DESIGN TECHNIQUES

By Bill Venners, JavaWorld

JUN 1, 1998 12:00 AM PST

Object finalization and cleanup

How to design classes for proper object cleanup

Three months ago, I began a mini-series of articles about designing objects with a discussion of design principles that focused on proper initialization at the beginning of an object's life. In this **Design Techniques** article, I'll be focusing on the design principles that help you ensure proper cleanup at the end of an object's life.

Why clean up?

Every object in a Java program uses computing resources that are finite. Most obviously, all objects use some memory to store their images on the heap. (This is true even for objects that declare no instance variables. Each object image must include some kind of pointer to class data, and can include other implementation-dependent information as well.) But objects may also use other finite resources besides memory. For example, some objects may use resources such as file handles, graphics contexts, sockets, and so on. When you design an object, you must make sure it eventually releases any finite resources it uses so the system won't run out of those resources.

Because Java is a garbage-collected language, releasing the memory associated with an object is easy. All you need to do is let go of all references to the object. Because you don't have to worry about explicitly freeing an object, as you must in languages such as C or C++, you needn't worry about corrupting memory by accidentally freeing the same object twice. You do, however, need to make sure you actually release all references to the object. If you don't, you can end up with a memory leak, just like the memory leaks you get in a C++ program when you forget to explicitly free objects. Nevertheless, so long as you release all references to an object, you needn't worry about explicitly "freeing" that memory.

Similarly, you needn't worry about explicitly freeing any constituent objects referenced by the instance variables of an object you no longer need. Releasing all references to the unneeded object will in effect invalidate any constituent object references contained in that object's instance variables. If the now-invalidated references were the only remaining references to those constituent objects, the constituent objects will also be available for garbage collection. Piece of cake, right?

The rules of garbage collection

Although garbage collection does indeed make memory management in Java a lot easier than it is in C or C++, you aren't able to completely forget about memory when you program in Java. To know when you may need to think about memory management in Java, you need to know a bit about the way garbage collection is treated in the Java specifications.

Garbage collection is not mandated

The first thing to know is that no matter how diligently you search through the Java Virtual Machine Specification (JVM Spec), you won't be able to find any sentence that commands, *Every JVM must have a garbage collector.* The Java Virtual Machine Specification gives VM designers a great deal of leeway in deciding how their implementations will manage memory, including deciding whether or not to even use garbage collection at all. Thus, it is possible that some JVMs (such as a bare-bones smart card JVM) may require that programs executed in each session "fit" in the available memory.

Of course, you can always run out of memory, even on a virtual memory system. The JVM Spec does not state how much memory will be available to a JVM. It just states that

SponsoredPost Sponsored by NTT DATA Services
Don't Leave Your Cloud Migration to Chance

Nevertheless, to give Java applications the best chance of executing without running out of memory, most JVMs will use a garbage collector. The garbage collector reclaims the memory occupied by unreferenced objects on the heap, so that memory can be used again by new objects, and usually de-fragments the heap as the program runs.

Garbage collection algorithm is not defined

Another command you won't find in the JVM specification is *All JVMs that use garbage collection must use the XXX algorithm.* The designers of each JVM get to decide how garbage collection will work in their implementations. Garbage collection algorithm is one area in which JVM vendors can strive to make their implementation better than the competition's. This is significant for you as a Java programmer for the following reason:

Because you don't generally know how garbage collection will be performed inside a JVM, you don't know when any particular object will be garbage collected.

So what? you might ask. The reason you might care when an object is garbage collected has to do with finalizers. (A *finalizer* is defined as a regular Java instance method named `finalize()` that returns void and takes no arguments.) The Java specifications make the following promise about finalizers:

Before reclaiming the memory occupied by an object that has a finalizer, the garbage collector will invoke that object's finalizer.

Given that you don't know when objects will be garbage collected, but you do know that finalizable objects will be finalized as they are garbage collected, you can make the

You don't know when objects will be finalized.

You should imprint this important fact on your brain and forever allow it to inform your Java object designs.

Finalizers to avoid

The central rule of thumb concerning finalizers is this:

Don't design your Java programs such that correctness depends upon "timely" finalization.

In other words, don't write programs that will break if certain objects aren't finalized by certain points in the life of the program's execution. If you write such a program, it may work on some implementations of the JVM but fail on others.

Don't rely on finalizers to release non-memory resources

An example of an object that breaks this rule is one that opens a file in its constructor and closes the file in its `finalize()` method. Although this design seems neat, tidy, and symmetrical, it potentially creates an insidious bug. A Java program generally will have only a finite number of file handles at its disposal. When all those handles are in use, the program won't be able to open any more files.

A Java program that makes use of such an object (one that opens a file in its constructor and closes it in its finalizer) may work fine on some JVM implementations. On such whose garbage collector doesn't finalize often enough to keep the program from running out of file handles. Or, what's even more insidious, the program may work on all JVM implementations now but fail in a mission-critical situation a few years (and release cycles) down the road.

Other finalizer rules of thumb

Two other decisions left to JVM designers are selecting the thread (or threads) that will execute the finalizers and the order in which finalizers will be run. Finalizers may be run in any order -- sequentially by a single thread or concurrently by multiple threads. If your program somehow depends for correctness on finalizers being run in a particular order, or by a particular thread, it may work on some JVM implementations but fail on others.

You should also keep in mind that Java considers an object to be finalized whether the `finalize()` method returns normally or completes abruptly by throwing an exception. Garbage collectors ignore any exceptions thrown by finalizers and in no way notify the rest of the application that an exception was thrown. If you need to ensure that a particular finalizer fully accomplishes a certain mission, you must write that finalizer so that it handles any exceptions that may arise before the finalizer completes its mission.

One more rule of thumb about finalizers concerns objects left on the heap at the end of the application's lifetime. By default, the garbage collector will not execute the finalizers of any objects left on the heap when the application exits. To change this default, you must invoke the `runFinalizersOnExit()` method of class `Runtime` or `System`, passing `true` as the single parameter. If your program contains objects whose finalizers must absolutely be invoked before the program exits, be sure to invoke `runFinalizersOnExit()` somewhere in your program.

So what are finalizers good for?

By now you may be getting the feeling that you don't have much use for finalizers. While it is likely that most of the classes you design won't include a finalizer, there are some reasons to use finalizers.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.