Java theory and practice: Good housekeeping practices

Are your resources overstaying their welcome?

Level: Introductory

Brian Goetz (brian@quiotix.com), Principal Consultant, Quiotix

21 Mar 2006

Garbage collection is nearly everyone's favorite feature of the Java™ platform; it simplifies development and eliminates entire categories of potential code errors. But while garbage collection generally allows you to ignore resource management, sometimes you have to do some housekeeping on your own. In this month's *Java theory and practice*, Brian Goetz discusses the limitations of garbage collection and identifies situations when you have to do your own housecleaning.

Our parents used to remind us to put our toys away when we were done with them. If you look closely enough, the motivation for such nagging was probably not so much an abstract desire to keep things clean as much as the practical limitation that there is only so much floor space in the house, and if it is covered with toys, it can't be used for other things -- like walking around.

Given enough space, the motivation to clean up one's mess is lessened. The more space you have, the less motivation you have to always keep it clean. Arlo Guthrie's famous ballad *Alice's Restaurant Massacre* illustrates this point:

> Havin' all that room, seein' as how they took out all the pews, they decided that they didn't have to take out their garbage ... for a long time.

For better or worse, garbage collection can make us a little sloppy about cleaning up after ourselves.

Explicitly releasing resources

The vast majority of resources used in Java programs are objects, and garbage collection does a fine job of cleaning them up. Go ahead, use as many `String`s as you want. The garbage collector eventually figures out when they've outlived their usefulness, with no help from you, and reclaims the memory they used.

On the other hand, nonmemory resources like file handles and socket handles must be explicitly released by the program, using methods with names like `close()`, `destroy()`, `shutdown()`, or `release()`. Some classes, such as the file handle stream implementations in the platform class library, provide finalizers as a "safety net" so that if the program forgets to release the resource, the finalizer can still do the job when the garbage collector determines that the program is finished with it. But even though file handles provide finalizers to clean up after you if you forget, it is still better to close them explicitly when you are done with them. Doing so closes them much earlier than they otherwise would be, reducing the chance of resource exhaustion.

For some resources, waiting until finalization to release them is not an option. For virtual resources like lock acquisitions and semaphore permits, a `Lock` or `Semaphore` is not likely to get garbage collected until it is too late; for resources like database connections, you would surely run out of resources if you waited for finalization. Many database servers only accept a certain number of connections, based on licensed capacity. If a server application were to open a new database connection for each request and then just drop it on the floor when done, the database would likely reach its capacity long before the no-longer-needed connections were

closed by the finalizer.

Resources confined to a method

Most resources are not held for the lifetime of the application; instead, they are acquired for the lifetime of an activity. When an application opens a file handle to read in so it can process a document, it typically reads from the file and then has no further need for the file handle.

In the easiest case, the resource is acquired, used, and hopefully released in the same method call, such as the `loadPropertiesBadly()` method in Listing 1:

Listing 1. Incorrectly acquiring, using, and releasing a resource in a single method -- don't do this

```
public static Properties loadPropertiesBadly(String fileName)
        throws IOException {
    FileInputStream stream = new FileInputStream(fileName);
    Properties props = new Properties();
    props.load(stream);
    stream.close();
    return props;
}
```

Unfortunately, this example has a potential resource leak. If all goes well, the stream will be closed before the method returns. But if the `props.load()` method throws an `IOException`, then the stream will not be closed (until the garbage collector runs its finalizer). The solution is to use the try...finally mechanism to ensure that the stream is closed no matter what goes wrong, as shown in Listing 2:

Listing 2. Correctly acquiring, using, and releasing a resource in a single method

```
public static Properties loadProperties(String fileName)
        throws IOException {
    FileInputStream stream = new FileInputStream(fileName);
    try {
        Properties props = new Properties();
        props.load(stream);
        return props;
    }
    finally {
        stream.close();
    }
}
```

Note that the resource acquisition (opening the file) is outside the try block; if it were placed inside the try block, then the finally block would run even if resource acquisition threw an exception. Not only would this approach be inappropriate (you can't release a resource you haven't acquired), but the code in the finally block is then likely to throw an exception of its own, such as `NullPointerException`. An exception thrown from a finally block supersedes the exception that caused the block to exit, which means the original exception is lost and cannot be used to aid in the debugging effort.

Not always as easy as it looks

Using `finally` to release resources acquired in a method is reliable but can easily get unwieldy when multiple resources are involved. Consider a method that uses a JDBC `Connection` to execute a query and iterate the `ResultSet`. It acquires a `Connection`, uses it to create a `Statement`, and executes the `Statement` to yield a `ResultSet`. But the intermediate JDBC objects `Statement` and `ResultSet` have

`close()` methods of their own, and they should be released when you are done with them. However, the "obvious" way to clean up, shown in Listing 3, doesn't work:

Listing 3. Unsuccessful attempt to release multiple resources -- don't do this

```
public void enumerateFoo() throws SQLException {
    Statement statement = null;
    ResultSet resultSet = null;
    Connection connection = getConnection();
    try {
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM Foo");
        // Use resultSet
    }
    finally {
        if (resultSet != null)
            resultSet.close();
        if (statement != null)
            statement.close();
        connection.close();
    }

}
```

The reason this "solution" doesn't work is that the `close()` methods of `ResultSet` and `Statement` can themselves throw `SQLException`, which could cause the later `close()` statements in the finally block not to execute. That leaves you with several choices, all of which are annoying: wrap each `close()` with a `try..catch` block, nest the `try...finally` blocks as shown in Listing 4, or write some sort of mini-framework for managing the resource acquisition and release.

Listing 4. Reliable (if unwieldy) means of releasing multiple resources

```
public void enumerateBar() throws SQLException {
    Statement statement = null;
    ResultSet resultSet = null;
    Connection connection = getConnection();
    try {
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM Bar");
        // Use resultSet
    }
    finally {
        try {
            if (resultSet != null)
                resultSet.close();
        }
        finally {
            try {
                if (statement != null)
                    statement.close();
            }
            finally {
                connection.close();
            }
        }
    }
}

private Connection getConnection() {
    return null;
}
```

Nearly everything can throw an exception

We all know that we should use `finally` to release heavyweight objects like database connections, but we're not always so careful about using it to close streams (after all, the finalizer will get that for us, right?). It's also

easy to forget to use `finally` when the code that uses the resource doesn't throw checked exceptions. Listing 5 shows the implementation of the `add()` method for a bounded collection that uses `Semaphore` to enforce the bound and efficiently allow clients to wait for space to become available:

Listing 5. Vulnerable implementation of a bounded collection -- don't do this

```
public class LeakyBoundedSet<T> {
    private final Set<T> set = ...
    private final Semaphore sem;

    public LeakyBoundedSet(int bound) {
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = set.add(o);
        if (!wasAdded)
            sem.release();
        return wasAdded;
    }
}
```

`LeakyBoundedSet` first waits for a permit to be available (indicating that there is space in the collection), then tries to add the element to the collection. If the add operation fails because the element was already in the collection, it releases the permit (because it did not actually use the space it had reserved).

The problem with `LeakyBoundedSet` doesn't necessarily jump out immediately: What if `Set.add()` throws an exception? This scenario could happen because of a flaw in the `Set` implementation, or a flaw in the `equals()` or `hashCode()` implementation (or the `compareTo()` implementation, in the case of a `SortedSet`) for the element being added, or an element already in the `Set`. The solution, of course, is to use `finally` to release the semaphore permit; an easy enough -- but all-too-often-forgotten -- approach. These types of mistakes are rarely disclosed during testing, making them time bombs waiting to go off. Listing 6 shows a more reliable implementation of `BoundedSet`:

Listing 6. Using a Semaphore to reliably bound a Set

```
public class BoundedSet<T> {
    private final Set<T> set = ...
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        }
        finally {
            if (!wasAdded)
                sem.release();
        }
    }
}
```

Code auditing tools like FindBugs (see Resources) can detect some instances of improper resource release, such as opening a stream in a method and not closing it.

Resources with arbitrary lifecycles

For resources with arbitrary lifecycles, we're back to where we were with C -- managing resource lifecycles manually. In a server application where clients make a persistent network connection to the server for the duration of a session (like a multiplayer game server), any resources acquired on a per-user basis (including the socket connection) must be released when the user logs out. Good organization can help; if the sole reference to per-user resources is held in an ActiveUser object, they can be released when the ActiveUser is released (whether explicitly or through garbage collection).

Resources with arbitrary lifecycles are almost certainly going to be stored in (or reachable from) a global collection somewhere. To avoid resource leaks, it is therefore critical to identify when the resource is no longer needed and remove it from this global collection. (A previous article, "Plugging memory leaks with weak references," offers some helpful techniques.) At this point, because you know the resource is about to be released, any nonmemory resources associated with the resource can also be released at this time.

Resource ownership

A key technique for ensuring timely resource release is to maintain a strict hierarchy of ownership; with ownership comes the responsibility to release the resource. If an application creates a thread pool and the thread pool creates threads, the threads are resources that must be released (allowed to terminate) before the program can exit. But the application doesn't own the threads; the thread pool does, and therefore the thread pool must take responsibility for releasing them. Of course, it can't release them until the thread pool itself is released by the application.

Maintaining an ownership hierarchy, where each resource owns the resources it acquires and is responsible for releasing them, helps keep the mess from getting out of control. A consequence of this rule is that each resource that cannot be released solely by garbage collection, which includes any resource that directly or indirectly owns a resource that cannot be released solely by garbage collection, must provide some sort of lifecycle support, such as a `close()` method.

Finalizers

If the platform libraries provide finalizers for cleaning up open file handlers, which greatly reduces the risk of forgetting to close them explicitly, why aren't finalizers used more often? There are a number of reasons, foremost of which is that finalizers are very tricky to write correctly (and very easy to write incorrectly). Not only is it difficult to code them correctly, but the timing of finalization is not deterministic, and there is no guarantee that finalizers will ever even run. And finalization adds overhead to instantiation and garbage collection of finalizable objects. Don't rely on finalizers as the primary means of releasing resources.

Summary

Garbage collection does an awful lot of the cleanup for us, but some resources still require explicit release, such as file handles, socket handles, threads, database connections, and semaphore permits. We can often get away with using `finally` blocks to release a resource if its lifetime is tied to that of a specific call frame, but longer-lived resources require a strategy for ensuring their eventual release. For any object that may directly or indirectly own an object that requires explicit release, you must provide lifecycle methods -- `close()`, `release()`, `destroy()`, and the like -- to ensure reliable cleanup.

Resources

Learn

- *Concurrent Programming in Java* (Doug Lea, Addison-Wesley, 1999): Section 3.4.1.3 of Doug Lea's comprehensive work examines the process of using semaphores to bound collections.
- "Plugging memory leaks with weak references" (Brian Goetz, developerWorks, November 2005): Brian discusses how weak references make it easy to express object lifecycle relationships.
- "Finalization, Threads, and the Java Memory Model" (Sun Developer Network, Hans Boehm): Learn just how ugly finalizers can be.
- *Java theory and practice*: The complete series by Brian Goetz.
- The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- Alice's Restaurant (Warner Brothers, 1969): Learn all the words to Arlo Guthrie's classic folk anthem "Alice's Restaurant Massacre" from the movie soundtrack.
- FindBugs: This free code auditing tool can find unreleased resources and other bugs in your programs.

Discuss

- Participate in the discussion forum.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Brian Goetz has been a professional software developer for over 18 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. Brian's book, *Java Concurrency In Practice*, will be published in early 2006 by Addison-Wesley. See Brian's published and upcoming articles in popular industry publications.
