# Finalizers, Threads, and the Java Memory Model

**Hans-J. Boehm**
**HP Labs**

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Basics (`finalize`)

- **`Object.finalize()`**
  - "called … when there are no more references to the object."
  - Intended for object clean-up.
  - No promptness guarantee.
    - May never be called.
  - Used to reclaim resources (other than heap memory).

# Basics (`java.lang.ref`)

- **`java.lang.ref`**.{**`Phantom`**,**`Soft`**,**`Weak`**}**`Reference`**
  - Enqueues “unreachable” objects.
  - Can be used for cleanup.
    - Or to introduce reference that doesn’t prevent GC.
  - May provide better performance.
  - Interface more flexible in some ways.
  - Requires explicit queue, possibly separate thread.
  - Threading more explicit.
  - Issues similar to **`Object.finalize()`**.
    - At least for our purposes.
    - We concentrate on **`finalize()`**.

# Bad finalization advice 1

- Don't try this at home:
- Add finalizers to help the garbage collector.
- The facts:
  - If the finalizer is run, the GC already knows it's unreachable. It's done the work.
  - JVMs treat objects with default **`finalize()`** specially.
  - Non-default **`finalize`** methods:
    - add GC time and space overhead.
    - may interact badly with generational GC.

# Finalization performance impact

- A quick experiment:
  - GCBench: Large binary trees, small nodes:
    - Add `finalize()` to all tree objects (clears fields).
      - This is a ridiculous stress test. Don't do this.
    - gcj compiled (non-generational GC), old X86 machine:
      - Factor of about 7.3 slowdown.
    - BEA JRockit1.5, Itanium 2 machine:
      - Factor of about 11 slowdown.
    - Sun J2SE 1.4.2 client/server, old X86, with increased memory:
      - Operator error: insufficient patience.
      - (Fast without finalization.)
    - Pervasive finalization ➔ substantial slowdown

# Bad finalization advice 2

- It's a replacement for C++ destructors.
- It's completely useless.
- Avoid locking in finalizers.

- Let's start with a clean slate.

# What finalization is good for:

- Cleanup of non-memory resources for objects
  - With hard to predict lifetimes.
  - For which cleanup is not time-critical.
- If lifetimes are hard to predict:
  - Cleanup usually isn't time-critical.
- If clean-up is time critical:
  - Use explicit dispose/close calls.

# Finalization guidelines

- If you can easily avoid finalization, do so.
- One finalize() method per 10K to 100K lines of code seems common in well-written code.
- Finalization is the only way you can use the collector's knowledge about lifetime.
  - If you need that, use finalization.
  - If you don't, don't.
- Don't rewrite the garbage collector to avoid finalization.
- If you do need it, avoid the pitfalls …

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Possible uses of finalizers

- Explicit deallocation of native objects.
- Deallocation of non-memory resources embedded deeply inside linked structures (e.g. file references.)
- Last ditch reclamation of dropped resources.
  - Bug reporting.
  - Esoteric error recovery?
- Removing external data associated with an object (e.g. in separate table).
- Guaranteed resource cleanup.
  - At process termination, opportunistically earlier.
  - E.g. temporary files.

# Problem with finalizer examples

- Small programs don’t need finalizers.
  - For the same reason they don’t need garbage collectors.
- … and we’re only entitled to one after 10K lines of code ☺
- Thus this will have to be somewhat abstract.

# Example problem

- We have **`Resource`**s:

```
class Resource
{
  public Resource( … ) { … }
  public void recycle () { … }
  public … doSomething() { … }
}
```

- We want **`CleanResource`**s that
  - Recycle themselves when no longer used (become unreachable).
  - At process exit if all else fails.

# False Start 1

- Derive **`CleanResource`** from **`Resource`**.
- Add

```
protected void finalize() { recycle(); }
```

- Call

```
System.runFinalizersonExit();
```

# False Start 1: Problem

- `runFinalizersOnExit()` is deprecated.
- … for good reasons.
  - Resource.recycle() may refer to “permanent” data, directly or indirectly, e.g. to print to log file.
  - It probably does, since it’s useless to update itself.
    - It’s about to be garbage collected.
  - Those static class members may have been finalized first. The log file may already be history.
  - And daemon threads etc. may still be accessing objects that we just asked to be finalized.
- Finalizing reachable objects is bad.

# (Somewhat) False Start 2

- Observation: We need to explicitly control recycling at process termination.
  - The only chance to get the ordering right.
- Add **`finalize()`** method as before.
- Add all live Resources to a static container **`all`**.
- Add **`recycleAll()`** method.
  - Called explicitly during shutdown.
    - Before cleanup of other resources needed by **`recycle`**.
  - Recycles objects in **`all`**.

# A New Problem

- Better approach, but …
  - Container `all` refers to all `Resource`s.
  - No `Resource` ever becomes unreachable.
  - Nothing is ever finalized.

- We could attack this with `WeakReference,` but
  - Not completely trivial. (Referenced object unavailable.)
  - Instructive to handle purely with `finalize()`.

# A better approach

- Keep actual Resources in container **`all`**.
- A CleanResource is just an index into **`all`**.

```
public class CleanResource  {
  static ArrayList<Resource> all =
                 new ArrayList<Resource>();
  static BitSet taken = new BitSet();
  final int myIndex;
  …
}
```

# The Constructor: (incorrect, closer)

```
public CleanResource( … x) {
  Resource myResource = new Resource(x);
  myIndex = taken.nextClearBit(0);

  if (all.size() == myIndex)
    all.add(myResource);
  else {
    assert all.get(myIndex) == null;
    all.set(myIndex, myResource);
  }
  taken.set(myIndex);
}
```

# doSomething(): (incorrect, closer)

```
public long doSomething() {
  Resource myImpl;

  assert taken.get(myIndex);
  myImpl = all.get(myIndex);
  return myImpl.doSomething();
}
```

# Finalizer: (incorrect, closer)

```
protected void finalize() {
  if (taken.get(myIndex)) {
    all.get(myIndex).recycle();
    all.set(myIndex, null);
    taken.clear(myIndex);
  }
}
```

# recycleAll(): (incorrect, closer)

```
public static void recycleAll() {
  for (int i = 0; i < taken.length(); ++i)
    if (taken.get(i)) {
      all.get(i).recycle();
      taken.clear(i);
    }
}
```

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Problem: Concurrency

- If the client of **`CleanResource`** is multithreaded, we may get concurrent access to **`all`** data structure.
- This is currently unsafe.
  - **`ArrayList`**s, **`Bitset`**s are not synchronized.
  - It wouldn't help if they were.
- Even if the client is single-threaded, finalizers run in their own thread.
- **`finalize()`** call may introduce concurrent access to **`all`** data structure.
- Finalizers introduce concurrency.
- Synchronization required for "single-threaded" client!

# Solution: Synchronize access to `all`

- Use lock associated with **`all`** ArrayList to protect both **`all`** and **`taken`** vector.
- Wrap all accesses to combined data structure in

```
synchronized(all) { … }
```

# The Constructor: (final version)

```
public CleanResource( … x) {
  Resource myResource = new Resource(x);
  synchronized(all) {
    myIndex = taken.nextClearBit(0);
    if (all.size() == myIndex)
      all.add(myResource);
    else {
      assert all.get(myIndex) == null;
      all.set(myIndex, myResource);
    }
    taken.set(myIndex);
  }
}
```

# `doSomething()`: (near final version)

```
public long doSomething() {
  Resource myImpl;

  synchronized(all) {
    assert taken.get(myIndex);
    myImpl = all.get(myIndex);
  }
  return myImpl.doSomething();
}
```

# Finalizer: (near final version)

```
protected void finalize() {
  Resource myResource;
  synchronized(all) {
    if (!taken.get(myIndex)) return;
    myResource = all.get(myIndex);
    all.set(myIndex, null);
    taken.clear(myIndex);
  }
  myResource.recycle();
}
```

# `recycleAll()`: (final version)

```
public static void recycleAll() {
   for (int i = 0; i < taken.length(); ++i)
     {
      Resource myResource;
      synchronized(all) {
         if (!taken.get(i)) continue;
         myResource = all.get(i);
         taken.clear(i);
      }
      myResource.recycle();
   }
}
```

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Subtle Problem: Reachable?

- Finalizer may begin executing as soon as object is no longer reachable, i.e. GC could otherwise collect it.
- This may happen earlier than you think.
  - In last call to **`CleanResource.doSomething()`**, **`this`** pointer is last accessed to retrieve **`myIndex`**.
  - **`myIndex`** is final, can be read early.
  - Register containing **`this`** may be reused at that point, making **`CleanResource`** instance no longer reachable.
  - **`Resource.recycle()`** may run while **`Resource.doSomething()`** is still running. Oops.

# Related Issue: Memory visibility

- `Finalize()` runs in different thread.
- Any updates to `Resource` being recycled should be visible to finalizer.
- Finalizer rules ensure only visibility of writes
  - That happen-before the constructor finishes, or
  - To the `CleanResource` itself.
- Need synchronization
  - At end of ordinary methods.
  - At beginning of finalizer.
- Synchronizing on `all` is insufficient, as it stands.

# Consequences of reachability & visibility issues

- Probably most existing code has problems in this area.
- Failure is unlikely. Requires:
  - Compiler elimination of dead variable.
  - Register-based calling convention, no spills.
  - GC at just the wrong point.
- But
  - Really hard to debug.
  - More likely on some platforms, e.g. X86-64 vs. X86.
- JSR133 provides mechanisms to avoid it.
  - Not yet clear whether it went far enough.

# Reachability, visibility solutions

- We want something that
  - Ensures reachability
  - Synchronizes to create a happens-before relationship between ordinary methods and finalizer.
- Options provided by JSR133:
  - Store into volatile field in ordinary method.
    - Read field in finalizer.
  - Store reference to object into volatile static, then immediately clear it.
    - Read volatile static in finalizer.
  - Release lock on object at end of ordinary method.
    - Acquire lock at beginning of finalizer.

# Our reachability solution

- Define additional method:

```
synchronized void keepAlive() {}
```

- Call it at the end of any regular function that might be the last call on the object.

- Add to beginning of finalizer:

```
synchronized(this){}
```

# doSomething(): (final version)

```
public long doSomething() {
  Resource myImpl;
  long result;

  synchronized(all) {
    assert taken.get(myIndex);
    myImpl = all.get(myIndex);
  }
  result = myImpl.doSomething();
  keepAlive();
  return result;
}
```

# Finalizer: (final version)

```
protected void finalize() {
   synchronized(this){}
   Resource myResource;
   synchronized(all) {
      if (!taken.get(myIndex)) return;
      myResource = all.get(myIndex);
      all.set(myIndex, null);
      taken.clear(myIndex);
   }
   myResource.recycle();
}
```

# Finalization is Unordered

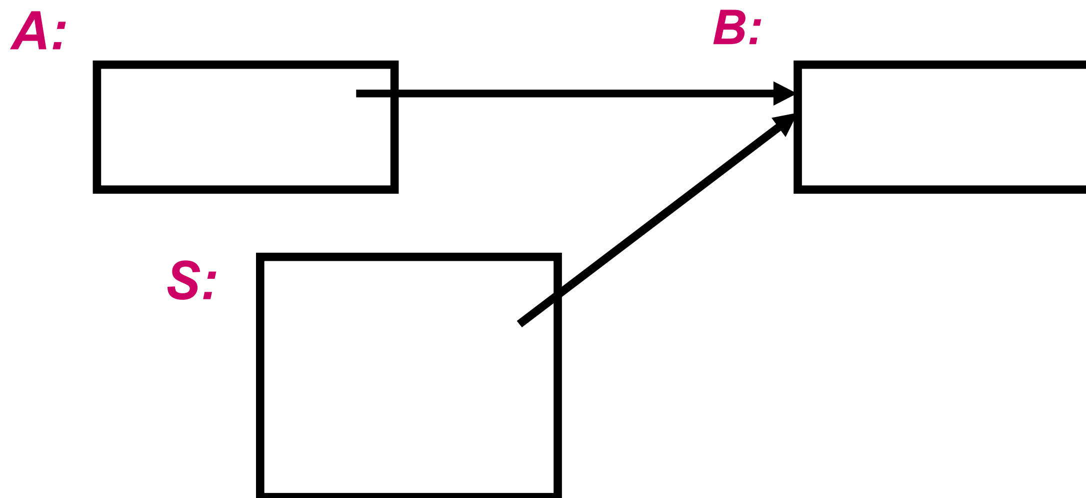
- Another potential finalization pitfall:
- If *A* refers to *B*, and both have finalizers:
  - *B* may be finalized first.
  - *A*'s finalizer should not use *B* without precautions.
    - Otherwise it may see a finalized object.
    - If *B*'s finalizer cleaned up native objects, *A*'s may dereference dangling *native* pointers.
- This applies if *A*'s finalizer needs *C*,
  - Which needs *D*
    - Which needs *E*
      - Which needs *B*, which is finalizable

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Enforcing ordering among finalizers

- If *A*'s finalizer needs *B*:
  - *A*'s constructor should ensure that *B* is added to static data structure *S* and hence not finalizable.
  - *A*'s finalizer should remove *B*.



# Ordering alternative

- Use **`java.lang.ref`**.
- Put information needed for cleanup in reference, not in the object.
- Probably an easier discipline to follow.
  - Fundamentally, it has the same effect.

# Agenda

- Review of finalizers & java.lang.ref
- Naïve Example
- Finalizers introduce concurrency
  - Synchronize!
- Finalizers can run earlier than you think!
  - Synchronize!
- Java finalization is unordered
- Summary

# Summary

- Finalizers are:
  - Often misunderstood, misused.
  - Rarely needed.
  - Occasionally extremely useful.
- Finalizers introduce concurrency.
  - Synchronization is normally required.
- Finalizers risk deallocating resources still in use by executing methods of unreachable object.
  - Can be addressed with synchronization.
- **`Finalize()`** methods that access other objects need to ensure finalization ordering.

# Acknowledgements, further info


- This is based on long discussions during the development of the JSR 133 spec. Active participants in relevant discussions included Jeremy Manson and Bill Pugh, and many others.
- Some of the general observations about finalizers were made by Barry Hayes, more than 10 years ago.


- More details on finalization issues can be found in Boehm, "Destructors, Finalizers, and Synchronization", POPL 2003.
- For some mostly orthogonal advice about finalizers and inheritance (correct since JSR 133), see Joshua Bloch, "Effective Java", chapter 2.