Eliminating Obsolete Object References

When you switch from a language with manual memory management like C to a garbage-collected one like Java you might go under the impression that you won’t have to worry about memory management anymore.

Consider the following implementation of a Stack:

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

/\*\*

\* Ensure space for at least one more element, roughly

\* doubling the capacity each time the array needs to grow.

\*/

private void ensureCapacity() {

if (elements.length == size)

elements = Arrays.copyOf(elements, 2 \* size + 1);

}}

There’s nothing obviously wrong with this program (but see Item 29 for a generic version). You could test it exhaustively, and it would pass every test with flying colors, but there’s a problem lurking.

Loosely speaking, the program has a “memory leak,” which can **silently manifest itself as**:

Read about these problems a little more:

* **reduced performance due to increased garbage collector activity or increased memory footprint.**
* **In extreme cases, such memory leaks can cause disk paging and even program failure with an OutOfMemoryError, but such failures are relatively rare.**

**So where is the memory leak?**

If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains *obsolete references* to these objects.

**An obsolete reference is simply a reference that will never be dereferenced again.** In this case, any references outside of the ”active portion” of the element array are obsolete.

The active portion consists of the elements whose index is less than size. Memory leaks in garbage-collected languages (more properly known as *unintentional object retentions*) are insidious.

*I add: you have a reference to an object, in this case elements[size] for example, but this reference hence the object it’s pointing to is never reached.*

**If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on. Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.**

The fix for this sort of problem is simple: **null out references once they become obsolete**. In the case of our Stack class, the reference to an item becomes obsolete as soon as it’s popped off the stack. The corrected version of the pop method looks like this:

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

**An added benefit of nulling out obsolete references is that if they are subsequently dereferenced by mistake, the program will immediately fail with a NullPointerException**, rather than quietly doing the wrong thing. It is always beneficial to detect programming errors as quickly as possible.

When programmers are first stung by this problem, they may overcompensate by nulling out every object reference as soon as the program is finished using it.

This is neither necessary nor desirable; it clutters up the program unnecessarily.

**Nulling out object references should be the exception rather than the norm.**

**The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope.** **This occurs naturally if you define each variable in the narrowest possible scope**(Item 57)

Let’s talk about when to be worried about memory leaks happening and how to prevent them.

# What Causes Memory Leaks and How to Prevent them?

## When a Class Manages Its Own Memory

**So, when should you null out a reference? What aspect of the Stack class makes it susceptible to memory leaks?**

Simply put, it *manages its own memory*.

The *storage pool* consists of the elements of the elements array (the object reference cells, not the objects themselves). The elements in the active portion of the array (as defined earlier) are *allocated*, and those in the remainder of the array are *free*.

**The garbage collector has no way of knowing this; to the garbage collector, all of the object references in the elements array are equally valid.**

Only the programmer knows that the inactive portion of the array is unimportant.

The programmer effectively **communicates this fact to the garbage collector by manually nulling out array elements as soon as they become part of the inactive portion.**

**Generally speaking, whenever a class manages its own memory, the programmer should be alert for memory leaks. Whenever an element is freed, any object references contained in the element should be nulled out.**

## Caches

Another common source of memory leaks is caches. Once you put an object reference into a cache, it’s easy to forget that it’s there and leave it in the cache long after it becomes irrelevant.

There are several solutions to this problem:

* If you’re lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a **WeakHashMap**; entries will be removed automatically after they become obsolete.

**Remember that WeakHashMap is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.**

* More commonly, the useful lifetime of a cache entry is less well defined, with entries becoming less valuable over time. Under these circumstances, **the cache should occasionally be cleansed of entries that have fallen into disuse.**

This can be done by:

* + a background thread (perhaps a **ScheduledThreadPoolExecutor**)
  + or as a side effect of adding new entries to the cache. The **LinkedHashMap** class facilitates the latter approach with its **removeEldestEntry** method.

**For more sophisticated caches, you may need to use java.lang.ref directly.**

## Listeners and Other Callbacks

If you implement an API where clients register callbacks but don’t deregister

them explicitly, they will accumulate unless you take some action.

One way to ensure that callbacks are garbage collected promptly is to store only weakreferencesto them, for instance, by storing them only as keys in a WeakHashMap.

I add: *or maybe just use a list of weak references to them? Anyways Can you think of a real world scenario where this item (1.3) can be useful?*

Because memory leaks typically do not manifest themselves as obvious failures, they may remain present in a system for years.

They are typically discovered only as a result of:

* careful code inspection
* or with the aid of a debugging tool known as a *heap profiler*.

Therefore, it is very desirable to learn to anticipate problems like this before they occur and prevent them from happening.