JAVAWORLD

NEWS

Java memory management

Are memory leaks possible, preventable in Java?

By Java Q&a Experts

JavaWorld |

AUG 27, 1999 2:00 AM PT

Q: What sort of memory leaks are possible in Java and how do I prevent them?

A: In the case of incomplete deallocation, there are two subcases: coding bugs and design bugs. Coding bugs are language dependent. In the case of C, this would involve `free()`ing less than was `malloc()`ed, while in C++ this might involve using `delete` in lieu of `delete[]`. Design bugs, on the other hand, do not depend on the language; instead, they involve simple programmer negligence.

In languages like C/C++, all memory management is handled by the programmer, so all of these problems can arise, even after the programmer has expended much effort to ensure the code is free of such defects. In fact, in C/C++ the more you try to avoid memory leaks, the more likely you are to create corrupted pointers, and vice versa. And, by nature the risk of such bugs increases with code size and complexity, so it's difficult to protect large C/C++ applications from these types of bugs.

In Java, on the other hand, the Java language and runtime together entirely eliminate the problems of corrupted pointers and code-level memory leaks. Here's how:

- In Java, memory is allocated only to objects. There is no explicit allocation of memory, there is only the creation of new objects. (Java even treats array types as objects.)

- The Java runtime employs a garbage collector that reclaims the memory occupied by an object once it determines that object is no longer accessible. This automatic process makes it safe to throw away unneeded object references because the garbage collector does not collect the object if it is still needed elsewhere. Therefore, in Java the act of letting go of unneeded references never runs the risk of deallocating memory prematurely.
- In Java, it's easy to let go of an entire "tree" of objects by setting the reference to the tree's root to `null`; the garbage collector will then reclaim all the objects (unless some of the objects are needed elsewhere). This is a lot easier than coding each of the objects' destructors to let go of its own dependencies (which is a coding-level problem with C++).

So what about memory leaks caused by poor program design? In such designs, unnecessary object references originating in long-lived parts of the system prevent the garbage collector from reclaiming objects that are in fact no longer needed. Such errors typically involve a failure "in the large" to properly encapsulate object references among various parts of the code.

Another design flaw occurs "in the small," at the level of a faulty algorithm; the use of `Collections` objects in such cases will typically magnify the error.

As the technology improves, a suitably implemented JVM could help reduce the effects of such designed-in memory leaks by using the garbage collector to track object usage over time. The garbage collector could then rearrange objects in memory according to a freshness factor based on when they were last referenced, for example. Stale objects would become eligible for physical RAM swap-out (even though, for safety, they would still exist).

Ideally of course, programmers would design applications with objects' lifecycles in mind, rather than rely on clever features of state-of-the-art JVM implementations.

In conclusion: Design problems can be mitigated by letting go of object references in one's own classes as soon as one can (knowing that in Java there is no risk of damaging another part of the code).

Tip: The KL Group's JProbe tool can help identify possible sources of memory leaks by showing which objects are holding on to graphs of references to other objects.

Note: Implementations of Sun's JVM prior to HotSpot used a technique called *conservative garbage collection,* which could introduce memory leaks of its own, beyond those caused by the programmer. To wit, as long as the application contained a 32-bit integer primitive (int) whose value coincided with the address of an object, the garbage collector would not reclaim the object. Although the likelihood was small, it wasn't zero. For this reason, and for the best performance in general, use HotSpot where possible.

Random Walk Computing is the largest Java/CORBA consulting boutique in New York, focusing on solutions for the financial enterprise. Known for their leading-edge Java expertise, Random Walk consultants publish and speak about Java in some of the most respected forums in the world.

Learn more about this topic

- The KL Group's home page
  http://www.klgroup.com/ (http://www.klgroup.com/)
- Sun's Java HotSpot Performance Engine home page
  http://java.sun.com/products/hotspot/index.html (http://java.sun.com/products/hotspot/index.html)
- **Related reading in JavaWorld**
- "Can HotSpot jumpstart your Java applications?" Steven Brody (June 1999). A look at the HotSpot JVM, including it's garbage collection features
  http://www.javaworld.com/javaworld/jw-06-1999/jw-06-hotspot.html (http://www.javaworld.com/javaworld/jw-06-1999/jw-06-hotspot.html)
- "Object finalization and cleanup," Bill Venners (June 1998). Covers how to design classes for proper object cleanup
  http://www.javaworld.com/jw-06-1998/jw-06-techniques.html (http://www.javaworld.com/jw-06-1998/jw-06-techniques.html)
- "Build your own ObjectPool in Java to boost app speed," Thomas E. Davis (June 1998). Explains how to use object pooling to share instantiated objects, improving speed while reducing memory requirements
  http://www.javaworld.com/jw-06-1998/jw-06-object-pool.html (http://www.javaworld.com/jw-06-1998/jw-06-object-pool.html)
- "Not using garbage collection," Chuck McManis (September 1996). Discusses how to minimize heap thrashing in your Java programs
  http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html (http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html)
- "Under the hoodJava's garbage-collected heap," Bill Venners (August 1996)provides an introduction to garbage collection
  http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html (http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html)