



# Performance Tuning: GC Friendly Java Programming

**Sang Shin**

Technology Evangelist  
Sun Microsystems, Inc.



# Agenda

- Garbage Collection (GC) Concepts
- Programming Tips
- Problems With Finalization
- Using Reference Objects
- Conclusions

# Object Allocation

# Creating Work for the GC

- Allocation
  - > But, typically, **super** fast
    - > Maybe more expensive for non-compacting GCs
  - > Higher allocation rate implies more frequent GCs
- Live data size
  - > More work for the GC to find what is live
- Reference field updates
  - > More overhead on the application, ...
  - > And it also creates more work for the GC
    - > Especially on generational/incremental GCs

# Object Allocation

- Typically, object allocation is very cheap!
  - > 10 native instructions in the fast common case
  - > No remembered set overhead on new objects
  - > C/C++ has faster allocation? Not!
- Reclamation of new objects is very cheap too!
  - > Young GCs in generational systems
- So
  - > Do not be afraid to allocate small objects for intermediate results
  - > GCs **love** small, immutable objects
  - > Generational GCs **love** small, short-lived objects

# Object Allocation (2/2)

- This is NOT advising needless allocation
  - > More allocation means more GC
- What this does mean is
  - > Try to use short-lived immutable objects instead of long-lived mutable objects
  - > Use clearer, simpler code with more allocations instead of more obscure code with fewer allocations

# Large Objects

# Large Objects

- Very large objects are:
  - > Expensive to allocate (maybe not through the fast path)
  - > Expensive to initialize (zeroing)
  - > Can cause performance issues
- Large objects of different sizes can cause fragmentation
  - > For non-compacting or partially-compacting GCs
- Avoid if you can
  - > And, yes, this is not always possible or desirable

# Explicit GCs

# Explicit GCs

- DO NOT DO IT!
  - > Applications do not have enough information
  - > GC does (knows allocation/promotion rate, etc.)
  - > System.gc() at the wrong time
    - > Hurts performance with no benefit
- Exceptions
  - > Between well-defined application phases (maybe)
  - > When performance does not matter (e.g., late at night)
- Java HotSpot™ virtual machine
  - > System.gc() does a stop-the-world full GC
  - > Use -XX:+DisableExplicitGC to ignore System.gc()

# Explicit GCs

- Incremental GCs
  - > Designed to avoid full GCs...
  - > But `System.gc()` does exactly that!
- In the Java HotSpot virtual machine (CMS)
  - > `-XX:+ExplicitGCInvokesConcurrent`
- Beware
  - > Libraries that call `System.gc()`
    - > Run FindBugs over your libraries to check for that
  - > Java™ RMI calls `System.gc()` for its distributed GC algorithm
    - > Decrease its frequency, or invoke concurrent, or both!

# Data Structure Sizing

# Data Structure Sizing

- Array-based data structures
  - > Avoid frequent re-sizing
- e.g., this will allocate the associated array twice

```
ArrayList<String> list = new ArrayList<String>();  
list.ensureCapacity(1024);
```

- The preferred version
  - > (Part of periodic audits of the Java Platform, Standard Edition (Java SE) libraries)

```
ArrayList<String> list = new ArrayList<String>(1024);
```

# Data Structure Sizing

- Additionally, try to size data structures as realistically as possible

```
ArrayList<String> list = new ArrayList<String>(1024);
```

- If 1M strings are added to it:
  - > Several array-resizing operations will take place
  - > They will allocate several large-ish arrays
  - > They will cause a lot of array copying
  - > They might cause fragmentation issues on non-compacting GCs

# Object Pooling

# Object Pooling

- Legacy of older VMs with terrible allocation performance
- Remember
  - > Generational GCs **love** short-lived, immutable objects...
  - > Not long-lived, highly mutable objects
- Unused objects in pools
  - > Are like a bad tax
  - > Are live; the GC must process them
  - > Provide no benefit; the application does not use them

# Object Pooling

- Exceptions
  - > Objects that are expensive to allocate and/or initialize
  - > Objects that represent scarce resources
  - > Examples
    - > Threads pools
    - > Database connection pools
- Caveats to the exceptions
  - > Use existing libraries wherever possible
  - > Can you write a better thread pool than Doug Lea?

# Problems with Finalization

# Finalization Description

- Finalization
  - > Essentially, a postmortem hook
  - > Allows cleanup when GC finds an object unreachable
  - > Typically used to reclaim native resources
- Finalizable objects
  - > Have a non-trivial finalize() method

# Allocation/Reclamation

- Finalizable object allocation
  - > Much slower
  - > The VM must track finalizable objects
- Finalizable object reclamation
  - > It takes at least two GC cycles
    - > The GC cycles are slower too
  - > First cycle identifies object as garbage
    - > Enqueues object on finalization queue
  - > Second cycle reclaims space after finalize() completes
    - > Unless finalize() resurrects the object!

# Finalizers vs. Destructors

- Beware
  - > Finalizers are not like C++ destructors!
- No guarantees
  - > When they will be called
  - > Whether they will be called
  - > The order in which they will be called
- The closest concept to a destructor
  - > Finally clause

# Finalizers and Memory Retention

- Finalizable objects
  - > Are retained longer
  - > Along with everything reachable from them
  - > finalize() is an application-defined method
    - > It may access any field
- More pressure on the GC

# “Sneaky” Memory Retention

- You do not have to explicitly use finalizers
  - > To be affected by finalization-induced heap pressure
  - > Library classes you extend might define finalizers
- Below, buffer will survive at least two GC cycles
  - > In Java Development Kit (JDK™) 1.5 and earlier

```
class MyFrame extends JFrame {  
    private byte[] buffer = new byte[16 * 1024 * 1024];  
    ...  
}
```

# Memory Leak Types

- “Traditional” memory leaks
  - > Heap keeps growing, and growing, and growing...
  - > OutOfMemoryError
- “Temporary” memory leaks
  - > Heap usage is temporarily very high, then it decreases
  - > Bursts of frequent GCs
- Both finalizers and reference objects
  - > Can delay the reclamation of objects...
  - > As well as everything reachable from them
  - > Temporary heap usage spikes

# Memory Leak Detection Tools

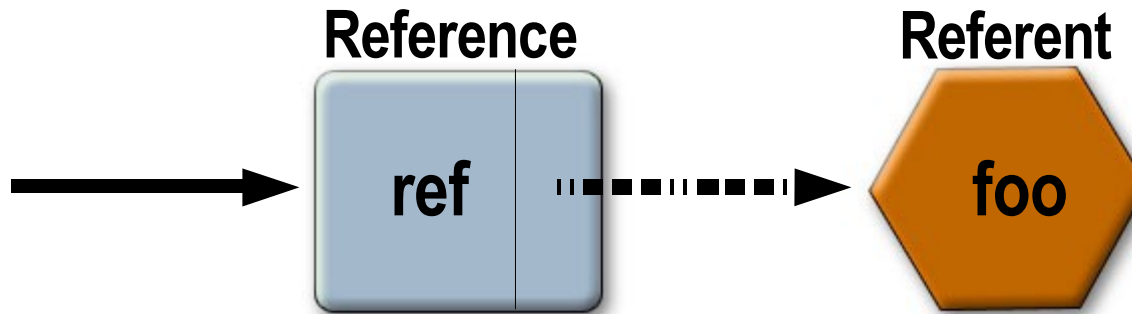
- Many tools to choose from
- *“Is there a memory leak?”*
  - > Monitor VM’s heap usage with jconsole and jstat
- *“Which objects are filling up the heap?”*
  - > Get a class histogram with jmap or
  - > -XX:+PrintClassHistogram and Ctrl-Break
- *“Why are these objects still reachable?”*
  - > Get reachability analysis with jhat

# Using Reference Objects

# Reference Objects

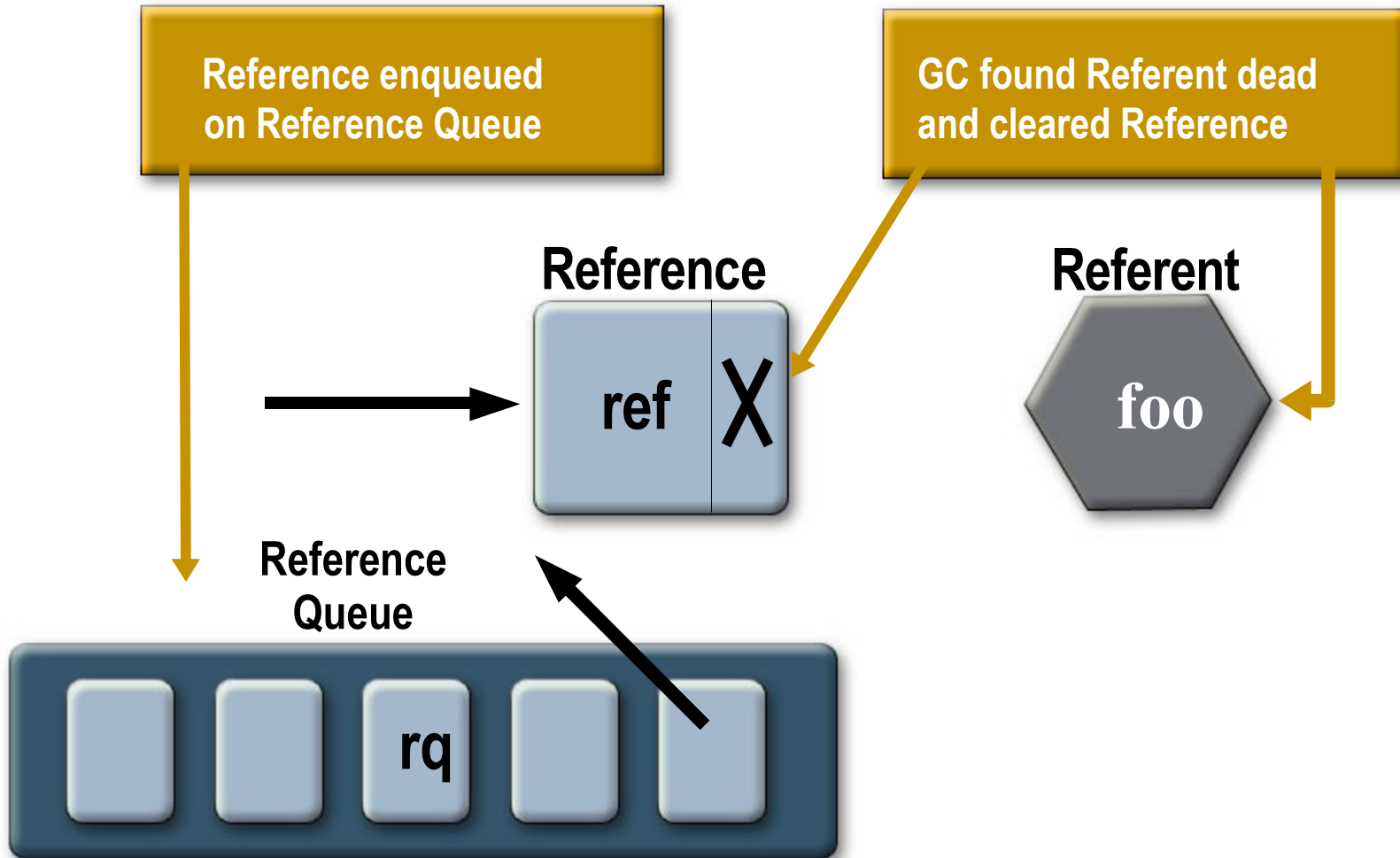
- Purpose
  - > Postmortem hooks, more flexible than finalization
- Three types of reference objects
  - > Weak references
  - > Soft references
  - > Phantom references
- All three
  - > Can enqueue the reference object...
  - > On a designated reference Queue...
  - > When the GC finds its referent to be unreachable

# Reference Objects: Illustration (1/2)



```
ref = new WeakReference(foo, rq);
```

# Reference Objects: Illustration



# Weak References

# Weak References

- Uses
  - > *Tell me if the object has been reclaimed by the GC*
  - > *Do not retain this object because of this reference*
- `get()` returns
  - > The referent, if not reclaimed
  - > null, otherwise
- Referent is cleared by the GC

# Weak References

- Using weak references you can implement a flexible version of finalization that allows you to...
  - > Prioritize object “finalization,”
  - > Decide when to run object “finalization,”
  - > Stop objects from being considered for “finalization,”
  - > Be unaffected by the VM’s finalization queue,
  - > Etc.
- See link below for a code sketch
  - > <http://www.devx.com/Java/Article/30192>

# Soft References

- Uses
  - > *Only reclaim this object if there is memory pressure*
- `get()` returns
  - > The referent, if not reclaimed
  - > null, otherwise
- Referent is cleared by the GC

# Soft References

- Implementing soft reference policy is tricky
  - > Hard to make informed decisions
    - > How much data reachable from each reference?
      - Prohibitively expensive to calculate
    - > How expensive to recreate?
- OK for quick and simple caches
  - > Remember: create strong references to data you want to keep

# Phantom References

- Uses
  - > *Keep some data around after the object becomes unreachable so that I can use that data to clean up after the object”*
- `get()` returns
  - > null, always
- Referent is **not** cleared by the GC automatically
  - > It is cleared when the reference object becomes unreachable

# Conclusions

- We covered a series of tips on how to write
  - > Simpler
  - > More readable
  - > More GC-friendly code
- GC Tuning
  - > You **will** improve your application performance if you tune GC for your application

# More Information

- Memory management white paper
  - > <http://java.sun.com/j2se/reference/whitepapers/>
- Destructors, Finalizers, and Synchronization
  - > <http://portal.acm.org/citation.cfm?id=604153>
- Finalization, Threads, and the Java Technology Memory Model
  - > <http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3281.html>
- Memory-retention due to finalization article
  - > <http://www.devx.com/Java/Article/30192>

# More Information

- Heap analysis tools
  - > Monitoring and Management in 6.0
    - > <http://java.sun.com/developer/technicalArticles/J2SE/monitoring/>
  - > Troubleshooting guide
    - > <http://java.sun.com/javase/6/webnotes/trouble/>
  - > JConsole
    - > <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>



# Performance Tuning: GC Friendly Java Programming

**Sang Shin**

Technology Evangelist  
Sun Microsystems, Inc.

