Neil Richards – IBM Senior Java Engineer, OpenJDK Committer
October 2012

Resource Management in Java: Tips, Traps & Techniques

Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Your speaker - a short introduction ...

- Hello, I'm Neil Richards
  - IBM: neil_richards@uk.ibm.com
  - OpenJDK: neil.richards@ngmr.net
- Developing and deploying Java SDKs for 15 years
- Previous IBM technical component lead:
  - IO
  - Networking
  - ORB, CORBA, RMI-IIOP
- Currently working for IBM in OpenJDK
  - OpenJDK 8
  - OpenJDK 7u
  - Common VM Interface
  - AIX/PPC porting

So, what shall we talk about?

- What resources need managing ?
- Definitions
- Explore some approaches ...
  - "Passive", "Active" & "Hybrid"
  - How to use
  - Potential pitfalls (aka "Traps")
  - Worked examples
  - Limitations
- Recommendations

What resources need managing ?

	C/C++	Java
Stack memory	Automatic	Automatic*
Heap memory	Manual	Automatic**
File descriptor	Manual	Manual
Network socket	Manual	Manual
Database connection	Manual	Manual
etc.	Manual	Manual

* NB: No stack-based object allocation in Java (except via JIT magic)

** NB: C/C++ heap memory management in a Java process is still manual

Definitions

"When I use a word,"
Humpty Dumpty said, in a rather scornful tone,
"it means just what I choose it to mean -
neither more nor less."

Lewis Carroll

- "Passive" management:
  - Places no requirements on the end user (developer) as to how the Java objects representing the resource are handled.
- "Active" management:
  - Places requirements on the end user (developer) as to how the Java objects representing the resource are handled.

Passive approach #1 - Finalization

Finalizers provide a chance to free up resources that cannot be freed automatically by an automatic storage manager.

♦

Before the storage for an object is reclaimed by the garbage collector, the Java virtual machine will invoke the finalizer of that object.

JLS, Java SE 7 Edition, section 12.6

- Override: `protected void Object.finalize() throws Throwable;`

Passive approach #1 – Finalization: Traps (part 1)

The Java programming language imposes no ordering on `finalize` method calls. Finalizers may be called in any order, or even concurrently.

♦

If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

♦

Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.
`<snip>` This should always be done …

♦

A finalizer may be invoked explicitly, just like any other method.

JLS, Java SE 7 Edition, section 12.6

- Usage usually gets exposed / baked into the class' API

Passive approach #1 – Finalization: Traps (part 2)

- Only implementable by developer of the resource class
- In `finalize()`, developer still has access to the object ('`this`')
  - –> Has access to other objects referenced by '`this`'
    - already finalized ?
    - in the midst of being finalized ?
    - scheduled to be (unavoidably) finalized ?
  - –> Lots of temptation / possibility to use “stale” data, objects
    - or even try to “resurrect” them ...

- `finalize()` might be called on an object multiple times …
  - – but only **once** from finalization!

Passive approach #1 – Finalization: example

```
public class MyResource1 {
    private final long handle;
    private final java.util.concurrent.atomic.AtomicBoolean closed = new java.util.concurrent.atomic.AtomicBoolean();

    public MyResource1(/*params*/) throws Throwable {
        handle = allocNativeResource(/*params*/);
    }

    protected void finalize() throws Throwable {
        try {
            Throwable saved = null;
            try {
                if (closed.compareAndSet(false, true)) freeNativeResource(handle);
            } catch (Exception e) {
                saved = e;
            } finally {
                try {
                    super.finalize();
                } catch (Throwable t) {
                    if (saved == null) throw t;
                    saved.addSuppressed(t);
                }
                if (saved != null) throw saved;
            }
        } catch (Throwable t2) {
            t2.printStackTrace();
            throw t2;
        }
    }

    private static native long allocNativeResource(/*params*/) throws Exception;
    private static native void freeNativeResource(long handle) throws Exception;
}
```

Passive approach #2 – java.lang.ref.PhantomReference

Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed.

♦

Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.

Java 7 API Javadoc

- Create 2nd "Reference" class, subclassing PhantomReference
  - Used for tracking main "Resource" objects
- Create statically held ReferenceQueue object
  - Source of "stale" reference objects to be processed
  - "stale" references removed from queue by:
    - calling `poll()`, usually before each operation
    - calling `remove()`, blocking on dedicated thread

Passive approach #2 – PhantomReferences: Traps

If the garbage collector determines at a certain point in time that the referent of a phantom reference is phantom reachable, then at that time **or at some later time** it will enqueue the reference.

♦

The `get` method of a phantom reference always returns `null`.

♦

An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable.

Java 7 API Javadoc

- Strong reference must be maintained to the Reference objects
  - Must be freed once stale reference is processed

Passive approach #2 – PhantomReferences: Benefits

The `get` method of a phantom reference always returns `null`.

Java 7 API Javadoc

- No interference with finalization / PhantomReference use in superclasses (and elsewhere)
  –> Avoids complicated coding for exception handling
- Possibility of implementing PhantomReferences “external” to resource being tracked
  – Will examine this more later ...

Passive approach #2 – PhantomReference: example (part 1)

```
public class MyResource2 {
    private static class MyResourceRef extends java.lang.ref.PhantomReference<MyResource2> {
        private static final java.util.Set<MyResourceRef> activeRefs = new java.util.HashSet<>();
        private static final java.lang.ref.ReferenceQueue<MyResource2> staleRefs = new java.lang.ref.ReferenceQueue<>();
        final long handle;

        MyResourceRef(MyResource2 myRes /*, params*/) throws Exception {
            super(myRes, staleRefs);
            handle = allocNativeResource(/*params*/);

            synchronized (activeRefs) {
                activeRefs.add(this);
            }
        }

        private boolean deactivate() {
            synchronized (activeRefs) {
                return activeRefs.remove(this);
            }
        }

        final void close() throws Exception {
            if (deactivate()) freeNativeResource(handle);
        }

        private static native long allocNativeResource(/*params*/) throws Exception;
        private static native void freeNativeResource(long handle) throws Exception;

        static void processQ() { /*...*/ }
    }

    /*...*/
}
```

Passive approach #2 – PhantomReference: example (part 2)

```
public class MyResource2 {
    private static class MyResourceRef extends java.lang.ref.PhantomReference<MyResource2> {
        private static final java.util.Set<MyResourceRef> activeRefs = new java.util.HashSet<>();
        private static final java.lang.ref.ReferenceQueue<MyResource2> staleRefs = new java.lang.ref.ReferenceQueue<>();
        final long handle;

        /*...*/

        static void processQ() {
            MyResourceRef staleRef = null;
            do {
                synchronized (staleRefs) {
                    staleRef = (MyResourceRef)staleRefs.poll();
                }
                if (staleRef == null) break;
                try {
                    staleRef.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } while (true);
        }
    }

    private final MyResourceRef ref;

    public MyResource2(/*params*/) throws Exception {
        ref = new MyResourceRef(this /*, params*/);
    }

    public void doSomething() {
        MyResourceRef.processQ();
        /* do something with 'handle' */
    }
}
```

Passive approach #2 – PhantomReference: example2

```
public class MyResource2v2 {
    private static class MyResourceRef extends java.lang.ref.PhantomReference<MyResource2v2> {
        /*...*/

        MyResourceRef(MyResource2v2 myRes /*, params*/) throws Exception {
            super(myRes, Reaper.staleRefs);
            /*...*/
        }

        //processQ() no longer needed

        private static final class Reaper extends Thread {
            static final Reaper thread = new Reaper();
            static final java.lang.ref.ReferenceQueue<MyResource2v2> staleRefs = new java.lang.ref.ReferenceQueue<>();

            static {
                thread.setDaemon(true);
                thread.start();
            }

            public void run() {
                do {
                    try {
                        MyResourceRef staleRef = (MyResourceRef)staleRefs.remove();
                        staleRef.close();
                    } catch (InterruptedException ie) {
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                } while (true);
            }
        }
    }
}
```

Passive approaches: General limitations

- Resources can only found as “stale” once GC occurs
  - – GC might be “conservative”
  - –> No guarantee stale resources are found even after GC
- GC generally triggered by constraint of Java heap memory (i.e. another resource type), so:
  - – Larger Java heap
  - –> Longer wait for other resources to be closed
  - –> Greater risk of OOME due to other resource exhaustion
- GC releasing the memory of resource Java objects delayed
  - – Finalizable objects processed by finalization thread(s)
  - – Reference removed from ReferenceQueue, processed
  - –> More constrained Java heap usage
  - –> More time spent performing GC

Active approach #1 – Management by fiat

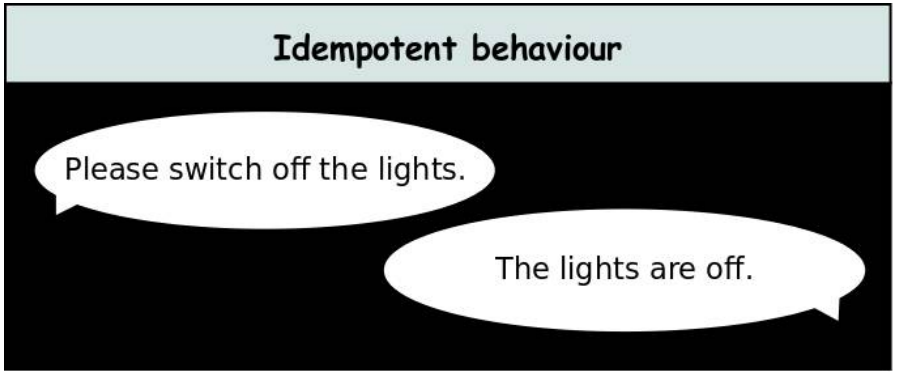
fi•at n.

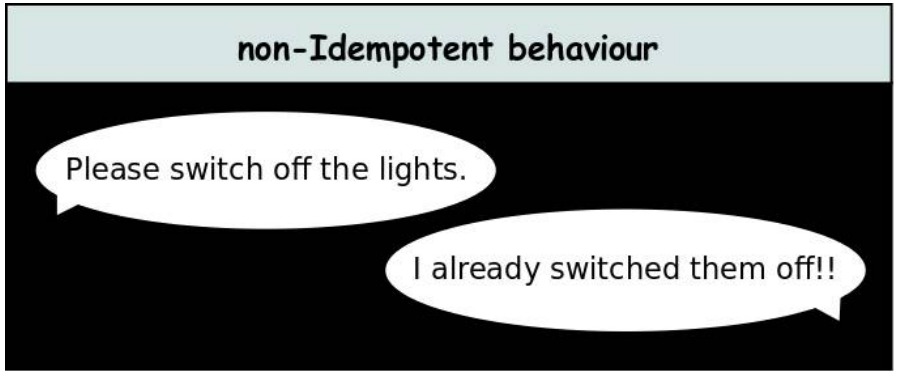
1. An arbitrary order or decree.

www.freedictionary.com

- Provide a method on resource object to release the underlying resource
  - Generally called `public void close() throws Exception;`
- Decree in the Javadoc that users must call the method to free the native resource.
- Tip: Helpful if function is “idempotent”

Sidebar: Idempotency





i•dem•po•tent adj.

(computing) Describing an action which, when performed multiple times, has no further effect on its subject after the first time it is performed.

en.wiktionary.org

- Recommendation: Make closing all resources idempotent

Active approach #1 – Management by fiat: Traps

- Relies upon end user calling `close()`.
- Tricky for user to ensure `close()` is always called at the correct moment.
  - Tip: Use try-finally blocks where possible
- Complexity explosion properly handling exceptions when closing several resources at the same time.

Active approach #1 – Management by fiat: example

```
public class MyResource3User {
    public static void main(String[] args) throws Exception {
        MyResource3 myRes = new MyResource3(/*params*/);
        try {
            /* do stuff using 'myRes' */
        } finally {
            myRes.close();
        }
    }

    public static class MyResource3 {
        private final long handle;
        private final java.util.concurrent.atomic.AtomicBoolean closed =
            new java.util.concurrent.atomic.AtomicBoolean();

        public MyResource3(/*params*/) throws Exception {
            handle = allocNativeResource(/*params*/);
        }

        /** Closes the resource. This MUST be called by the user. */
        public void close() throws Exception {
            if (closed.compareAndSet(false, true)) freeNativeResource(handle);
        }

        private static native long allocNativeResource(/*params*/) throws Exception;
        private static native void freeNativeResource(long handle) throws Exception;
    }
}
```

Active approach #2 – ARM (try-with-resources)

- try-with-resources, aka Automatic Resource Management (ARM)
- Introduced in Java SE 7 (best feature **ever** ?)
- Implement: `java.lang.AutoCloseable`
  –> `public void close() throws Exception;`
- AutoCloseable objects assigned within parentheses of try statement are automatically closed at the end of that try block
  - Automatic (implicit) generation of `finally` logic to close resources, correctly deal with exception handling
  - Support for multiple assignments in same `try` block

Active approach #2 – ARM (try-with-resources): Traps

- Needs user compliance to use try-with-resources syntax
  - Though user may call `close()` method directly instead
    - Loses exception handling “magic”
- AutoCloseable objects **assigned** within parentheses of try statement are automatically closed at the end of that try block
  –> AutoCloseable objects allocated in try statement not automatically closed (unless also assigned)
- Needs Java SE 7

Active approach #2 – ARM (try-with-resources): example

```
public class MyResource4User {
    public static void main(String[] args) throws Exception {
        try (MyResource4 myRes1 = new MyResource4(/*params*/);
                MyResource4 myRes2 = new MyResource4(/*params*/)) {
            /* do stuff using 'myRes1', 'myRes2' */
        }
    }

    public static class MyResource4 implements AutoCloseable {
        private final long handle;
        private final java.util.concurrent.atomic.AtomicBoolean closed =
            new java.util.concurrent.atomic.AtomicBoolean();

        public MyResource4(/*params*/) throws Exception {
            handle = allocNativeResource(/*params*/);
        }

        /** Closes the resource. This MUST be called by the user, optionally by using try-with-resources. */
        public void close() throws Exception {
            if (closed.compareAndSet(false, true)) freeNativeResource(handle);
        }

        private static native long allocNativeResource(/*params*/) throws Exception;
        private static native void freeNativeResource(long handle) throws Exception;
    }
}
```

Active approaches: General limitations

- Resources may be visible to “normal” code when closed
  –> Guard checking needed in each resource operation
- No guarantees user will follow instructions!!!
- **No guarantees user will follow instructions!!!**

Hybrid approach – Getting the best of both worlds

- Use active approach to allow timely freeing of resource by user
- Use passive approach to verify user is freeing resources

Combined approach: example (part 1)

```
public class MyResource5 implements AutoCloseable {
    private final long handle;
    private final MyResourceRef ref;

    private static class MyResourceRef extends java.lang.ref.PhantomReference<Object> {
        private static final java.util.Set<MyResourceRef> activeRefs = new java.util.HashSet<>();
        final Exception leakException;

        MyResourceRef(Object myRes) throws Exception {
            super(myRes, Reaper.staleRefs);
            leakException = new Exception("Resource leak detected - resource allocation call stack:");
            synchronized (activeRefs) {
                activeRefs.add(this);
            }
        }

        final boolean deactivate() { /* as before */ }

        private static final class Reaper extends Thread { /*...*/ }
    }

    public MyResource5(/*params*/) throws Exception {
        handle = allocNativeResource(/*params*/);
        ref = new MyResourceRef(this);
    }

    /** Closes the resource. This MUST be called by the user, optionally by using try-with-resources. */
    public void close() throws Exception {
        if (ref.deactivate()) freeNativeResource(handle);
    }

    private static native long allocNativeResource(/*params*/) throws Exception;
    private static native void freeNativeResource(long handle) throws Exception;
}
```

Combined approach: example (part 2)

```
public class MyResource5 implements AutoCloseable {
    private final long handle;
    private final MyResourceRef ref;

    private static class MyResourceRef extends java.lang.ref.PhantomReference<Object> {
        final Exception leakException;

        /*...*/

        private static final class Reaper extends Thread {
            private static final Reaper thread = new Reaper();
            static final java.lang.ref.ReferenceQueue<Object> staleRefs = new java.lang.ref.ReferenceQueue<>();

            static {
                thread.start();
            }

            private Reaper() {
                setDaemon(true);
            }

            public void run() {
                do {
                    try {
                        MyResourceRef staleRef = (MyResourceRef)staleRefs.remove();
                        if (staleRef.deactivate()) {
                            staleRef.leakException.printStackTrace();
                        }
                    } catch (InterruptedException ie) {}
                } while (true);
            }
        }
    }
}
```

Recommendations

- Use “Active” approach
  - Use to allow user to swiftly free resources
  - Implement AutoCloseable
    - Allows user to safely free resources using ARM
- Use “Passive” approach
  - Use to monitor resources are correctly freed by user
    - Even for key “standard” Java resources ?
  - Use PhantomReference over Finalization
    - Ease of correct implementation
    - Flexibility
  - Fail-fast if resource leakage is detected
- Use both approaches in combination
  - Allow user to free resources in a timely manner
  - Monitor for & Report resource leakage

Copyright and Trademarks