Sun
JavaOne

Garbage Collection-Friendly Programming

John Coomes, Peter Kessler, Tony Printezis

Java SE Garbage Collection Group
Sun Microsystems, Inc.
http://java.sun.com/

TS-2906

Our Goal

To give you tips on how to write **readable** and **clean** code that makes the most out of the garbage collector (in terms of throughput, responsiveness, etc.).

The One Thing You Should Remember

"Everything should be made as simple as possible, but not simpler."

—Albert Einstein

Agenda

Garbage Collection Concepts
Programming Tips
Problems With Finalization
Using Reference Objects
Memory Leak Avoidance
Conclusions

Agenda

Garbage Collection Concepts

Programming Tips

Problems With Finalization

Using Reference Objects

Memory Leak Avoidance

Conclusions

Java

JavaOne

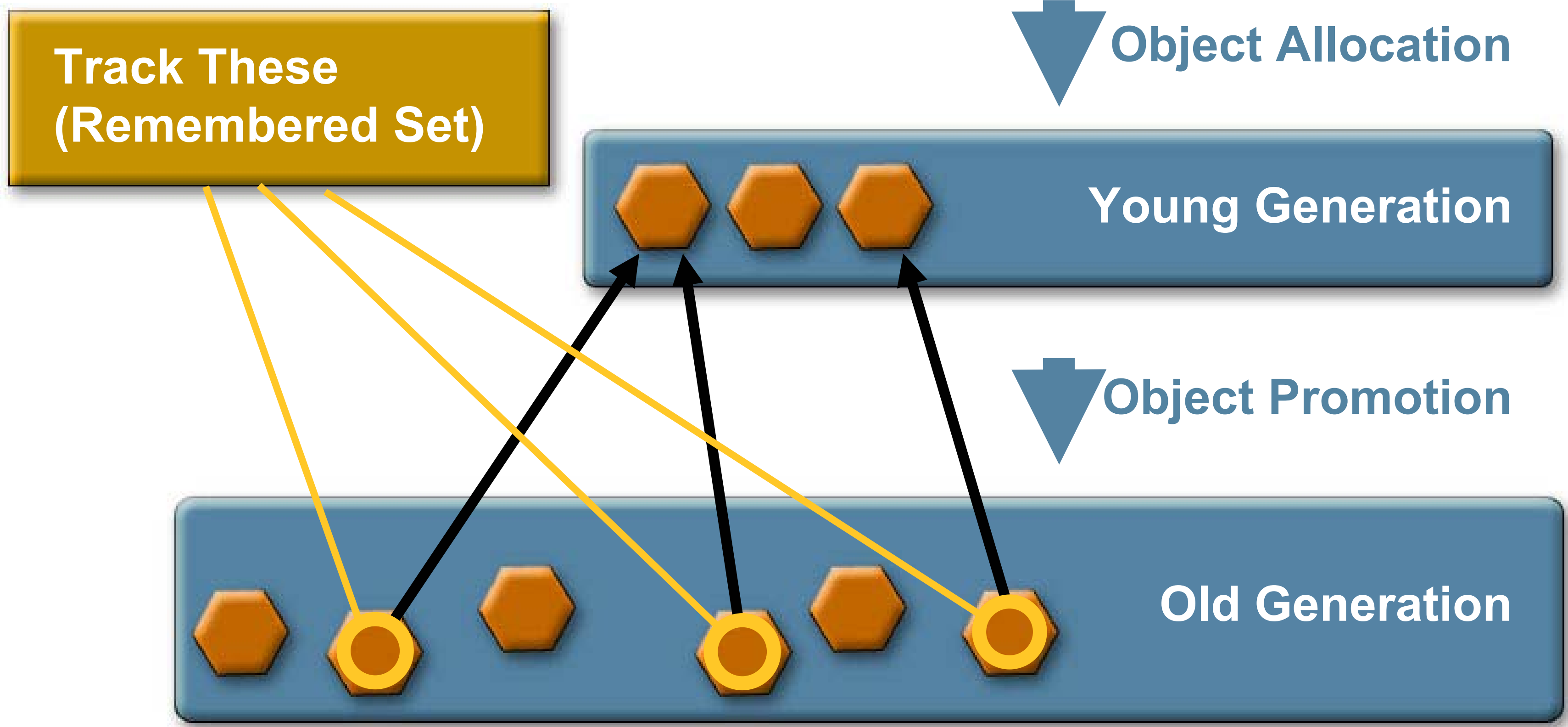
Garbage Collection

- Find and reclaim **unreachable** objects
  - Anything not transitively reachable from the application **roots** (thread stacks, static fields, etc.)
- Automatic and safe
- Easiest if the object graph is “frozen”
  - Stop-the-world pauses
- Variety of approaches
  - Compacting/non-compacting
  - Algorithms: copying, mark-sweep, mark-compact, etc.
  - Allocation: linear, free lists, etc.

Generational Garbage Collection (1/2)

- Keeps young and old objects separately
  - In spaces called **generations**
- The weak generational hypothesis
  - Most new objects will die young
  - Concentrate effort on young generation
    - Need to keep track of old-to-young pointers
    - Reference update tracking on old objects (write barrier)
  - Eventually, have to also collect the old generation
- Different GC algorithms for each generation
  - ***"Use the right tool for the job"***

Generational Garbage Collection (2/2)



Incremental Garbage Collection

- Tries to decrease/minimize GC disruption
- GC works at the same time as the application
  - The object graph is being mutated while the GC works
  - GC needs to be notified about object graph mutations
    - Reference update tracking (write barrier)
- If only old generation is incremental
  - No need to track updates on young objects

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

Java

JavaOne

Agenda

Garbage Collection Concepts

Programming Tips

Problems With Finalization

Using Reference Objects

Memory Leak Avoidance

Conclusions

Programming Tips

- Object allocation
- Large objects
- Pointer nulling
- Explicit GCs
- Data structure sizing
- NUMA
- Object pooling

Object Allocation (1/2)

- 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)

- We **do not** advise
  - Needless allocation
    - More frequent allocations will cause more frequent GCs
- We **do** advise
  - Using short-lived immutable objects instead of long-lived mutable objects
  - Using clearer, simpler code with more allocations instead of more obscure code with fewer allocations

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

Reference Field Nulling

- Nulling references rarely helps the GC
  - The GC does fine by itself!
  - **Best Case:** mostly worthless clutter in your code
  - **Worst Case:** introduces a bug
    (it may reveal itself later)
- Exceptions
  - Array-based data structures
    - e.g., the implementation of the ArrayList class
    - In this case, you’re managing your own memory…
    - So please, let the standard libraries do that!
  - Avoiding finalizer-induced memory retention
    - Avoid finalizers as much as possible (more on this later)

Local Variable Nulling

- Local variable nulling is not necessary
  - The JIT can do liveness analysis

```
void foo() {
    int[] array = new int[1024];
    populate(array);
    print(array); // last use of array in method foo()
    array = null; // unnecessary!
    // array is no longer considered live by the GC
    ...
}
```

Explicit GCs (1/2)

- Avoid them!
  - 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 (2/2)

- 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 (1/2)

- 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 (2/2)

- 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

NUMA

- Asymmetric memory access
  - Each CPU accesses its local memory faster
  - e.g., large SPARC® computers, Opteron™
- What we try to do
  - Allocate objects to memory of allocating CPU/thread
- Something to consider
  - Allocating thread also manipulates the objects too
  - You might see a performance benefit
- Avoiding thread ‘hops’ is a good idea anyway

Object Pooling (1/3)

- 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 (2/3)

- List of issues
  - Sizing
    - Too small: allocate anyway
    - Too large: too much footprint overhead + pressure on GC
  - Safety
    - Reintroduce malloc/free mistakes
  - Scalability
    - Must allocate/de-allocate efficiently
    - **synchronized** defeats the VM's fast allocation mechanism
  - Compatibility
    - Incompatible with most standard libraries

Java

JavaOne

Object Pooling (3/3)

- 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?

Agenda

Garbage Collection Concepts
Programming Tips
Problems With Finalization
Using Reference Objects
Memory Leak Avoidance
Conclusions

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!

Finalizers vs. Destructors

- Beware
  - Finalizers are not like C++ destructors!
- Let us repeat this again
  - **Finalizers are not like C++ destructors!**

Finalizers vs. Destructors

- Beware
  - Finalizers are not like C++ destructors!
- Let us repeat this again
  - **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];
   ...
}
```

Avoid Unnecessary Memory Retention

- Split the object
  - Finalize only what is necessary

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

Finalization and Scarce Resources

- Finalization to reclaim scarce resources
  - GC required before object is finalized
  - GCs triggered by memory usage
  - Memory is usually plentiful
  - The scarce resources will be exhausted before memory
- Recommendation: explicit management
  - Pool scarce resources
  - Return scarce resources to pool explicitly
- Finalization as a last resort!

finally {

- Using finalization has a score of other issues
  - e.g., synchronization
- Enumerated nicely in:
  - Destructors, Finalizers, and Synchronization
    - By Hans Boehm, POPL 2003
  - Finalization, threads, and the Java technology memory model
    - By Hans Boehm, TS-3281, 2005 JavaOne[SM] conference

}

Agenda

Garbage Collection Concepts

Programming Tips

Problems With Finalization

Using Reference Objects

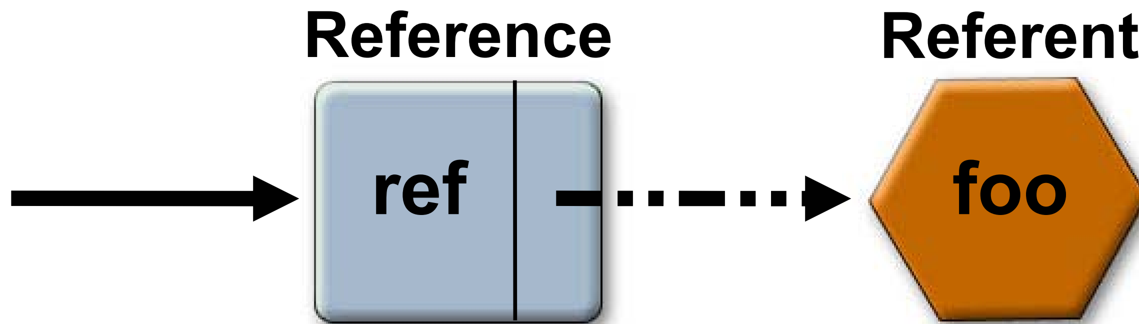
Memory Leak Avoidance

Conclusions

Reference Objects

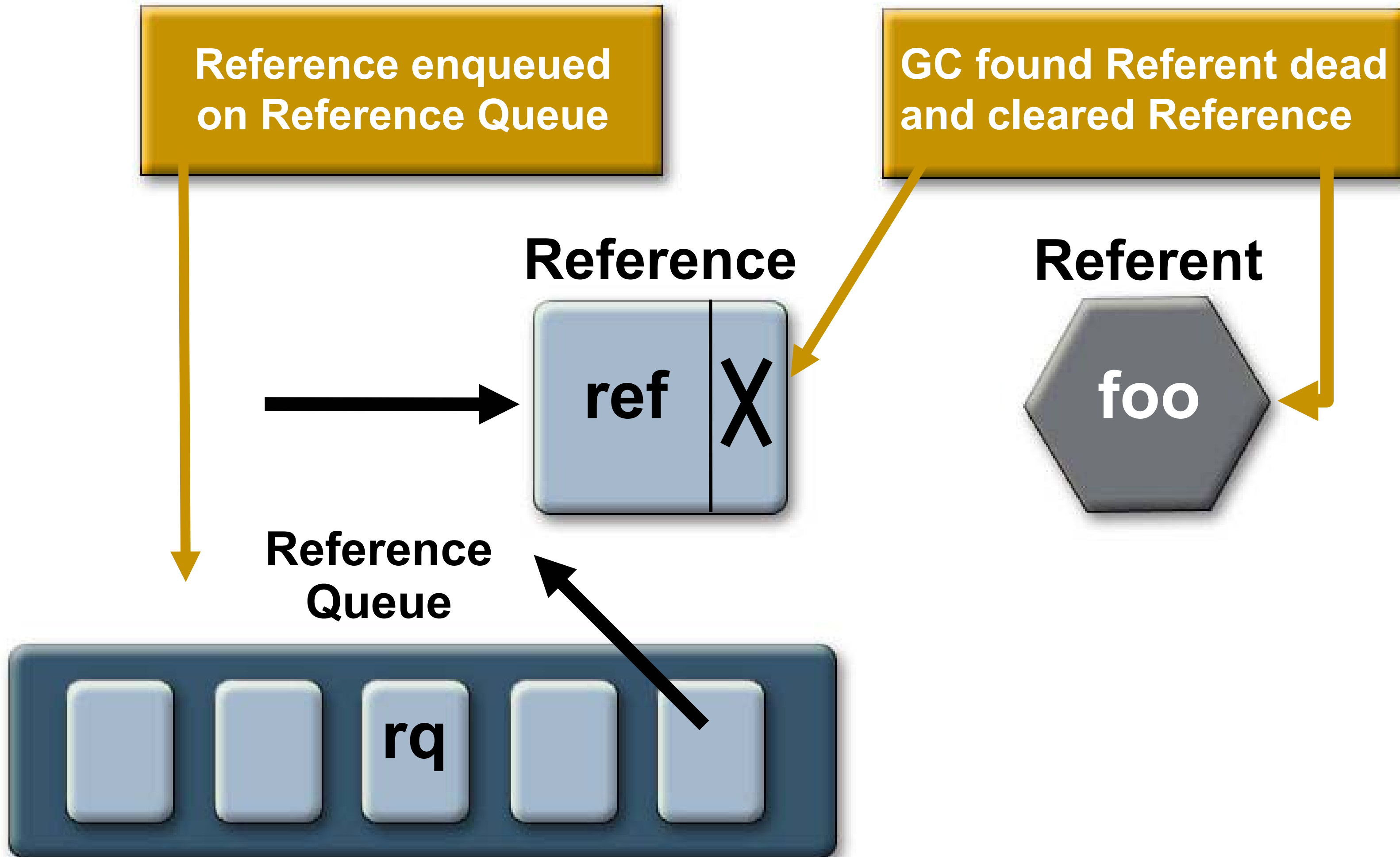
- Purpose
  - Postmortem hooks, more flexible than finalization
- Three types of reference objects
  - Weak references
  - Soft references
  - Phantom references
- All three
  - Can enqueue the reference object…
  - On a designated reference Queue…
  - When the GC finds its referent to be unreachable

Reference Objects: Illustration (1/2)



ref = new WeakReference(foo, rq);

Reference Objects: Illustration (1/2)



Weak References (1/2)

- 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 (2/2)

- 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 (1/2)

- 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 (2/2)

- 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*
  - *The GC will retain the referent until*
    - *It is explicitly cleared by the user, …*
    - *Or the reference object becomes unreachable*

Agenda

- Garbage Collection Concepts
- Programming Tips
- Problems With Finalization
- Using Reference Objects
- Memory Leak Avoidance
- Conclusions

Memory Leaks, Eh?

- But, the GC is supposed to fix memory leaks!
- The GC will collect all **unreachable** objects
- But, it will not collect objects that are still **reachable**
- Memory leaks in garbage collected heaps
  - Objects that are **reachable** but **unused**
  - Unintentional object retention

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

Memory Leak Sources

- Objects in the wrong scope
- Lapsed listeners
- Exceptions change control flow
- Instances of inner classes
- Metadata mismanagement
- Use of finalizers/reference objects

Objects in the Wrong Scope (1/2)

- Below, names really local to doIt()
  - It will not be reclaimed while the instance of Foo is live

```
class Foo {
   private String[] names;
   public void doIt(int length) {
        if (names == null || names.length < length)
             names = new String[length];
        populate(names);
        print(names);
   }
}
```

Objects in the Wrong Scope (2/2)

- Remember
  - Generational GCs love short-lived objects

```
class Foo {
   public void doIt(int length) {
         String[] names = new String[length];
         populate(names);
         print(names);
   }
}
```

Lapsed Listeners (1/2)

- Event listeners (Swing, AWT, etc.)

```
{
    ImageReader reader = new ImageReader();
    cancelButton.addActionListener(reader);
    reader.readImage(inputFile);
    // reader is still reachable as long as
    // cancelButton remains reachable
}
```

Lapsed Listeners (2/2)

- Need to explicitly remove it
  - When the listener object is not used any more

```
{
    ImageReader reader = new ImageReader();
    cancelButton.addActionListener(reader);
    reader.readImage(inputFile);
    cancelButton.removeActionListener(reader);
}
```

Exceptions Change Control Flow (1/2)

- Beware
  - Thrown exceptions can change control flow

```
    try {
        ImageReader reader = new ImageReader();
        cancelButton.addActionListener(reader);
        reader.readImage(inputFile);
        cancelButton.removeActionListener(reader);
} catch (IOException e) {
        // if thrown from readImage(), reader will not
        // be removed from cancelButton's listener set
}
```

Exceptions Change Control Flow (2/2)

- Always use finally blocks

```
ImageReader reader = new ImageReader();
cancelButton.addActionListener(reader);
try {
    reader.readImage(inputFile);
} catch (IOException e) {
    ...
} finally {
    cancelButton.removeActionListener(reader);
}
```

Instances of Inner Classes

- Instances of inner classes have an **implicit reference** to the outer instance

```
class ImageReader {
   class CancelListener implements ActionListener { ... }
   public ImageReader(JButton cancelButton) {
        CancelListener listener = new CancelListener();
        cancelButton.addActionListener(listener);
        // instance of CancelListener also 'holds onto'
        // the outer instance of ImageReader too
    }
}
```

JavaOne

Metadata Mismanagement (1/2)

- Sometimes, we want to:
  - Keep track of object metadata
  - In a separate map

```
class ImageManager {
    private Map<Image,File> map =
          new HashMap<Image,File>();
    public void add(Image image, File file) { ... }
    public void remove(Image image) { ... }
    Public File get(Image image) { ... }
}
```

Metadata Mismanagement (2/2)

- What happens if we forget to call remove(image)?
  - The image and file will never be removed from the map
  - Very common source of memory leaks
- We want:
  - The map to notice that the key is not reachable…
  - And purge the corresponding entry
- That's **exactly** what a WeakHashMap does

```
private Map<Image,File> map =
     new WeakHashMap<Image,File>();
```

Use of Finalizers/Reference Objects

- Both finalizers and reference objects
  - Can delay the reclamation of objects…
  - As well as everything reachable from them
- Due to slow processing/long length of:
  - Finalization queue
  - Reference queues
- 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

Agenda

Garbage Collection Concepts

Programming Tips

Problems With Finalization

Using Reference Objects

Memory Leak Avoidance

Conclusions

Conclusions

- We covered a series of tips on how to write
  - Simpler
  - More readable
  - More GC-friendly code
- You do not **have** to follow our advice
  - But you **will** get better GC performance if you do
  - We have helped a lot of customers with these tips

…And Don’t Forget!

“Everything should be made as simple as possible, but not simpler.”

—Albert Einstein

For More Information (1/2)

- 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

For More Information (2/2)

- FindBugs
  - http://findbugs.sourceforge.net
- 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

Java

JavaOne

Acknowledgments

- Many thanks to Brian Goetz

Q&A

John Coomes, Peter Kessler, Tony Printezis

Sun

JavaOne

Garbage Collection-Friendly Programming

John Coomes, Peter Kessler, Tony Printezis

Java SE Garbage Collection Group
Sun Microsystems, Inc.
http://java.sun.com/

TS-2906