# 

[**Handling Lifecycle In Android Applications**](#_5siq6p9jycn2) **1**

[Lifecycle](#_84hlzgu61jxk) 2

[LifecycleOwner](#_wgainfrk9u2z) 3

[Best Practices About Lifecycles](#_hr7qouqf7cpn) 5

[Addendum](#_jkkd2mwlc0lf) 6

[Implementing LifecycleOwner in Custom Activities / Fragments](#_6cv18ryi7ojc) 6

*(Shortlink:* [*go/flatfoot-lifecycles-draft*](https://goto.google.com/flatfoot-lifecycles-draft)*)*

# Handling Lifecycle In Android Applications

Most of the App Components that are defined in the Android Framework have lifecycles attached to them. These 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  }   void stop() {  // disconnect from system location service  } } |

|  |
| --- |
| class MyActivity extends AppCompatActivity {  private MyLocationListener myLocationListener;   public void onCreate(...) {  myLocationListener = new MyLocationListener(this, (location) -> {  // update UI  });  }   public void onStart() {  super.onStart();  myLocationListener.start();  }   public void onStop() {  super.onStop();  myLocationListener.stop();  } } |

Even though this sample looks fine, in a real app, you end up having way too many calls like this and the *onStart()* and *onStop()* methods become very large.

Moreover, some components cannot be just started in *onStart().* What if we needed to check some configuration before starting the location observer? It is possible that in some cases the check will finish after the *Activity* is stopped, which will mean that *myLocationListener.start()* will be called **after** *myLocationListener.stop()* is called, basically keeping the connection forever.

|  |
| --- |
| class MyActivity extends AppCompatActivity {  private MyLocationListener myLocationListener = ...;   public void onCreate(...) {  myLocationListener = new MyLocationListener(this, location -> {  // update UI  });  }   public void onStart() {  super.onStart();  Util.checkUserStatus(result -> {  // what if this callback is invoked AFTER activity is stopped?  if(result) {  myLocationListener.start();  }  });  }   public void onStop() {  super.onStop();  myLocationListener.stop();  } } |

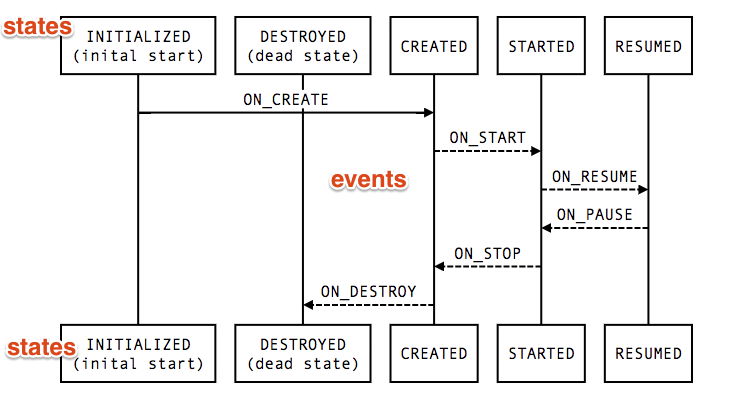
Lifecycles provides a set of classes that help you tackle these problems in a resilient and isolated way.

## Lifecycle

Lifecycle is a core 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 uses 2 main enumerations to track the lifecycle status for its associated component.

* **Event:** These are the lifecycle events that are dispatched from the Framework and the Lifecycle class. These events map to the callback events in Activities and Fragments.
* **State:** The current state of the component tracked by the Lifecycle object.



You can consider 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.

|  |
| --- |
| public class MyObserver implements LifecycleObserver {  @OnLifecycleEvent(Lifecycle.ON\_RESUME)  public void onResume() {  }   @OnLifecycleEvent(Lifecycle.ON\_PAUSE)  public void onPause() {  } } aLifecycleOwner.getLifecycle().addObserver(new MyObserver()); |

### LifecycleOwner

LifecycleOwner is a single method interface that denotes that the class has a Lifecycle. It has 1 method, *getLifecycle(),* which must be implemented by the class.

This class abstracts the ownership of a Lifecycle from individual classes (for example, Activities and Fragments) and allows writing components that can work with both of them. Any custom application class can implement the *LifecycleOwner* interface.

**Note:** Since Lifecycles project is in alpha stage, *Fragment* and *AppCompatActivity* classes cannot implement it (because we cannot add a dependency from a stable component to an unstable API). Until Lifecycle is stable, *LifecycleActivity* and *LifecycleFragment* classes are provided for convenience. After the Lifecycles project is released, support library Fragments and Activities will implement the LifecycleOwner interface. Also, see [Implementing LifecycleOwner in Custom Activities / Fragments](#_6cv18ryi7ojc).

For the example above, we can make our *MyLocationListener* class a LifecycleObserver, initialize it with our Lifecycle on *onCreate* and stop worrying about it afterwards. This allows *MyLocationListener* class to be self sufficient, meaning that it can do its own cleanup when necessary.

|  |
| --- |
| class MyActivity extends LifecycleActivity {  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 not invoking certain callbacks if the Lifecycle is not 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 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.ON\_START)  void start() {  if (enabled) {  // connect  }  }   public void enable() {  enabled = true;  if (lifecycle.getState() >= STARTED) {  // connect if not connected  }  }   @OnLifecycleEvent(Lifecycle.ON\_STOP)  void stop() {  // disconnect if connected  } } |

With this implementation, our LocationListener class is completely lifecycle-aware; it can do its own initialization and cleanup without being managed by the Activity. If we need to use our LocationListener from another Activity or another Fragment, all we need to do is to initialize it. All of the setup and teardown operations will be managed by the class itself.

Classes which can work with a Lifecycle are called **lifecycle-aware components.** Libraries which provide classes that need to work with the Android lifecycle are encouraged to provide lifecycle-aware components, so that their clients can easily integrate those classes without manual lifecycle management on the client side.

[LiveData](https://docs.google.com/document/d/17r2Mh88a2u9z2kDwCYEZfBRTSQPSWYaHAJkGGLUlPqw/edit) is an example of a lifecycle-aware component. Using LiveData together with [ViewModel](https://docs.google.com/document/d/1kXxcB7_0xvexTMLgpddd8lA2znIe-DV2mCKSsypm8Ww/edit) makes it much easier to populate UIs with data while respecting Android lifecycles.

## Best Practices About Lifecycles

* Keep your UI Controllers (activities & fragments) as lean as possible. They should not try to acquire their data; instead, use a ViewModel to do that work, and observe the ViewModel 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.
* Put your data logic in your ViewModel class. ViewModel should serve as the connector between your UI Controller and the rest of your application. Be careful though, it is **not** ViewModel's responsibility to fetch data (for example, from a network). Instead, Viewmodel should call the appropriate component to do that work, then provide the result back to the UI Controller.
* Use [Data Binding](https://developer.android.com/topic/libraries/data-binding/index.html) to have a clean interface between your views and the UI Controller. This will allow 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 Java, use a library like [Butter Knife](http://jakewharton.github.io/butterknife/) to avoid boilerplate code and have a better abstraction.
* If you have a complex UI, consider creating a Presenter class to handle UI modifications. This is usually an overkill but might be useful to make your UIs easier to test.
* Never ever reference a View or Activity context in your ViewModel. Keep in mind that they outlive the Activity (in case of configuration changes).

## Addendum

### Implementing LifecycleOwner in Custom Activities / Fragments

Any Fragment or Activity can be turned into a LifecycleOwner by using the built in LifecycleRegistry class.

|  |
| --- |
| public class MyFragment extends Fragment implements LifecycleRegistryOwner {  LifecycleRegistry lifecycleRegistry = new LifecycleRegistry(this);   @Override  public LifecycleRegistry getLifecycle() {  return lifecycleRegistry;  } } |

If you have a custom class that you would like to make a LifecycleOwner, you can use the *LifecycleRegistry* class, but you will need to forward events into that class. (This forwarding is done automatically for fragments and activities if they implement *LifecycleRegistryOwner* interface).