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.