Finding Memory Leaks in Java with JDeveloper

Benjamin Livshits
Computer Science Department
Stanford University
Stanford, USA
livshits@cs.stanford.edu

Learn how memory profiling and heap debugging can help you get rid of memory leaks such as those caused by use of the listener pattern.

Contrary to the popular believe, Java's garbage collector does not solve all memory problems that might occur in practice. In this article we look at how Oracle JDeveloper's memory profiler feature can be used to help find leaking memory in your applications. We also show how memory leaks, once detected, can be eliminated.

1 Background

A key feature of memory management in Java is its garbage-collected heap. A typical garbage collector that comes with Java is a tracing collector, which determines which objects should be preserved in memory by tracing all objects reachable from a set of roots. These reachable objects survive collections because they may conceivably be used in the course of program execution.

While this approach usually works well for reclaiming memory no longer needed by the program, it is a common error on the part of the programmer to leave an inadvertent reference to an object that will never be accessed by the program. This causes a memory leak: the object is no longer used by the program, yet is not reclaimed by the JVM.

A common error pattern dubbed the "lapsed listener" bug often arises from the application of the listener pattern. The listener pattern, a specific class of the observer pattern, consists of a subject and an observer. Listeners are commonly used to specify event handlers for events that occur to the object. Some commonly used event types are: GUI events: there are typically dedicated listener classes implementing `java.util.EventListener` interface for GUI events in AWT, Swing, and other GUI libraries. Database connection events; listeners implement `ConnectionEventListener` interface in `javax.sql`. While the listener pattern is widespread, it is error-prone and may cause memory leaks if not used properly. We will demonstrate the leak that may arise by showing how listeners may be implemented internally.

Consider a situation where we register a listener for events happening to a widget in Figure 1. Usually, there is a listener table reachable from the root set. For instance, there could be a global event table pointing to the listener table. In the figure, the listener table it is implemented as a linked list. Event dispatch works by walking through the listener table and invoking an event handler on listeners of the right kind. The listener table has a reference to the listener so that it can deliver an event when it arrives; this reference is shown as a dashed arrow in the figure

However, if that reference is preserved, the object will never be collected, because it is reachable from the root set. In many cases, the object may consume a lot of system resources. It could be a large GUI object utilizing native system resources or a limited resource such as database connections. While leaking listeners, which are usually pretty small objects, may not present an immediate problem, leaking those larger objects reachable from them usually has much more serious consequences leading to deterioration in performance and eventual crashes.

In practice, registering a listener includes adding a reference from the listener to the object (shown in bold). Listener registration is achieved through a call like this: `object.addMyListener(newMyListener(...));`

It is important to remove this reference with a matching call to `object.removeMyListener(l);` thus unregistering the listener and allowing object to be garbage-collected.

A typical large-scale application with a GUI and a database back end may use dozens of different types of listeners, all of which have to be unregistered to avoid memory leaks.

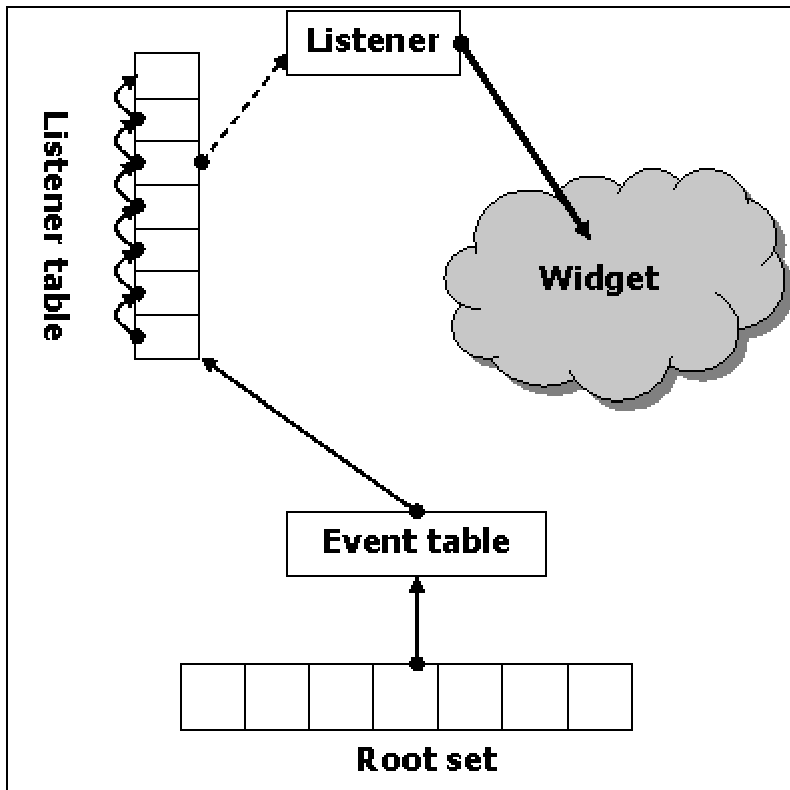

Figure 1: Design of a typical event-based system.

2 Memory Profiling

While lapsed listeners may not be a problem with short-running programs, in long-running applications, continuous memory leakage over a long period of time may lead to memory exhaustion. The Memory profiling feature of Oracle JDeveloper 10g allows the developer to collect memory statistics on long-running programs.

To illustrate the use of memory profiling to look for potential leaks, we created `ListenerTest`, a test program that creates object/listener pairs in a loop that repeats 3,000 times. The structure of the programs matches Figure 1: listeners are accessible from a global listener table and all listeners have a reference to the corresponding object. At the end of each iteration, there is a call to `System.gc()` to initial a round of garbage collection.

JDeveloper's memory profiler is accessible from Run — Memory profile You can configure the frequency of snapshots under

5/5

Class	Count	Size	No. Alloc	Sz Alloc	No. Freed	Sz Freed	Diff Alloc	Diff Sz
Totals:	10044	276,931	544	33,354	378	29,274	166	4,080
java.util.LinkedList$Entry	2279	45,612	42	840	0	0	42	840
int[]	2279	118,980	42	2,184	0	0	42	2,184
ListenerTest$Component	2278	36,484	42	672	0	0	42	672
ListenerTest$1	2278	27,376	42	504	0	0	42	504
char[]	261	22,512	126	17,472	126	17,472	0	0
java.lang.String	256	6,144	42	1,008	42	1,008	0	0
java.lang.StringBuffer	2	34	42	714	42	714	0	0
java.util.HashMap$Entry	35	840	0	0	0	0	0	0
java.net.URL	30	1,676	0	0	0	0	0	0
java.lang.Object[][]	26	644	0	0	0	0	0	0
java.util.Locale	23	580	0	0	0	0	0	0

Detail for class: ListenerTest$Component

Code Position	Count	Size	No. Alloc	Sz Alloc	No. Freed	Sz Freed	Diff Alloc	Diff Sz
ListenerTest.createComponentListenerPair : line 144	2278	36,484	0	0	0	0	0	0

Figure 2: Snapshot 5 output of the memory profiler when run on `ListenerTest`.

```
Tools|Project properties|Profiles|Development|Profiler|Memory
```

In this case we set the update interval to 0.8 seconds resulting in 9 memory snapshots at the end of the run.

The memory profile view shows allocation and garbage collection statistic for each snapshot. One quick way to look for potential memory leaks is to sort by "Diff Alloc", the difference between the number of allocated objects of each type between this and the previous snapshot. Large numbers in this column as well as the "Diff sz" column indicate objects that are being constantly allocated without being deleted, thus pinpointing potential leaks.

To further help in narrowing the problem down, pausing the memory profiling process and double-clicking on the class of interest will also show allocation sites for objects of the type in question. When the source code is available, you may double-click on an allocation site to jump right to the source.

3 Heap Debugging

Knowing that potentially leaking objects were allocated is sometimes not enough, however. In order to get at the source of the leak, the question that needs to be asked is why the garbage collector still keeps these objects around.

To answer that question, we can use Oracle JDeveloper 10g's heap debugging capabilities. Once we have determined which classes are potentially leaking through memory profiling, it is possible to obtain more detailed information by using the Heap window of JDeveloper's debugger. For LapsedListenerTest, the one of the classes that appears to be leaking from the memory profiler view is the inner class `ListenerTest$Component`.

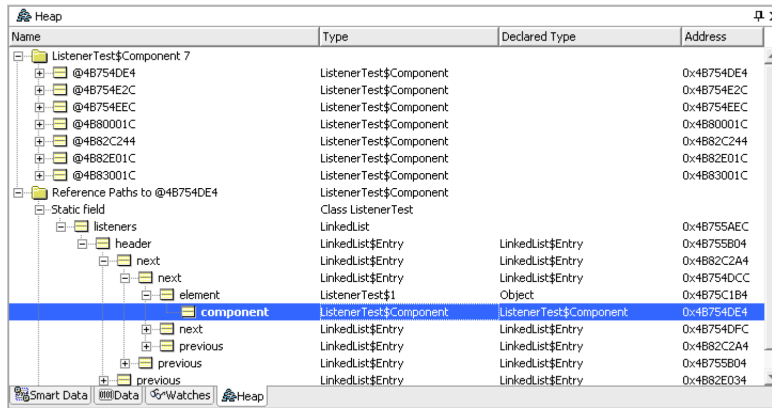

Figure 3: Using the Heap window in JDeveloper to isolate memory leaks.

We ran the test program in debug mode, right-clicked in the Heap window, and selected Add class folder to show all instances of the class we are interested in available at runtime. In Figure 2, there are 7 instances of that class. We first looked for reference paths for a `ListenerTest$Component` object located at `0X4B7B4DE4`. Then we right-clicked on the first of two reference paths corresponding to a static field and selected Expand reference path, which expands the tree view to show how to reach the object in question. As expected, the `ListenerTest$Component` object in question is reachable from the roots through the listener table at `0X48755AEC` and then one of the listeners at `0X4875C1B4`.

4 Getting Rid of the Leaks

There are several ways to get rid of memory leaks once you find them. One approach consists of using `WeakReferences`, which are not traversed by the garbage collector when determining which objects are reachable. Thus, if we make the reference from the listener list to the listener a weak one as shown below, the object of type `ListenerTest$Component` will eventually become unreachable and will be collected:

```
public static class Component {
   ...
public void addListener(ComponentListener listener){
    synchronized (listeners){
        listeners.add(new WeakReference(listener));
    }
```

```
        }
            ...
        }
```

However, is a large system, keeping track of the listeners will be typically done behind the scenes and the developer's only way to get rid of the leak will be to call the remove method on all program paths that register the listener causing the object being listened on to eventually be collected. Conclusions While Java's garbage collector may not solve all memory problems and may leave leaks causing memory exhaustion, Oracle JDeveloper comes fully equipped with tools to address memory problems that might arise. In this article we have shown how to effectively use JDeveloper's memory profiler and heap debugger to locate memory errors.

5 Bio

Benjamin Livshits is a Ph.D. candidate at Stanford University. His interests are in designing tools for automatic error detection for systems written in C and Java. He has lately been focusing on static program analysis for finding security errors in Java applications. Benjamin received a B.S. from Cornell University in 1999 and an MS from Stanford University in 2002.