Java theory and practice: Plugging memory leaks with weak references

Weak references make it easy to express object lifecycle relationships

Brian Goetz

November 22, 2005

While programs in the Java™ language are theoretically immune from "memory leaks," there are situations in which objects are not garbage collected even though they are no longer part of the program's logical state. This month, sanitation engineer Brian Goetz explores a common cause of unintentional object retention and shows how to plug the leak with weak references.

View more content in this series

For garbage collection (GC) to reclaim objects no longer in use by the program, the *logical* lifetime of an object (the time that the application will use it) and the *actual* lifetime of outstanding references to that object must be the same. Most of the time, good software engineering techniques ensure that this happens automatically, without us having to expend a lot of attention on the issue of object lifetime. But every once in a while, we create a reference that holds an object in memory much longer than we expected it to, a situation referred to as *unintentional object retention*.

Memory leaks with global Maps

The most common source of unintentional object retention is the use of a `Map` to associate metadata with transient objects. Let's say you have an object with an intermediate lifetime -- longer than that of the method call that allocated it, but shorter than that of the application -- such as a socket connection from a client. You want to associate some metadata with that socket, such as the identity of the user that has made the connection. You don't know this information at the time the `Socket` is created, and you cannot add data to the `Socket` object because you do not control the `Socket` class or its instantiation. In this case, the typical approach is to store such information in a global `Map`, as shown in the `SocketManager` class in Listing 1:

Listing 1. Using a global Map to associate metadata with an object

```
public class SocketManager {
    private Map<Socket,User> m = new HashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
    public void removeUser(Socket s) {
        m.remove(s);
    }
}

SocketManager socketManager;
...
socketManager.setUser(socket, user);
```

The problem with this approach is that the lifetime of the metadata needs to be tied to the lifetime of the socket, but unless you know exactly when the socket is no longer needed by the program and remember to remove the corresponding mapping from the `Map`, the `Socket` and `User` objects will stay in the `Map` forever, long after the request has been serviced and the socket has been closed. This prevents the `Socket` and `User` objects from being garbage collected, even though the application is never going to use either of them again. Left unchecked, this could easily cause a program to run out of memory if it runs for long enough. In all but the most trivial cases, the techniques for identifying when the `Socket` becomes no longer used by the program resemble the annoying and error-prone techniques required for manual memory management.

Identifying memory leaks

The first sign that your program has a memory leak is usually that it throws an `OutOfMemoryError` or starts to exhibit poor performance because of frequent garbage collection. Fortunately, the garbage collector is willing to share a lot of information that can be used to diagnose a memory leak. If you invoke the JVM with the `-verbose:gc` or the `-Xloggc` option, a diagnostic message is printed on the console or to a log file every time the GC runs, including how long it took, the current heap usage, and how much memory was recovered. Logging GC usage is not intrusive, and so it is reasonable to enable GC logging in production by default in the event you ever need to analyze memory problems or tune the garbage collector.

Tools can take the GC log output and display it graphically; one such tool is the free JTune (see Related topics). By looking at the graph of heap size after GC, you can see the trend of your program's memory usage. For most programs, you can divide memory usage into two components: the *baseline* usage and the *current load* usage. For a server application, the baseline usage is what the application uses when it is not subjected to any load but is ready to accept requests; the current load usage is what is used in the process of handling requests but released when the request processing is complete. So long as load is roughly constant, applications typically reach a steady state level of memory usage fairly quickly. If memory usage continues to trend upwards even though the application has completed its initialization and its load is not increasing, the program is probably retaining objects generated in the course of processing prior requests.

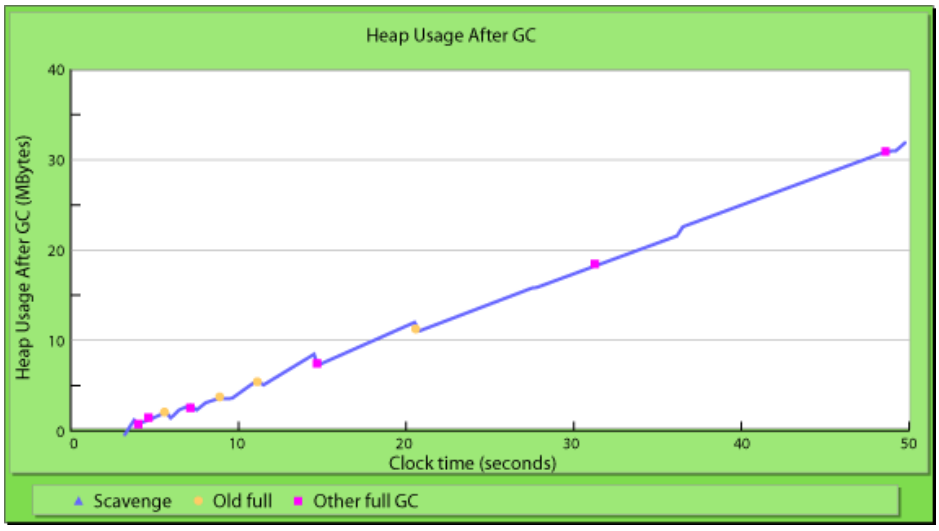
Listing 2 shows a program that has a memory leak. `MapLeaker` processes tasks in a thread pool and records the status of each task in a `Map`. Unfortunately, it never removes the entry when the task is finished, so the status entries and the task objects (along with their internal state) accumulate forever.

Listing 2. Program with a Map-based memory leak

```
public class MapLeaker {
    public ExecutorService exec = Executors.newFixedThreadPool(5);
    public Map<Task, TaskStatus> taskStatus
        = Collections.synchronizedMap(new HashMap<Task, TaskStatus>());
    private Random random = new Random();

    private enum TaskStatus { NOT_STARTED, STARTED, FINISHED };

    private class Task implements Runnable {
        private int[] numbers = new int[random.nextInt(200)];

        public void run() {
            int[] temp = new int[random.nextInt(10000)];
            taskStatus.put(this, TaskStatus.STARTED);
            doSomeWork();
            taskStatus.put(this, TaskStatus.FINISHED);
        }
    }

    public Task newTask() {
        Task t = new Task();
        taskStatus.put(t, TaskStatus.NOT_STARTED);
        exec.execute(t);
        return t;
    }
}
```

Figure 1 shows a graph of the application heap size after GC for `MapLeaker` over time. The upward-sloping trend is a telltale sign of a memory leak. (In real applications, the slope is never this dramatic, but it usually becomes apparent if you gather GC data for long enough.)

Figure 1. Persistent upward memory usage trend



Once you are convinced you have a memory leak, the next step is to find out what type of objects are causing the problem. Any memory profiler can produce snapshots of the heap broken down by object class. There are some excellent commercial heap profiling tools available, but you don't have to spend any money to find memory leaks -- the built-in `hprof` tool can also do the trick. To use `hprof` and instruct it to track memory usage, invoke the JVM with the `-Xrunhprof:heap=sites` option.

Listing 3 shows the relevant portion of the `hprof` output breaking down the application's memory usage. (The `hprof` tool produces a usage breakdown after the application exits, or when you signal your application with a `kill -3` or by pressing Ctrl+Break on Windows.) Notice that there has been significant growth in `Map.Entry`, `Task`, and `int[]` objects between the two snapshots.

See Listing 3.

Listing 4 shows yet another portion of the `hprof` output, giving call stack information for allocation sites for `Map.Entry` objects. This output tells us which call chains are generating the `Map.Entry` objects; with some program analysis, it is usually fairly easy to pinpoint the source of the memory leak.

Listing 4. HPROF output showing allocation site for Map.Entry objects

```
TRACE 300446:
 java.util.HashMap$Entry.<init>(<Unknown Source>:Unknown line)
 java.util.HashMap.addEntry(<Unknown Source>:Unknown line)
 java.util.HashMap.put(<Unknown Source>:Unknown line)
 java.util.Collections$SynchronizedMap.put(<Unknown Source>:Unknown line)
 com.quiotix.dummy.MapLeaker.newTask(MapLeaker.java:48)
 com.quiotix.dummy.MapLeaker.main(MapLeaker.java:64)
```

Weak references to the rescue

The problem with `SocketManager` is that the lifetime of the `Socket`-`User` mapping should be matched to the lifetime of the `Socket`, but the language does not provide any easy means to enforce this rule. This forces the program to fall back on techniques that resemble manual memory management. Fortunately, as of JDK 1.2, the garbage collector provides a means to declare such object lifecycle dependencies so that the garbage collector can help us prevent this sort of memory leak -- with *weak references*.

A weak reference is a holder for a reference to an object, called the *referent*. With weak references, you can maintain a reference to the referent without preventing it from being garbage collected. When the garbage collector traces the heap, if the only outstanding references to an object are weak references, then the referent becomes a candidate for GC as if there were no outstanding references, and any outstanding weak references are *cleared*. (An object that is only referenced by weak references is called *weakly reachable*.)

The referent of a `WeakReference` is set at construction time, and its value, if it has not yet been cleared, can be retrieved with `get()`. If the weak reference has been cleared (either because the referent has already been garbage collected or because someone called `WeakReference.clear()`),

`get()` returns `null`. Accordingly, you should always check that `get()` returns a non-null value before using its result, as it is expected that the referent will eventually be garbage collected.

When you copy an object reference using an ordinary (strong) reference, you constrain the lifetime of the referent to be at least as long as that of the copied reference. If you are not careful, this can be the lifetime of your program -- such as when you place an object in a global collection. On the other hand, when you create a weak reference to an object, you do not extend the lifetime of the referent at all; you simply maintain an alternate way to reach it *while it is still alive*.

Weak references are most useful for building weak collections, such as those that store metadata about objects only for as long as the rest of the application uses those objects -- which is exactly what the `SocketManager` class is supposed to do. Because this is such a common use for weak references, `WeakHashMap`, which uses weak references for keys (but not for values), was also added to the class library in JDK 1.2. If you use an object as a key in an ordinary `HashMap`, that object cannot be collected until the mapping is removed from the `Map`; `WeakHashMap` allows you to use an object as a `Map` key without preventing that object from being garbage collected. Listing 5 shows a possible implementation of the `get()` method from `WeakHashMap`, showing the use of weak references:

Listing 5. Possible implementation of WeakReference.get()

```
public class WeakHashMap<K,V> implements Map<K,V> {

    private static class Entry<K,V> extends WeakReference<K>
      implements Map.Entry<K,V> {
        private V value;
        private final int hash;
        private Entry<K,V> next;
        ...
    }

    public V get(Object key) {
        int hash = getHash(key);
        Entry<K,V> e = getChain(hash);
        while (e != null) {
            K eKey= e.get();
            if (e.hash == hash && (key == eKey || key.equals(eKey)))
                return e.value;
            e = e.next;
        }
        return null;
    }
```

When `WeakReference.get()` is called, it returns a strong reference to the referent (if it is still alive), so there is no need to worry about the mapping disappearing in the body of the `while` loop because the strong reference keeps it from being garbage collected. The implementation of `WeakHashMap` illustrates a common idiom with weak references -- that some internal object extends `WeakReference`. The reason for this becomes clear in the next section when we discuss reference queues.

When you add a mapping to a `WeakHashMap`, remember that it is possible that the mapping could "fall out" later because the key is garbage collected. In that case, `get()` returns `null`, making it even more important than usual to test the return value of `get()` for `null`.

Plugging the leak with WeakHashMap

Fixing the leak in `SocketManager` is easy; simply replace the `HashMap` with a `WeakHashMap`, as shown in Listing 6. (If `SocketManager` needs to be thread-safe, you can wrap the `WeakHashMap` with `Collections.synchronizedMap()`). You can use this approach whenever the lifetime of the mapping must be tied to the lifetime of the key. However, you should be careful not to overuse this technique; most of the time an ordinary `HashMap` is the right `Map` implementation to use.

Listing 6. Fixing SocketManager with WeakHashMap

```
public class SocketManager {
    private Map<Socket,User> m = new WeakHashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
}
```

Reference queues

`WeakHashMap` uses weak references for holding map keys, which allows the key objects to be garbage collected when they are no longer used by the application, and the `get()` implementation can tell a live mapping from a dead one by whether `WeakReference.get()` returns `null`. But this is only half of what is needed to keep a `Map`'s memory consumption from increasing throughout the lifetime of the application; something must also be done to prune the dead entries from the `Map` after the key object has been collected. Otherwise, the `Map` would simply fill up with entries corresponding to dead keys. And while this would be invisible to the application, it could still cause the application to run out of memory because the `Map.Entry` and value objects would not be collected, even if the key is.

Dead mappings could be eliminated by periodically scanning the `Map`, calling `get()` on each weak reference, and removing the mapping if get() returns `null`. But this would be inefficient if the `Map` has many live entries. It would be nice if there was a way to be notified when the referent of a weak reference is garbage collected, and that is what *reference queues* are for.

Reference queues are the garbage collector's primary means of feeding back information to the application about object lifecycle. Weak references have two constructors: one takes only the referent as an argument and the other also takes a reference queue. When a weak reference has been created with an associated reference queue and the referent becomes a candidate for GC, the reference object (not the referent) is *enqueued* on the reference queue after the reference is cleared. The application can then retrieve the reference from the reference queue and learn that the referent has been collected so it can perform associated cleanup activities, such as expunging the entries for objects that have fallen out of a weak collection. (Reference queues offer the same dequeuing modes as `BlockingQueue` -- polled, timed blocking, and untimed blocking.)

`WeakHashMap` has a private method called `expungeStaleEntries()` that is called during most `Map` operations, which polls the reference queue for any expired references and removes the

associated mappings. A possible implementation of `expungeStaleEntries()` is shown in Listing 7. The `Entry` type, which is used to store the key-value mapping, extends `WeakReference`, so when `expungeStaleEntries()` asks for the next expired weak reference, it gets back an `Entry`. Using reference queues to clean up the `Map` instead of periodically trawling through its contents is more efficient because live entries are never touched in the cleanup process; it only does work if there are actually enqueued references.

Listing 7. Possible implementation of WeakHashMap.expungeStaleEntries()

```
   private void expungeStaleEntries() {
Entry<K,V> e;
       while ( (e = (Entry<K,V>) queue.poll()) != null) {
           int hash = e.hash;

           Entry<K,V> prev = getChain(hash);
           Entry<K,V> cur = prev;
           while (cur != null) {
               Entry<K,V> next = cur.next;
               if (cur == e) {
                   if (prev == e)
                       setChain(hash, next);
                   else
                       prev.next = next;
                   break;
               }
               prev = cur;
               cur = next;
           }
       }
   }
```

Conclusion

Weak references and weak collections are powerful tools for heap management, allowing the application to use a more sophisticated notion of reachability, rather than the "all or nothing" reachability offered by ordinary (strong) references. Next month, we'll look at *soft references*, which are related to weak references, and we'll look at the behavior of the garbage collector in the presence of weak and soft references.

Related topics

- JTune: The free JTune tool can take GC logs and graphically display heap size, GC duration, and other useful memory management data.
- "HPROF": This paper from Sun describes how to use the built-in HPROF profiling tool.