How to Leak a Context: Handlers & Inner Classes

Posted Jan 14, 2013 by Alex Lockwood

0

Consider the following code:

```
public class SampleActivity extends Activity {

  private final Handler mLeakyHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
      // ...
    }
  }
}
```

While not readily obvious, this code can cause cause a massive memory leak. Android Lint will give the following warning:

> *In Android, Handler classes should be static or leaks might occur.*

But where exactly is the leak and how might it happen? Let's determine the source of the problem by first documenting what we know:

1. When an Android application first starts, the framework creates a `Looper` object for the application's main thread. A `Looper` implements a simple message queue, processing `Message` objects in a loop one after another. All major application framework events (such as Activity lifecycle method calls, button clicks, etc.) are contained inside `Message` objects, which are added to the `Looper`'s message queue and are processed one-by-one. The main thread's `Looper` exists throughout the application's lifecycle.

2. When a `Handler` is instantiated on the main thread, it is associated with the `Looper`'s message queue. Messages posted to the message queue will hold a reference to the `Handler` so that the framework can call `Handler#handleMessage(Message)` when the `Looper` eventually processes the message.

3. In Java, non-static inner and anonymous classes hold an implicit reference to their outer class. Static inner classes, on the other hand, do not.

So where exactly is the memory leak? It's very subtle, but consider the following code as an example:

```
1  public class SampleActivity extends Activity {
2
3    private final Handler mLeakyHandler = new Handler() {
4      @Override
5      public void handleMessage(Message msg) {
6        // ...
7      }
8    }
9
10   @Override
11   protected void onCreate(Bundle savedInstanceState) {
12     super.onCreate(savedInstanceState);
13
14     // Post a message and delay its execution for 10 minu
15     mLeakyHandler.postDelayed(new Runnable() {
16       @Override
17       public void run() { /* ... */ }
18     }, 1000 * 60 * 10);
19
20     // Go back to the previous Activity.
21     finish();
22   }
23 }
```

When the activity is finished, the delayed message will continue to live in the main thread's message queue for 10 minutes before it is processed. The message holds a reference to the activity's `Handler`, and the `Handler` holds an implicit reference to its outer class (the`SampleActivity`, in this case). This reference will persist until the message is processed, thus preventing the activity context from being garbage collected and leaking all of the application's resources. Note that the same is true with the anonymous Runnable class on line 15. Non-static instances of anonymous classes hold an implicit reference to their outer class, so the context will be leaked.

To fix the problem, subclass the `Handler` in a new file or use a static inner class instead. Static inner classes do not hold an implicit reference to their outer class, so the activity will not be leaked. If you need to invoke the outer activity's methods from within the `Handler`, have the Handler hold a `WeakReference` to the activity so you don't accidentally leak a context. To fix the

memory leak that occurs when we instantiate the anonymous Runnable class, we make the variable a static field of the class (since static instances of anonymous classes do not hold an implicit reference to their outer class):

```
1  public class SampleActivity extends Activity {
2
3    /**
4     * Instances of static inner classes do not hold an imp
5     * reference to their outer class.
6     */
7    private static class MyHandler extends Handler {
8      private final WeakReference<SampleActivity> mActivity
9
10     public MyHandler(SampleActivity activity) {
11       mActivity = new WeakReference<SampleActivity>(activ
12     }
13
14     @Override
15     public void handleMessage(Message msg) {
16       SampleActivity activity = mActivity.get();
17       if (activity != null) {
18         // ...
19       }
20     }
21   }
22
23   private final MyHandler mHandler = new MyHandler(this);
24
25   /**
26    * Instances of anonymous classes do not hold an implic
27    * reference to their outer class when they are "static
28    */
29   private static final Runnable sRunnable = new Runnable(
30       @Override
31       public void run() { /* ... */ }
32   };
33
34   @Override
35   protected void onCreate(Bundle savedInstanceState) {
36     super.onCreate(savedInstanceState);
37
38     // Post a message and delay its execution for 10 minu
39     mHandler.postDelayed(sRunnable, 1000 * 60 * 10);
40
41     // Go back to the previous Activity.
42     finish();
43   }
44 }
```

The difference between static and non-static inner classes is subtle, but is something every Android developer should understand. What's the bottom line? Avoid using non-static inner

classes in an activity if instances of the inner class could outlive the activity's lifecycle.Instead, prefer static inner classes and hold a weak reference to the activity inside.

As always, leave a comment if you have any questions and don't forget to +1 this blog in the top right corner! :)

Last updated December 12, 2014.

« Newer Older »