Destructors, Finalizers, and Synchronization

Hans-J. Boehm
HP Laboratories

Hans.Boehm@hp.com

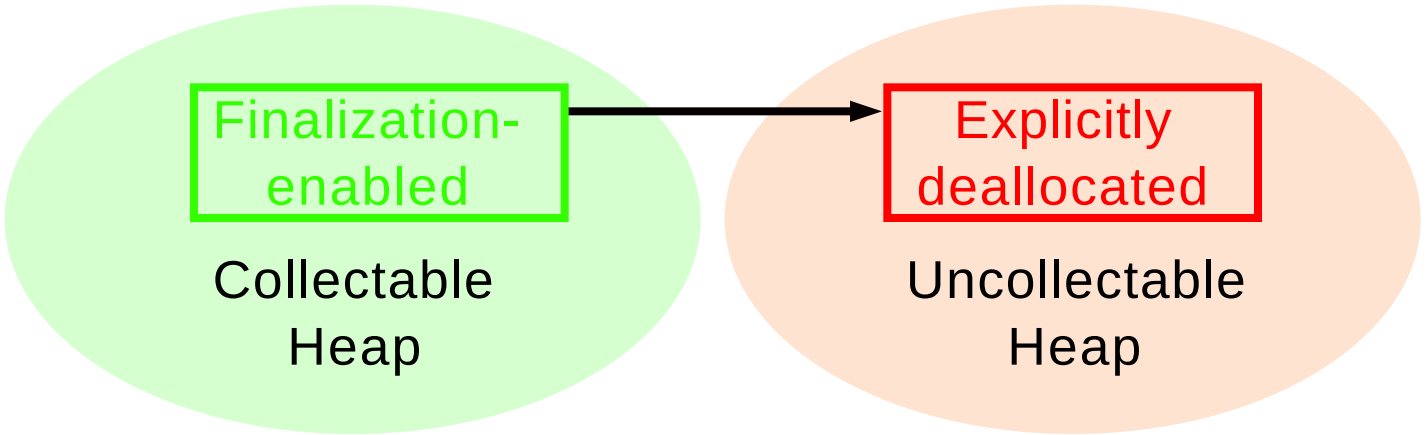
Object cleanup

- C++ destructors
  - Executed synchronously at specific program point.
  - Convenient notation.
  - Used to manage cleanup after exceptions.
  - Often used pervasively in C++.
  - Canonical example:

```
{
    scoped_lock sl(L);
    do_something();
}
```

Object Cleanup (2)

- Java Finalization (a.k.a. C# destructors)
  - Leverages garbage collector for non-memory resources.
  - Cleanup code is executed for otherwise unreachable objects.
  - Rarely used, but very hard to avoid.
  - Canonical use:



Implementing finalization

- (Small) subset of objects *F* is finalization-enabled.
- Runtime keeps a data structure representing *F*.
- After GC, untraced objects in *F* are finalizable.
  - These objects are enqueued for finalization.
- Details depend on finalizer ordering:
  - May not want to finalize objects reachable from finalization-enabled objects (Modula-3).
  - May need to prevent collection of objects accessed during finalization (Java, C#).
  - No significant impact on performance.

Overview (rest of talk):

- Paper discusses
  - Example uses of finalization.
  - Observations about programming with finalizers.
    - Concurrency issues.
  - Language design issues.
  - Why finalizer ordering does and doesn't matter.

- Talk instead looks at specific "myths".
  - Many misunderstandings.
  - Complexity is largely self-inflicted, not inherent.

- Assume Java unless otherwise stated.

Myth #1:

Java 2 Black Book (introductory Java book):

[Dubious discussion of circular references.]
When an object is being "garbage collected" ... , the garbage collector will call a method named finalize in the object, if it exists. In this method, you can execute cleanup code, [good so far]

and it's often a good idea to get rid of any references to other objects that the current object has in order to eliminate the possibility of circular references ...

Really 3 myths?

- Cyclic garbage is hard to collect.
  - Applies at most to reference counting.
  - Almost all JVMs use tracing GC.
- Finalizers can help the collector.
  - The collector needs to determine that the object is unreachable to run the finalizers.
  - Cycles may affect finalizability, but not in Java.
- Finalization is cheap
  - Finalization-enabling an object usually increases allocation and collection cost, perhaps by 3x.

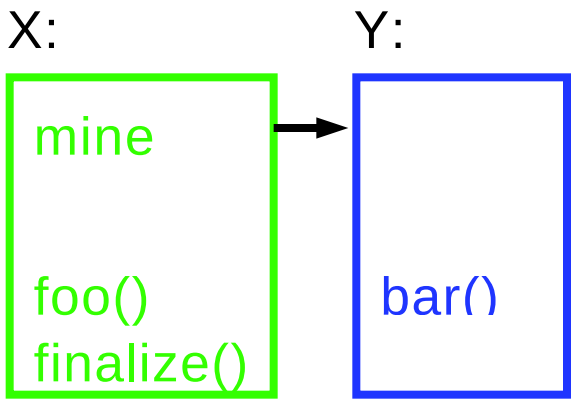
Myth #2: (usually implicit)

Finalizers run only after all other method calls on the object have completed.

- Java finalizers may run run when the object can no longer "be accessed in any potential continuing computation ..." This may occur with a running method, e.g.:

```
class X {
     Y mine;
        // mine is not shared.

     public foo() {
        ...
        mine.bar();
     }};
```



Myth #3:

Finalizers should avoid synchronization.

- Useful finalizers update external state.
- External state is typically shared.
- Needs to synchronize (perhaps implicitly).
- Finalizers *introduce* concurrency (stay tuned ...).
  - Finalizers in *single-threaded* Java/C# code may need to lock

Myth #4:

Finalizers are crippled because they may be run too late, instead of immediately when an object becomes unreachable.

- Running finalizers "immediately" is not meaningful unless they are run from the thread overwriting the last pointer.
- Unlike the destructor case, it is not practically predictable when finalizers will be run. (If it were, we wouldn't need a garbage collector.)
- The thread overwriting the last pointer may already hold lock needed by finalizer.

  ==> deadlock (or worse)

- Garbage collectors should run finalizers from a separate thread.
  - Tracing collectors should never run finalizers from allocator.
    - Unfortunately version 1 usually does.
    - What about System.runFinalization()?
  - A reference count decrement should not trigger finalization calls.
    - Unfortunately, standard reference count libraries usually do.
  - Workarounds (explicit queueing) may be possible.

Late finalization is *necessary*, but *early* finalization may be a problem:

- Object is finalized when the collector discovers it to be unreachable.
- One of its fields may still be in a register.
- If that field is a handle / file descriptor:
  - finalizer may close it while being accessed.
  - or not?

In my view:

- Java / C# "reachability" are underspecified.
  - Not just in this respect (see also myth 8).
- Java objects appear to be reachable while locked.
  ==> synchronize accesses to finalizable objects.

Myth #5

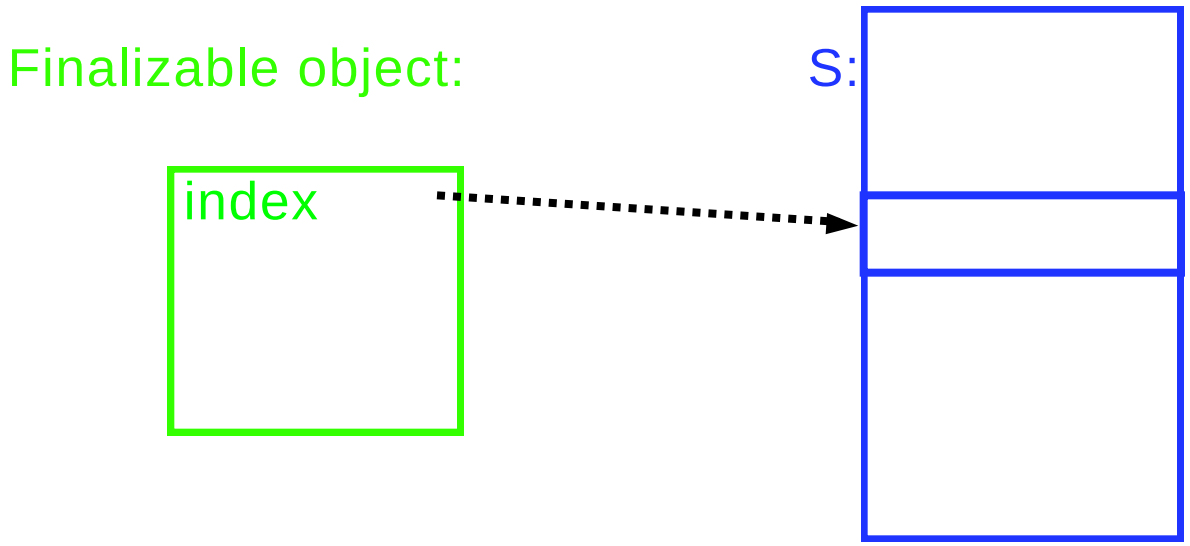
All finalizers should be run before process exit to ensure proper cleanup.

- Can't be done safely.
  - *Must run finalizers on reachable objects.*
  - Any finalizer may be the last one to be run.
  - All other objects in the system have been finalized at that point.
  - Cannot safely stop all threads beforehand.
  - Will finalize objects being accessed by daemon threads.

Myth #6:

Finalizers cannot ensure reliable cleanup of *e.g.* temporary files.

- Keep state needing cleanup in a separate array *S*.
- Run explicit cleanup routine over *S* at exit.



Myth #7:

Finalizers cannot manage scarce resources, because the collector may run too infrequently.

- Resource allocator can run GC and finalization, but:
- This requires careful attention to deadlocks.
  - Thread calling allocator may hold lock.
  - Remember finalizer dependencies!
    - Other finalizers need to run, too.
  - Allocators of scarce resources should not be called with locks held?

Myth #8:

If A is reachable and points to B, then B is reachable.

- Usually true for standard implementations.
- *Not* guaranteed by Java spec.

Conclusions

Finalizers:

- are rarely needed.
- may need thousands of lines of code to avoid.
- are inherently asynchronous.
- clean up objects of unpredictable lifetime.
- are usually misunderstood.

Destructors:

- are used pervasively.
- can be easily (but inconveniently) avoided.
- are synchronous.
- clean up objects of predictable lifetime.
- are reasonably well understood.