Does Java™ Technology Have Memory Leaks?

Ed Lycklama
Chief Technology Officer, KL Group Inc.

Overview

- Garbage collection review
- What is a memory leak?
- Common patterns
- Tools & Demo
- Q&A
- Wrap-up

Garbage Collection in Java™ Technology

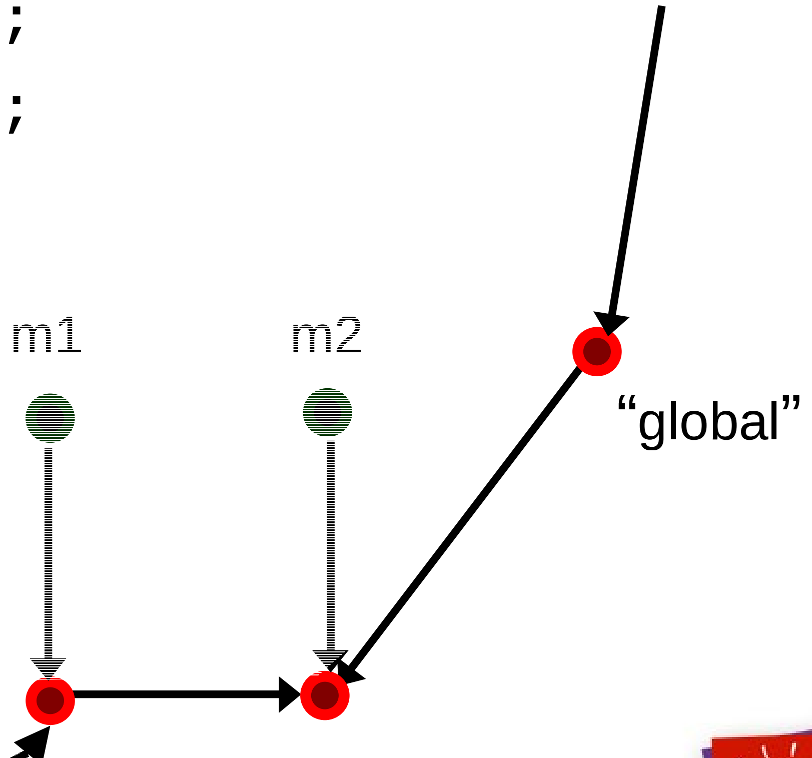
- As objects created, stored in Java heap
  - Set of allocated objects forms a directed graph
  - Nodes are objects, edges are references
- GC: remove objects no longer needed
  - Undecidable in general; use approximation
    - Remove objects no longer reachable
  - Start search at roots
    - Locals on thread stack
    - Static fields

Simple GC Example

```
Public void useless() {
MyObject m1 = new MyObject();
MyObject m2 = new MyObject();

m1.ref = m2;
global.ref = m2;

return;
}
```



now unreachable

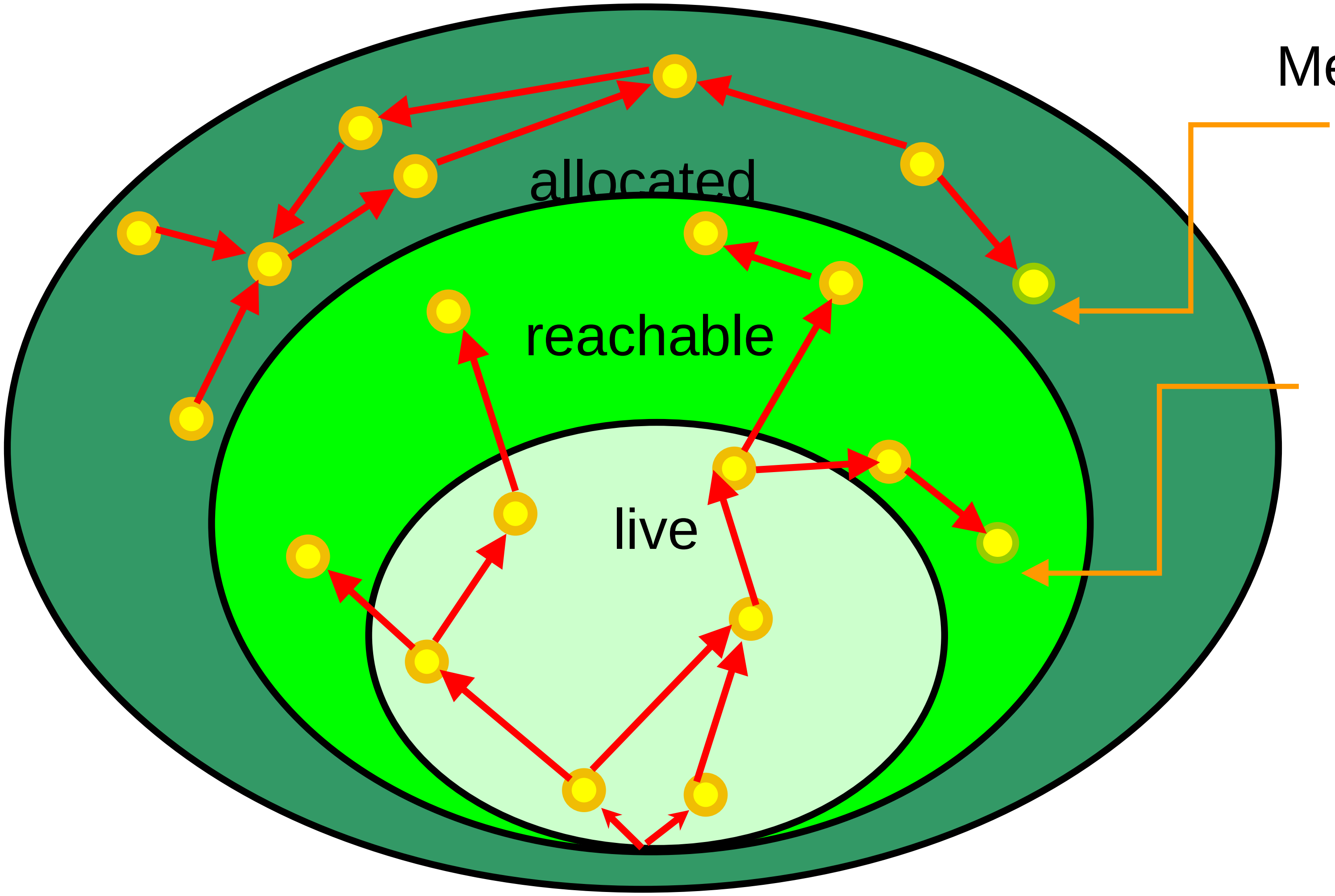
Garbage Collection Myths

- GC doesn’t handle reference cycles
  - Not based on reference counting (e.g. COM)
- Finalizer is like a destructor
  - Called when about to be collected
    - Never call it directly
  - May never be called
    - Depends on free memory, GC implementation
  - Finalizer may “resurrect” object!
    - Another object makes reference to it

What Is a Memory Leak?

- **Allocated**
  - Exists on the heap
- **Reachable**
  - A path exists from some root to it
- **Live**
  - Program may use it along some future execution path

What Is a Memory Leak?



Memory leak in C/C++

Memory leak in Java

C++ vs. The Java Programming Language

- Memory leak in C++
  - Object allocated but not reachable
    - Malloc/new, forgot about free/delete
  - Once unreachable, leak can't be fixed
- Memory leak in the Java programming language
  - Object reachable, but not live
    - Reference set, forgot to clear it
  - Object reachable, but code to fix leak may not be
    - e.g. private field

Nodes vs. Edges

- C++, manage nodes and edges
  - Explicitly add/remove nodes and edges
  - Dangling edges corrupt memory
  - Dangling node is a memory leak
- The Java programming language, manage the edges
  - Explicitly add nodes/edges, remove edges only
  - Nodes won’t go away unless it’s cut-off from graph

Less Common, More Severe

- Rarer than in C++ ...
  - GC does most of the work
- … but impact more severe
  - Rarely a single object, but a whole sub-graph
  - e.g. in Project Swing technology, path from any UI component to another (parent/child relationship)
  - Typically subclass, add other references
  - Single lingering reference can have massive memory impact

Need a New Term

- Loiterer
- Object remains past its usefulness
- Distinct from memory leaks in C++
- Some of the native Java libraries have conventional memory leaks
  - Programmers using Java technology can’t fix them

Lexicon of Loiterers

- Four main patterns of loitering
  - Lapsed listener
  - Lingerer
  - Laggard
  - Limbo

Lapsed Listener

- Object added to collection, not removed
  - e.g. event listener, observer
  - Collection size can grow without bound
  - Iteration over “dead” objects degrades performance
- Most frequent loitering pattern
  - Swing and AWT have had many
  - Occurs easily in any large framework
- C++: small loiterer or dangling pointer

Lapsed Listener Example

- Java 2 platform, desktop properties
  - awt.Toolkit.addPropertyChangeListener()
  - Toolkit is a singleton
  - Listeners will usually be shorter lifespan
  - Listener must call removePropertyChangeListener() when it is “being destroyed”

Lapsed Listener Strategies

- Ensure add/remove calls are paired
- Pay attention to object lifecycles
- Consider implementing a listener registry
- Beware of framework that claims to handle cleanup automatically
  - Understand assumptions made

Lingerer

- Reference used transiently by long-term object
  - Reference always reset on next use
    - e.g. associations with menu items
    - e.g. global action or service
- Not a problem in C++
  - Benign dangling reference

Lingerer Example

- Print action as singleton
  - Printable target;
  - call target.doPrint();
  - Target not set to null on completion
  - Target is a lingering reference
    - Target cannot be GC'ed until next print

Lingerer Strategies

- Don’t use a set of fields to maintain state
  - Enclose in object
    - Easier to maintain
    - One reference to clean up
- Draw state diagram
  - Quiescent state=no outgoing references
- Early exit methods or multi-stage process
  - Setup; process; cleanup

Laggard

- Object changes state, some references still refer to previous state
  - Also a hard-to-find bug
- Common culprits
  - Changing life-cycle of class (e.g. into a singleton)
    - Constructor sets up state
    - Caches expensive-to-determine object
    - State changes, cache not maintained
- C++: dangling pointer

Laggard Example

- List of files in directory
  - Maintains several metrics
    - Largest, smallest, most complex file
  - Change to new directory
  - Only largest and smallest updated
  - Reference to most complex is a laggard
    - Won’t notice unless code is coverage tested
    - Memory debugging would uncover it

Laggard Strategies

- Cache cautiously
  - Only expensive, frequently used calculations
    - Use a profiler to guide you
- Encapsulate in a single method
  - Do all calculations in one spot

Limbo

- Reference pinned by long-running thread
  - References on stack
  - GC doesn’t do local liveness analysis
- Common culprits
  - Thread stall
  - Expensive setup calls long-running analysis
- C++: placement of destructors

Limbo Example

```
Void method() {
  Biggie big =
     readIt();
  Item item =
     findIt(big);
   big = null;
  parseIt(item);
}
```

- Read file using std. parser
- Big consumes a lot of memory
- Item condenses it
- Iterate over elements of item
- Big can’t be GC’ed until method returns

Limbo Strategy

- Be aware of long-running methods
  - Profilers can help
- Pay attention to large allocations that precede it
  - Use a memory debugger to help find them
  - Add explicit null assignments to assist GC
- Blocked threads can also be a problem
  - Use a thread-analysis tool

Tools and Techniques

- ObjectTracker
  - Lightweight instance tracking infrastructure
  - Invasive: requires code modification
  - Find loiterers of a particular class
    - You decide which classes to track
    - Won’t tell you why it loiters
  - Relies on unique hashcode
    - Will not work in the Java 2 virtual machine
    - May not work in other VM’s

Tools and Techniques

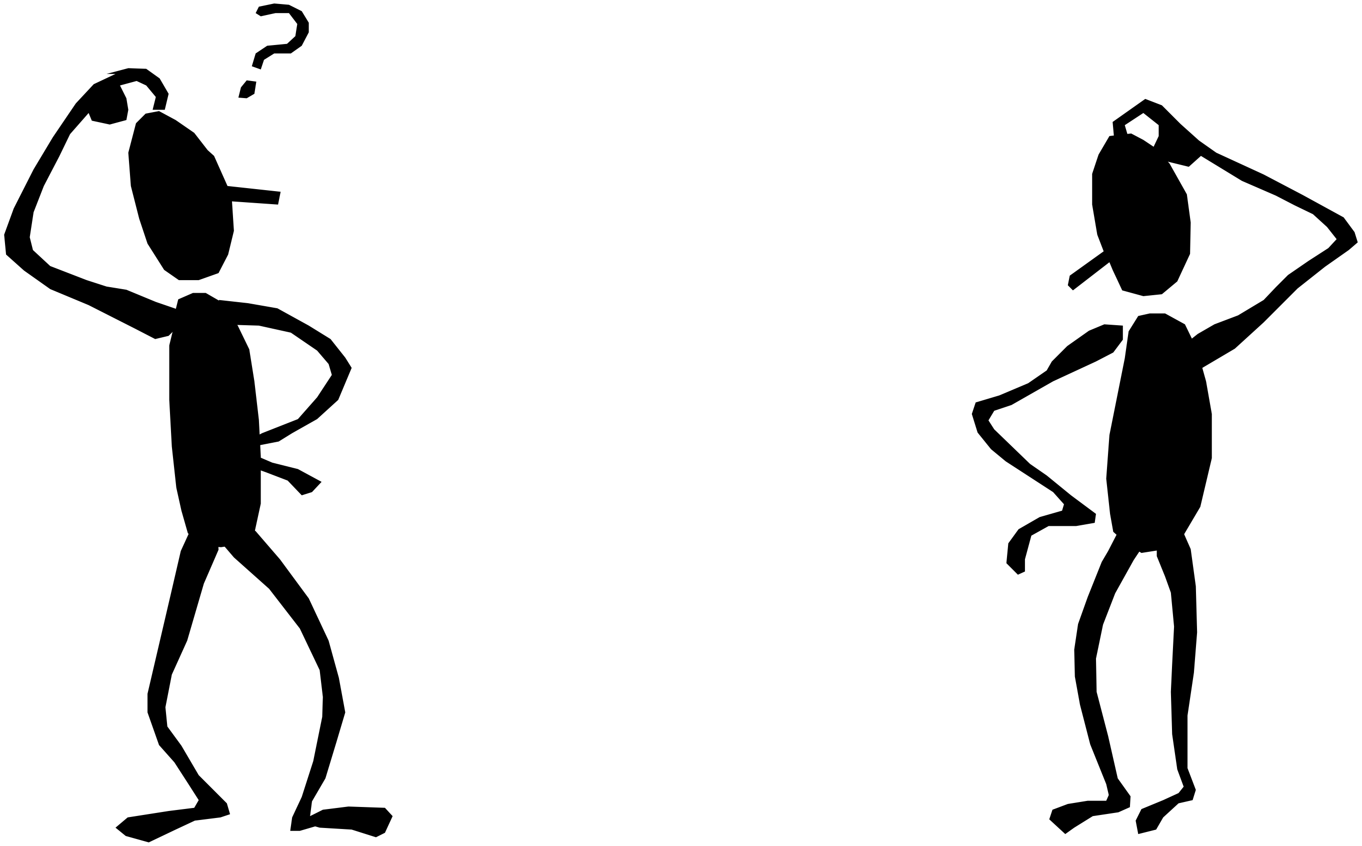
- Memory Debugger
  - No code modification required
  - Monitor overall heap usage
  - Understand allocation activity by class
  - Pinpoint excessive object allocation
  - Identify memory leaks (loiterers)

Tools and Techniques

- Finding Loiterers in a Memory Debugger
  - Track all instances of all classes
  - Each instance:
    - Time allocation occurred
    - References (incoming and outgoing)
    - Stack back-trace of allocation
  - Visualize reference graph back to roots
  - Checkpoint creation times

Demo

Questions?



Wrap-Up

- Most non-trivial Java technology-based programs have loiterers
  - GC is not a silver bullet
  - Manage the edges, not the nodes
- Loiterers different than memory leaks
  - Harder to find
  - Less frequent, but generally much larger

Wrap-Up

- Object lifecycles are key
- Build memory-management framework into your development practices
- Tools are indispensable for finding out why loiterers are occurring

Contact Info

- These slides and ObjectTracker at: http://www.klgroup.com/javaone
- See JProbe at KL Group’s booth #727
- Contact me: eal@klgroup.com

JavaOne℠

Sun's 1999 Worldwide Java Developer Conference℠