED) and `ON_STOP` (https://developer.android.google.cn/reference/android/arch/lifecycle/Lifecycle.Event.html#ON_STOP) is dispatched when `onSaveInstanceState()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onSaveInstanceState(android.os.Bundle)) is called without waiting for a call to the `onStop()` (https://developer.android.google.cn/reference/android/support/v7/app/AppCompatActivity.html#onStop()) method. This is unlikely to impact your code but it is something you need to be aware of as it doesn't match the call order in the `Activity` (https://developer.android.google.cn/reference/android/app/Activity.html) class in API level 26 and lower.