Memory Leaks in Java Applications: Different Tools for Different Types of Leaks

Gregg Sporar

Senior Staff Engineer

Sun Microsystems, Inc.

Austin, TX

gregg.sporar@sun.com

Goal

To understand the different types of tools and techniques available for finding memory leaks.

Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Disclaimers

Using Sun's JDK

Mostly talking about JDK 5 and 6

The demos use example code

Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...
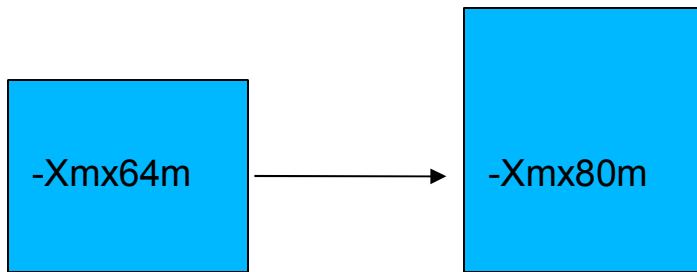
Q&A

Have You Ever Seen This?

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

And Then Do Your Users Do This?

So What Do You Do?

- **Increase the size of the heap**

-Xmx64m → -Xmx80m

- **And hope that the problem is fixed....**

But If That Does Not Fix the Problem....

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at app.leaking.UhOh(leaking.java:41)
        at app.leaking.WeHadHoped(leaking.java:51)
        at app.leaking.IfWeKeptIncreasing(leaking.java:55)
        at app.leaking.TheHeapSize(leaking.java:59)
        at app.leaking.ThenMaybeThisProblemWouldGoAway(leaking.java:63)
        at app.leaking.LooksLikeItHasNotGoneAway(leaking.java:67)
        at app.leaking.Bummer(leaking.java:61)
        at app.leaking.main(leaking.java:31)
```

Result: Additional User Frustration….
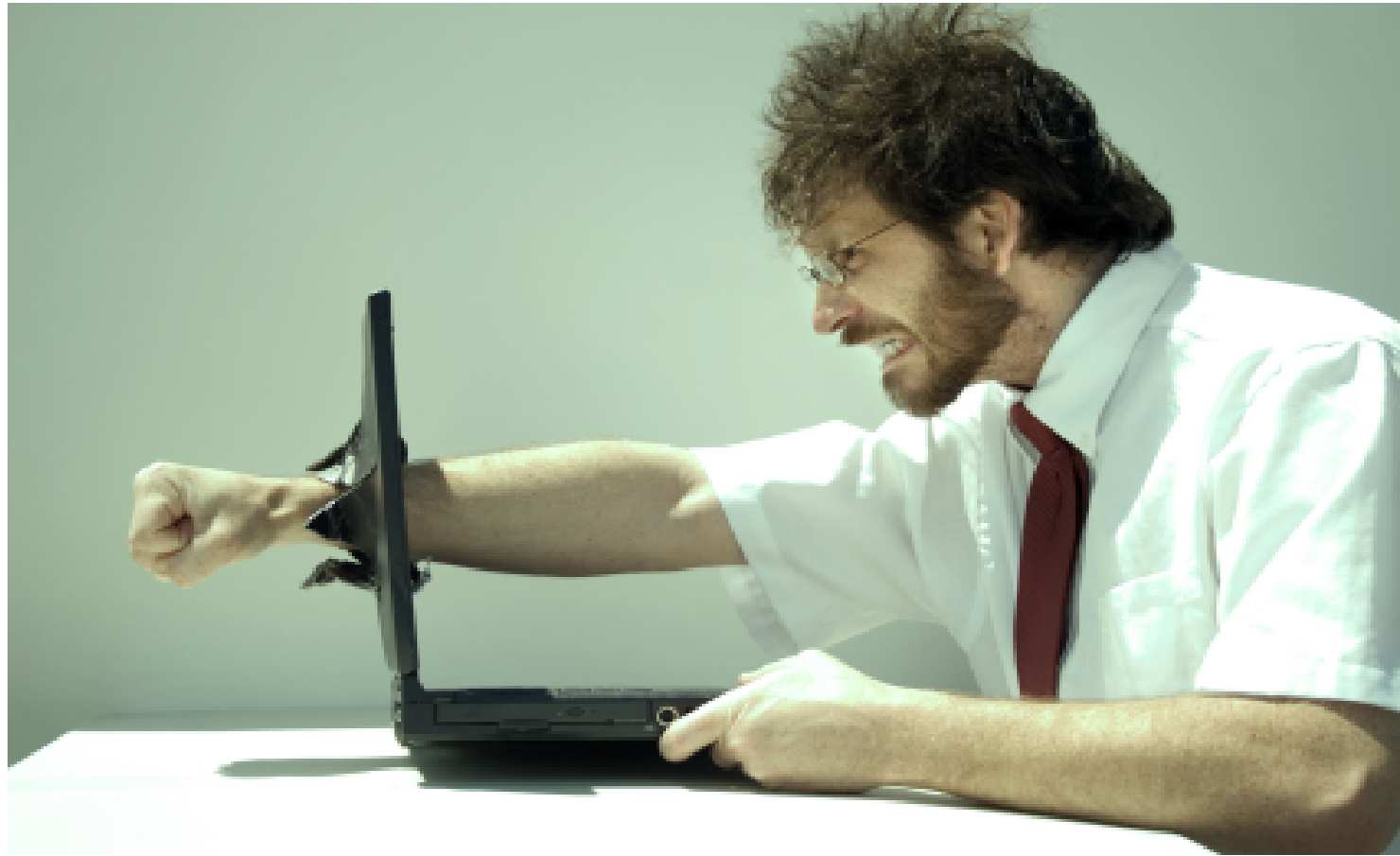
Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Observing the Problem

java -verbose:gc *YourApp*

[GC 189K->128K(1984K), 0.0016829 secs]
[Full GC 128K->128K(1984K), 0.0108424 secs]
[GC 10378K->10368K(12228K), 0.0004978 secs]
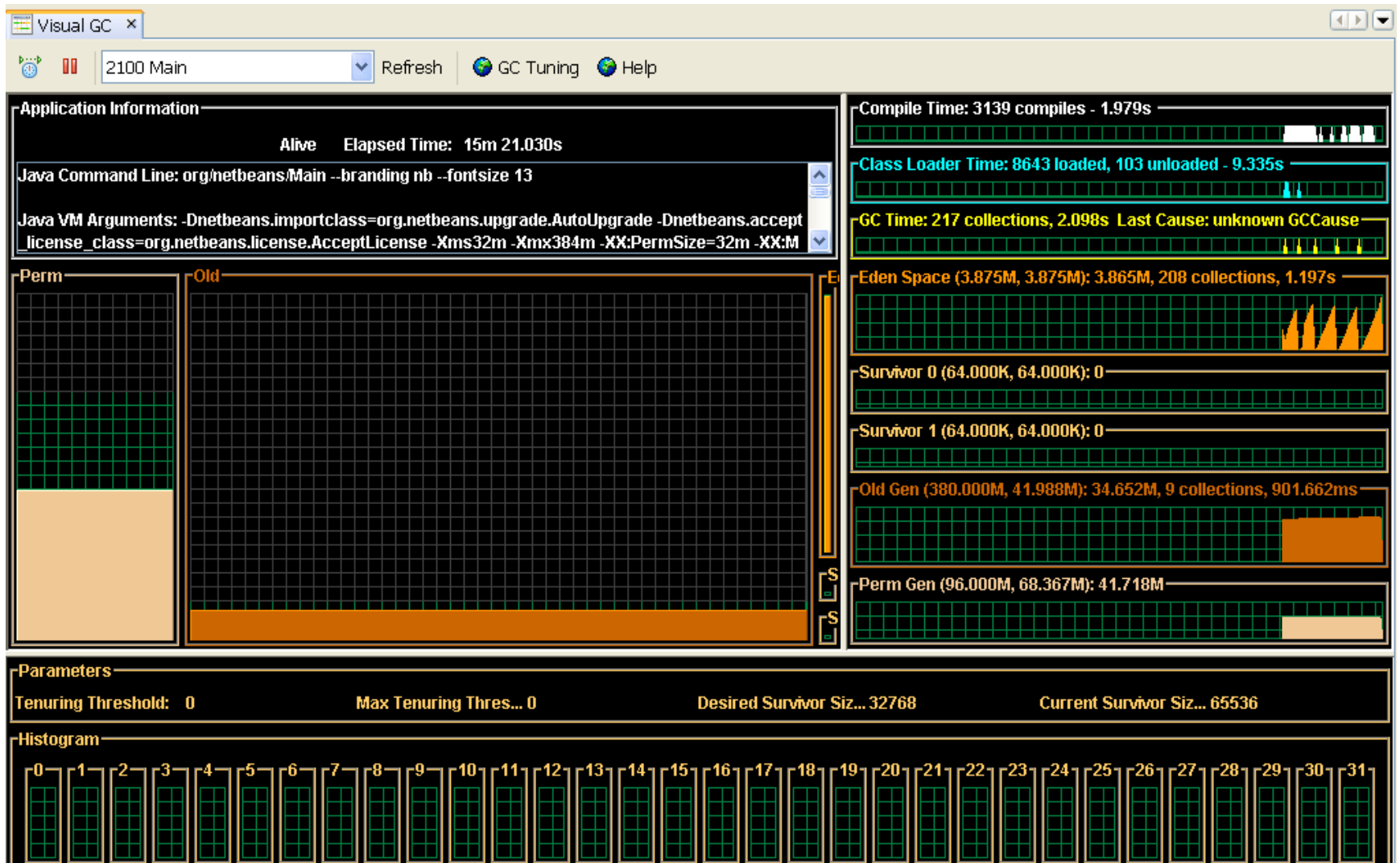**[Full GC 10368K->10368K(12228K), 0.0097568 secs]**
[GC 20633K->20608K(28872K), 0.0002025 secs]
[Full GC 20608K->20608K(28872K), 0.0097892 secs]
[GC 30896K->30848K(36972K), 0.0002380 secs]
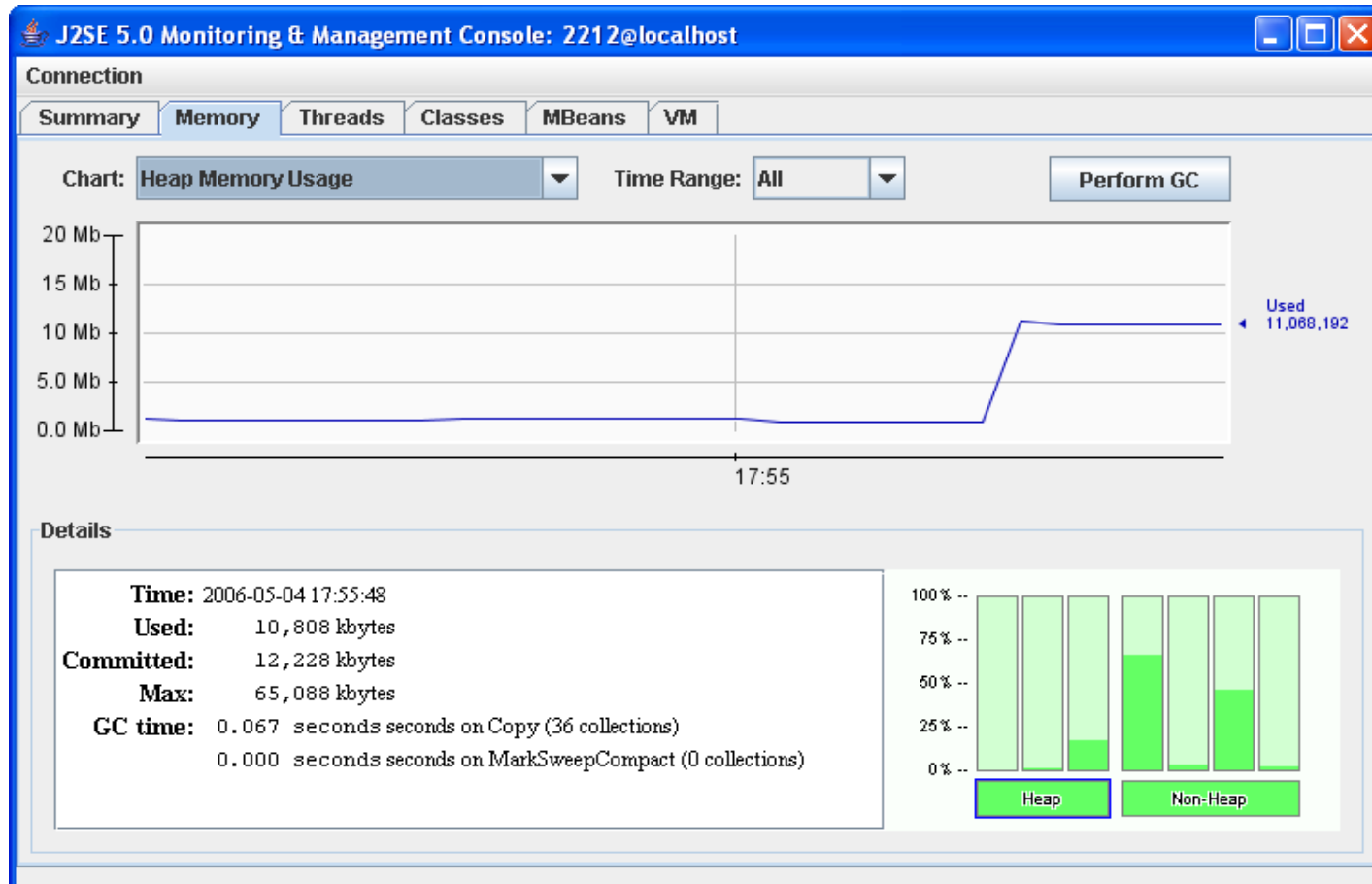**[Full GC 30848K->30847K(36972K), 0.0641433 secs]**
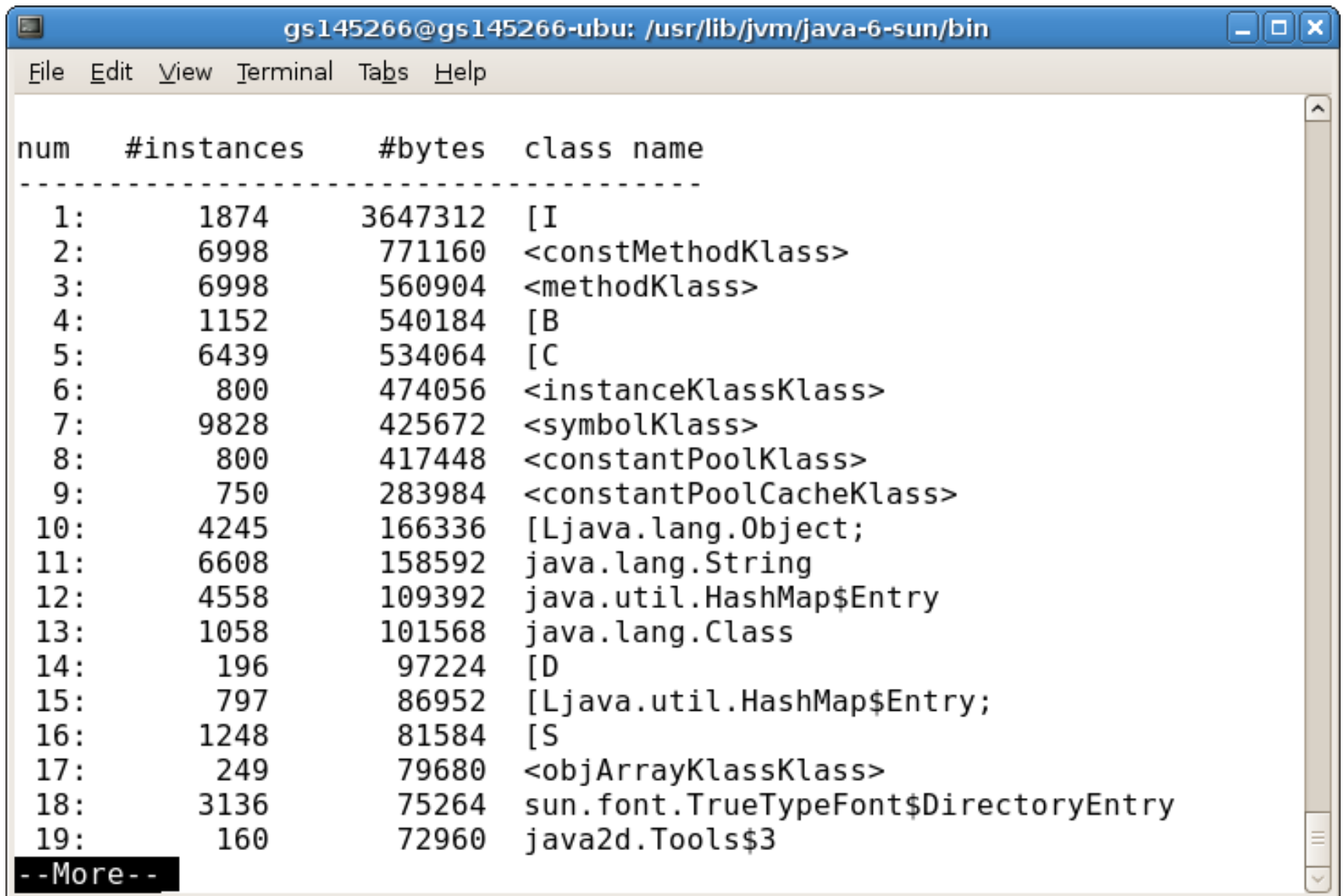
Observing the Problem (continued)

Observing the Problem (continued)

On JDK 1.4.2 or JDK 5:

java -Dcom.sun.management.jmxremote *YourApp*
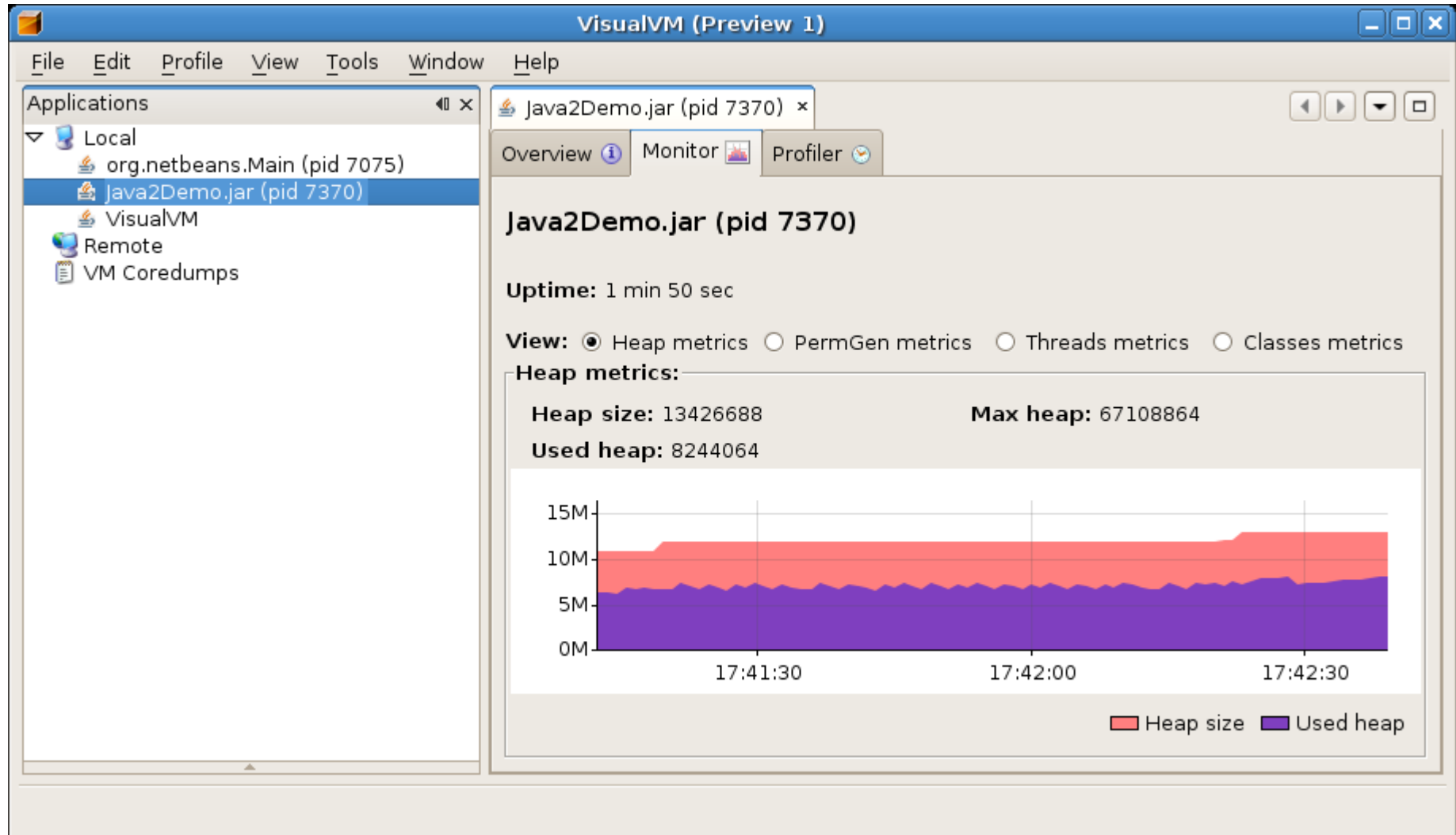
Observing the Problem (continued)

```
gs145266@gs145266-ubu: /usr/lib/jvm/java-6-sun/bin

File   Edit   View   Terminal   Tabs   Help

num      #instances       #bytes   class name
--------------------------------------------------
   1:         1874        3647312   [I
   2:         6998         771160   <constMethodKlass>
   3:         6998         560904   <methodKlass>
   4:         1152         540184   [B
   5:         6439         534064   [C
   6:          800         474056   <instanceKlassKlass>
   7:         9828         425672   <symbolKlass>
   8:          800         417448   <constantPoolKlass>
   9:          750         283984   <constantPoolCacheKlass>
  10:         4245         166336   [Ljava.lang.Object;
  11:         6608         158592   java.lang.String
  12:         4558         109392   java.util.HashMap$Entry
  13:         1058         101568   java.lang.Class
  14:          196          97224   [D
  15:          797          86952   [Ljava.util.HashMap$Entry;
  16:         1248          81584   [S
  17:          249          79680   <objArrayKlassKlass>
  18:         3136          75264   sun.font.TrueTypeFont$DirectoryEntry
  19:          160          72960   java2d.Tools$3
--More--
```

Observing the Problem (continued)

Observing the Problem (continued)

More Tools in JDK6:

- **-XX:+HeapDumpOnOutOfMemoryError**
  - **(Also available in JDK 1.4.2 and JDK 5)**

- **jhat**

- **jmap for Windows**

- **Stack trace on OutOfMemoryError**

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap
space
        at app.leaking.UhOh(leaking.java:41)
        at app.leaking.WeHadHoped(leaking.java:51)
        at app.leaking.IfWeKeptIncreasing(leaking.java:55)
        at app.leaking.TheHeapSize(leaking.java:59)
        at
app.leaking.ThenMaybeThisProblemWouldGoAway(leaking.java:63)
        at app.leaking.LooksLikeItHasNotGoneAway(leaking.java:67)
        at app.leaking.Bummer(leaking.java:61)
        at app.leaking.main(leaking.java:31)
```

Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Case Study: A Swing Application

- **Production Planning application**

- **Developed during 1999-2002**

- **JDK 1.2 (later moved to JDK 1.3 and then 1.4)**

- **~263,000 LOCs**

- **~1,600 Classes**

- **Memory leak found during QA, right *before* going live**

- **Easy to reproduce the problem, with the right data, but still not obvious what the cause was**

DEMO

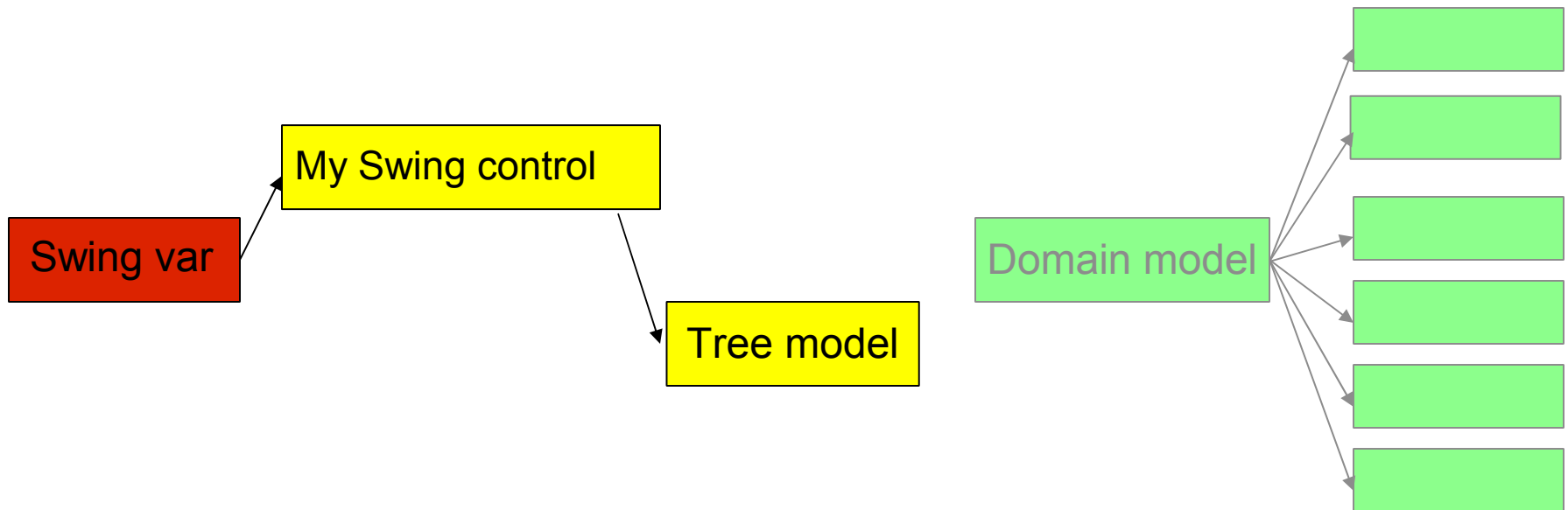**Inspecting the Heap With a Profiler**

So What Happened?

- **A bug in someone else's code prevented garbage collection of *my* objects**

- **4215796: RepaintManager DoubleBuffer can cause leak...**

Swing var → My Swing control → Tree model → Domain model → [six green boxes]
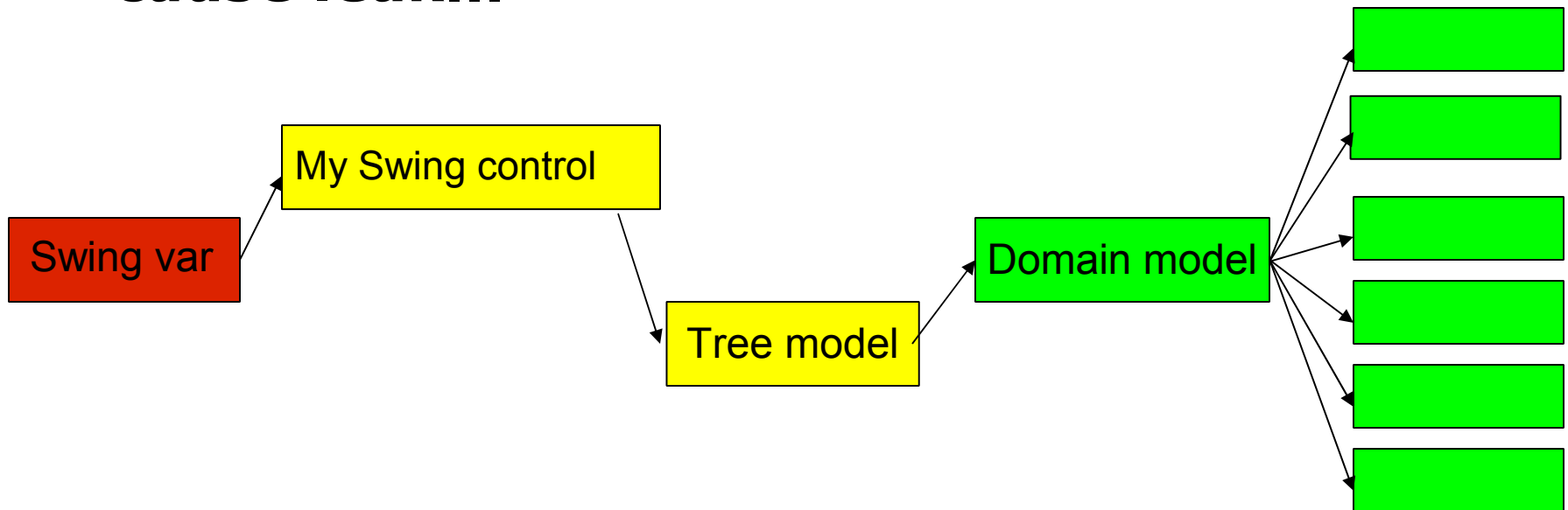
So What Happened? (cont'd)

- **Easy to work around – once you know the problem....**

Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Case Study: A Web Application

- **Hardware/software analysis system**

- **Developed during 2000-2004**

- **JDK 1.? (Later moved to JDK 1.4)**

- **>150,000 LOCs, which does not include:**
  - **the JSPs**
  - **a subsystem written in Perl**

- **Memory leak in the live, production system**

- **Hard to reproduce the problem - seemed to occur randomly**

DEMO

Using Instrumentation

So What Happened?

- ***Multiple*** **places in the code were allocating AnalysisResults objects, but only *some* of those allocations were causing leaks.**

: AnalysisResult allocated by the foreground process

: AnalysisResult allocated by the background process
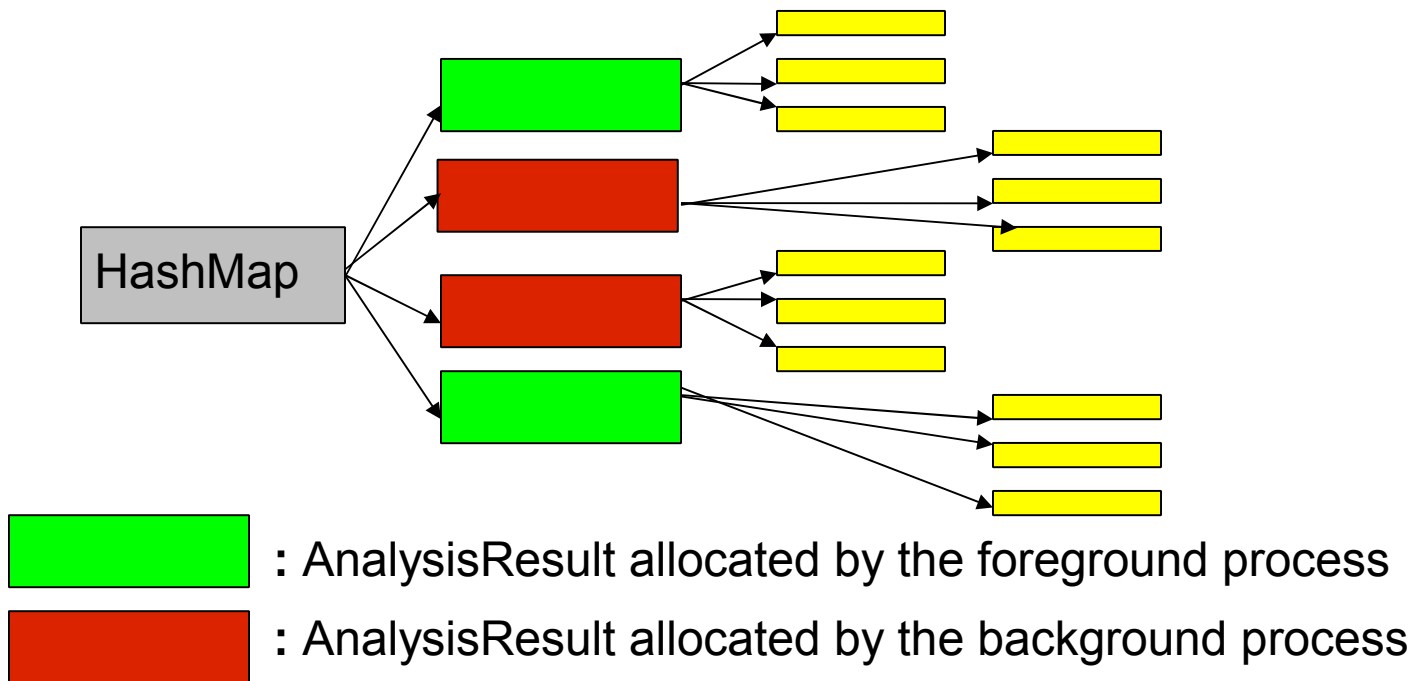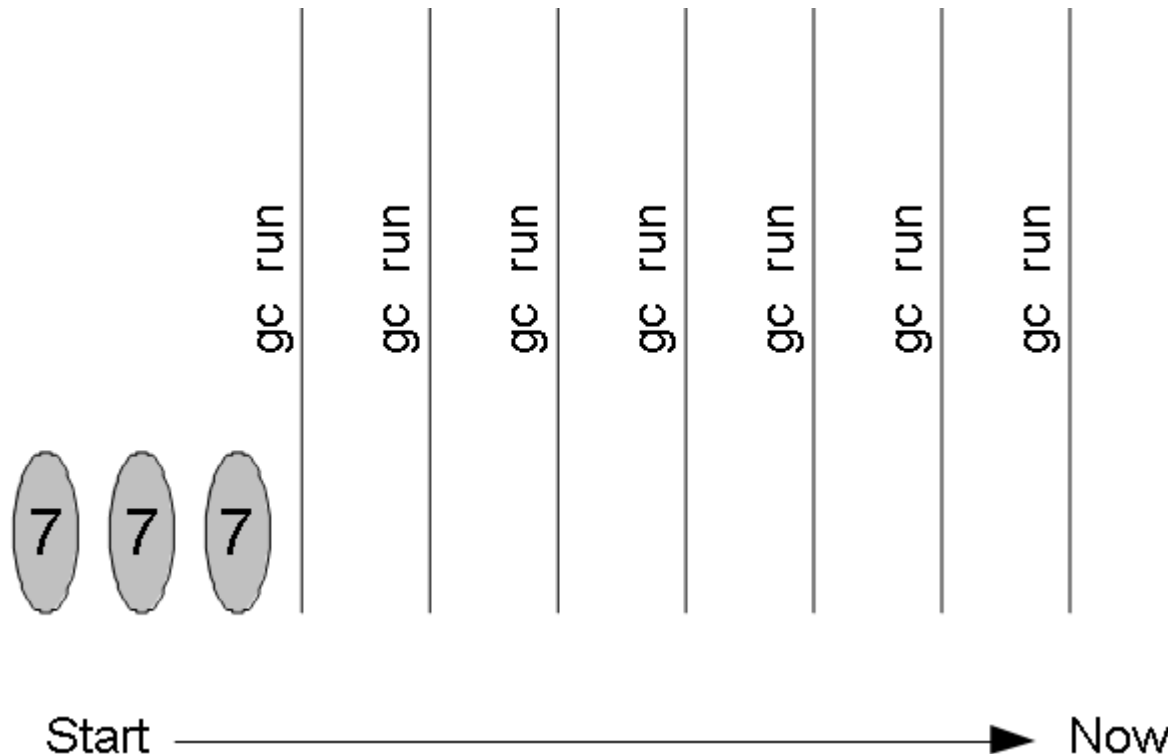
So What Happened? (continued)

The foreground code always removed its entries from the HashMap. The background code *never* removed its entries.
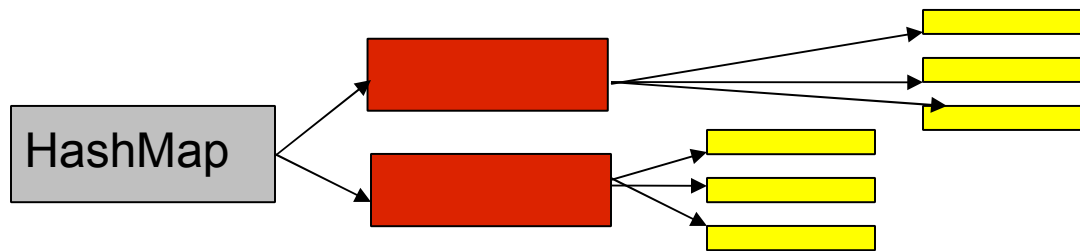
: AnalysisResult allocated by the background process

How Does "Generation Count" Help?

One Example of Healthy Behavior:

gc run   gc run   gc run   gc run   gc run   gc run   gc run
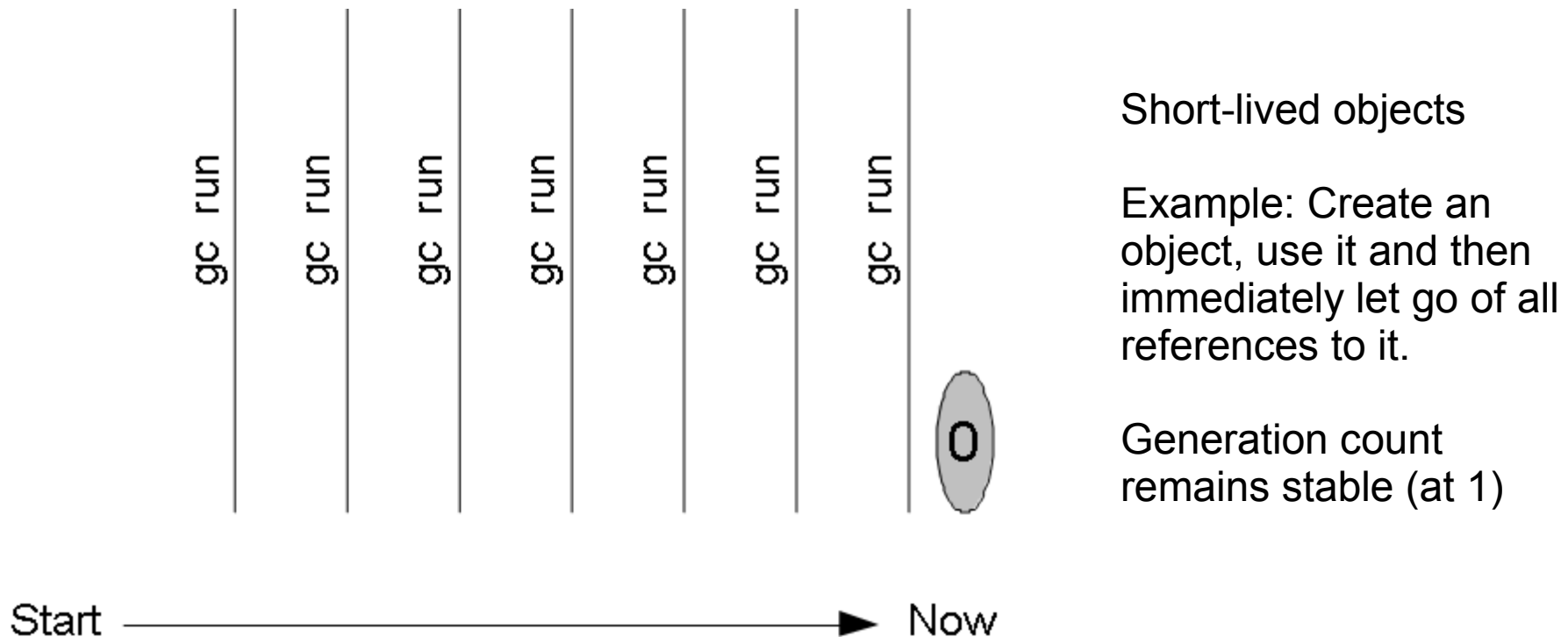
7 7 7

Start ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯➤ Now

Long-lived objects.

Example: Three object instances created at startup.

Their ages continue to increase, but generation count remains stable (at 1)

How Does "Generation Count" Help?

Another Example of Healthy Behavior:

Short-lived objects

Example: Create an object, use it and then immediately let go of all references to it.

Generation count remains stable (at 1)

gc run   gc run   gc run   gc run   gc run   gc run   gc run

O

Start ⟶ Now

How Does "Generation Count" Help?

Unhealthy Behavior (a Memory Leak):
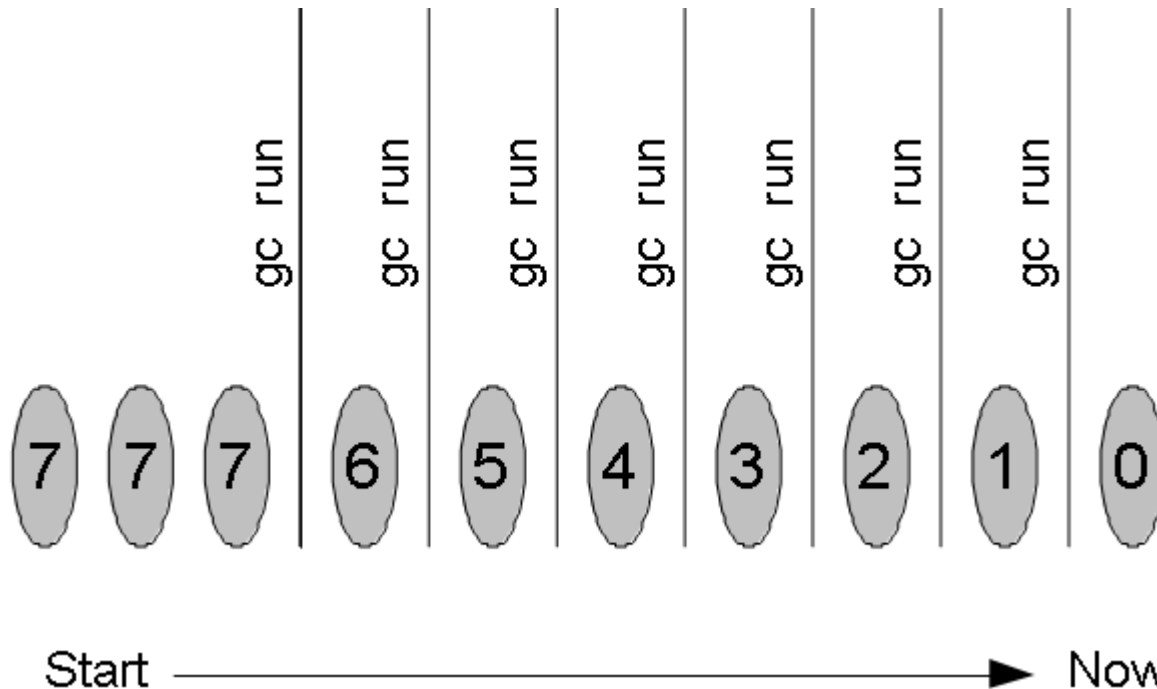
Example: Continue to allocate objects without letting go of all references.

Ten objects with eight *different* ages.

Generation count is *always increasing*.

Agenda

What's the Problem?

Observing the Problem

Inspecting the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Lessons Learned

- **Plenty of good, free tools available that provide a high-level view of the memory used by a Java application**

- **Beyond that, there are two broad categories:**
  - **Inspecting the Heap**
  - **Instrumentation**

Lessons Learned (continued)

Inspecting the Heap	Instrumentation

•Strengths:
    •Less impact on performance
    •Easy to see relationships between objects
 	
•Weaknesses:
    •No information about how the objects got onto the heap – or whether they should still be there
    •Large heap size can lead to information overload
    •Can be tough to use if you don't know the code
	•Strengths:
	    •Identifies objects that are candidate memory leaks
	    •Does not require as much knowledge of the code
	    •Scales well
	
	•Weaknesses:
	    •Introduces runtime overhead
	    •Does not show relationships between the objects

Agenda

What's the Problem?

Observing the Problem

Walking the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

Have You Ever Seen This?

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen full
```

These Guys Have....

**http://blogs.sun.com/edwardchou/entry/javaone_bof
_on_memory_leaks
http://blogs.sun.com/fkieviet/entry/javaone_2007**

The basics: heap memory generations

Young	Tenured	Permanent

Usually referred to as "the heap." Controlled by -Xmx and -Xms

Used by the JVM to store classes. Controlled by -XX:MaxPermSize and -XX:PermSize
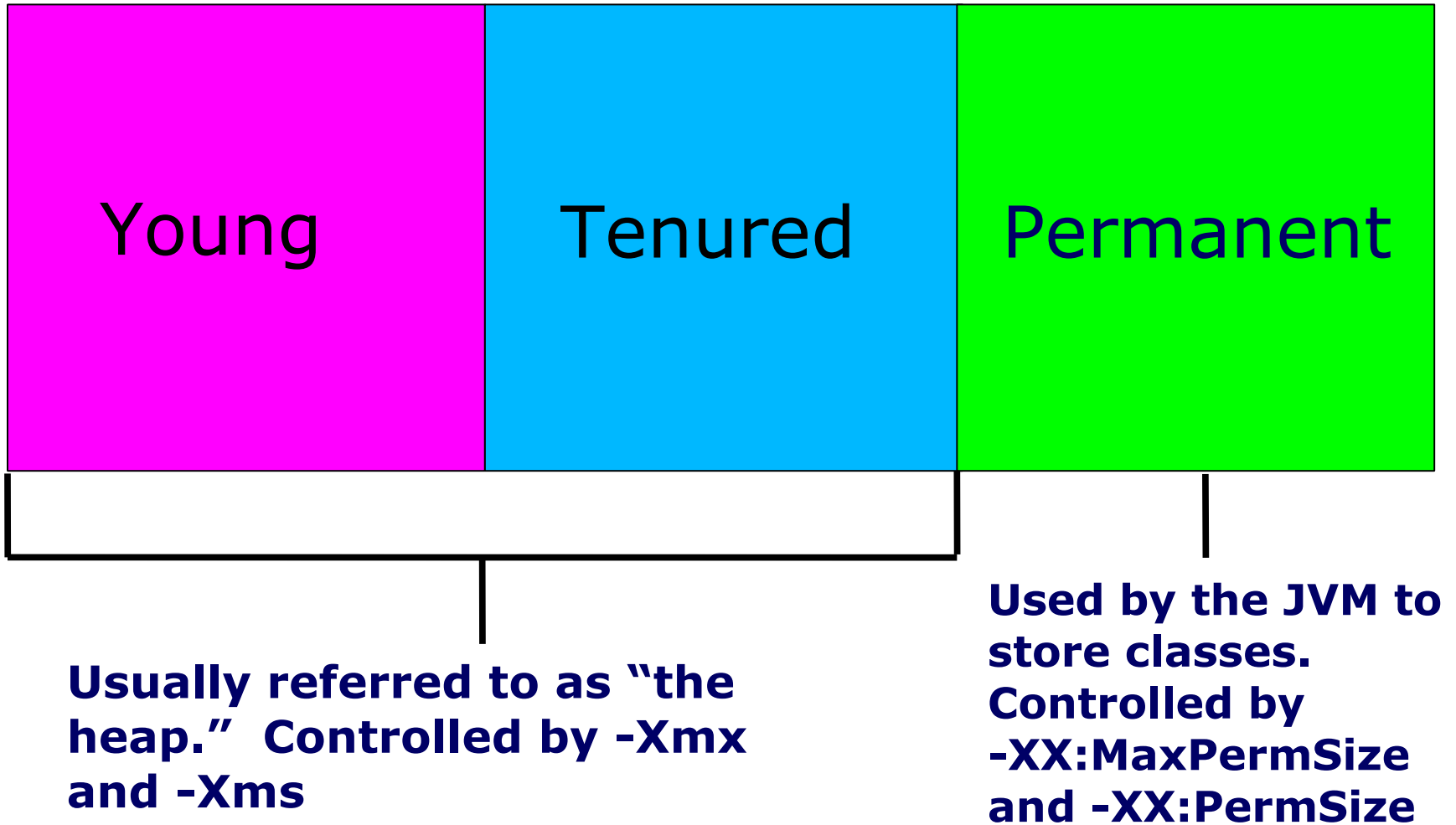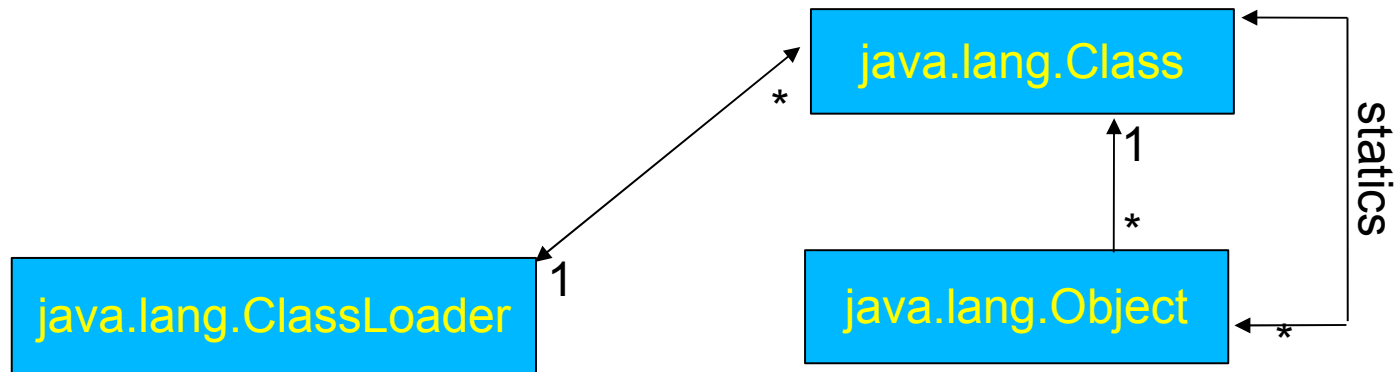
The basics: classes and classloaders

- **Each object is an instance of a class**

- **A class itself is an object (class object)**
  - **instance of the class** `Class`

- **Each class object references its classloader**

- **A classloader references all classes it loaded**

- **Class objects hold static members**

The basics: classes and classloaders

Why use classloaders?

- **Containers use classloaders to**
    - **dynamically load applications**
    - **isolate applications from each other**
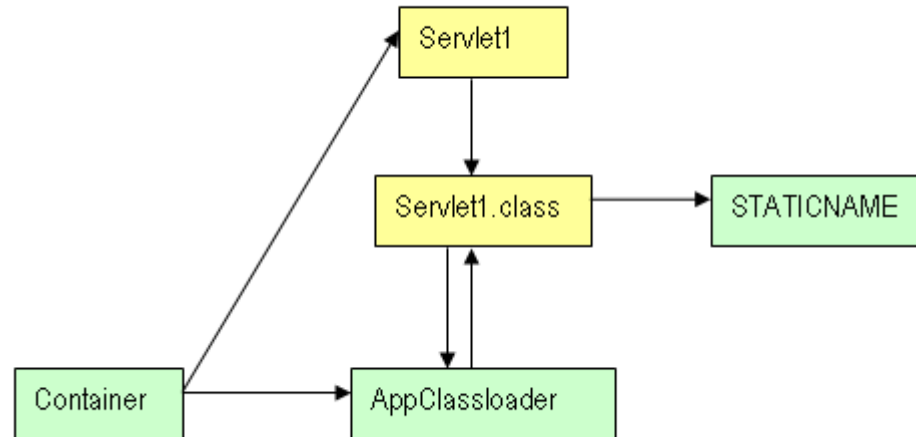   - **dynamically unload applications**

Example: empty servlet

```java
package com.stc.test;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet1 extends HttpServlet {
  private static final String STATICNAME = "Simple";
  protected void doGet(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {
    // nothing
  }
}
```
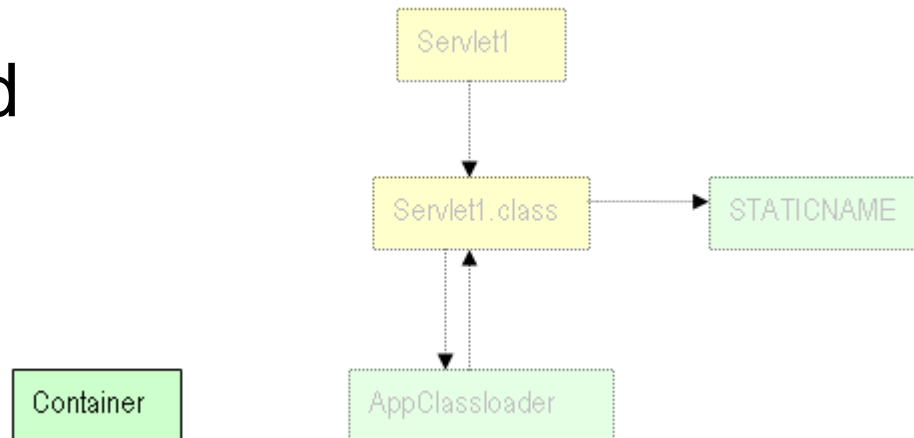
Deployed

Undeployed

Classloader leaks

- **A classloader cannot be garbage collected if any of the instances of any of the classes it loaded are *reachable*.**

- **Such a classloader keeps *all* its classes with all their static members in memory.**

  - **Not immediately apparent from a memory dump what is a leak and what is not.**

  - **Cause of the leak difficult to find.**

Example: a leaking servlet
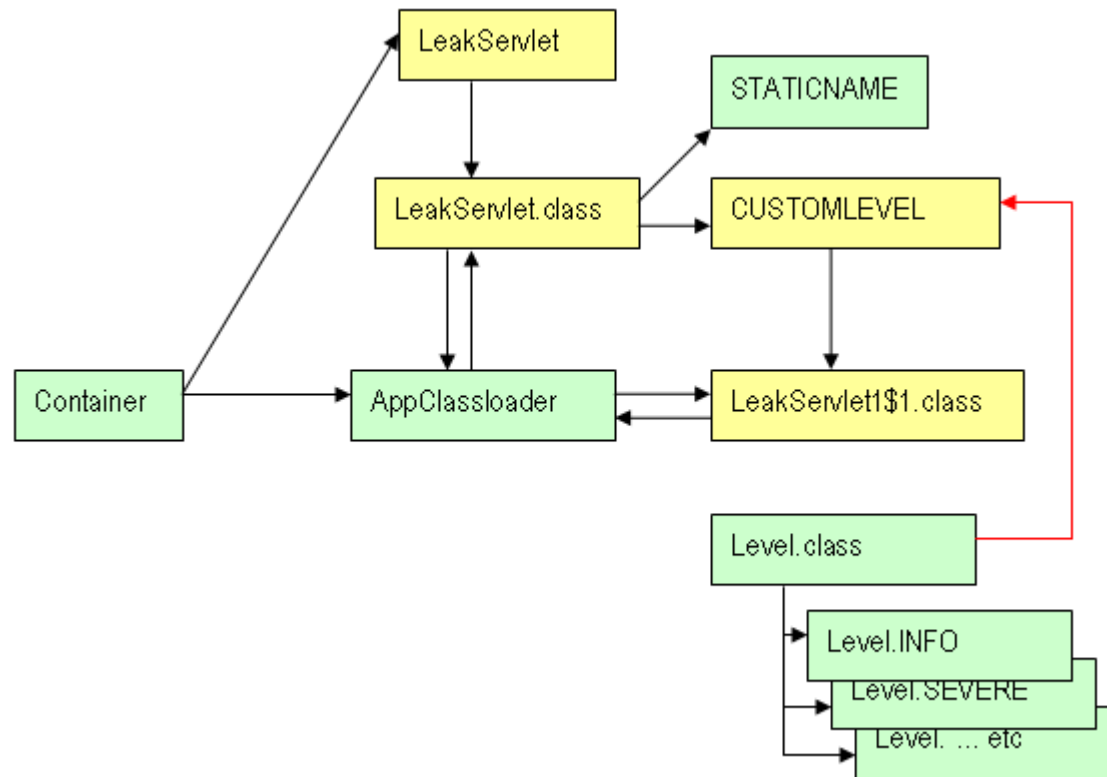
```java
package com.stc.test;

import java.io.*;
import java.util.logging.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LeakServlet extends HttpServlet {
  private static final String STATICNAME = "Leak!";
  static Level CUSTOMLEVEL = new Level("OOPS", 555) {};
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
  // Log at a custom level
  Logger.getLogger("test").log(CUSTOMLEVEL, "x called");
 }
}
```

java.util.logging.Level class

```java
private static List known = new ArrayList();

protected Level(String name, int value) {
  this.name = name;
  this.value = value;
  synchronized (Level.class) {
    known.add(this);
  }
}
```
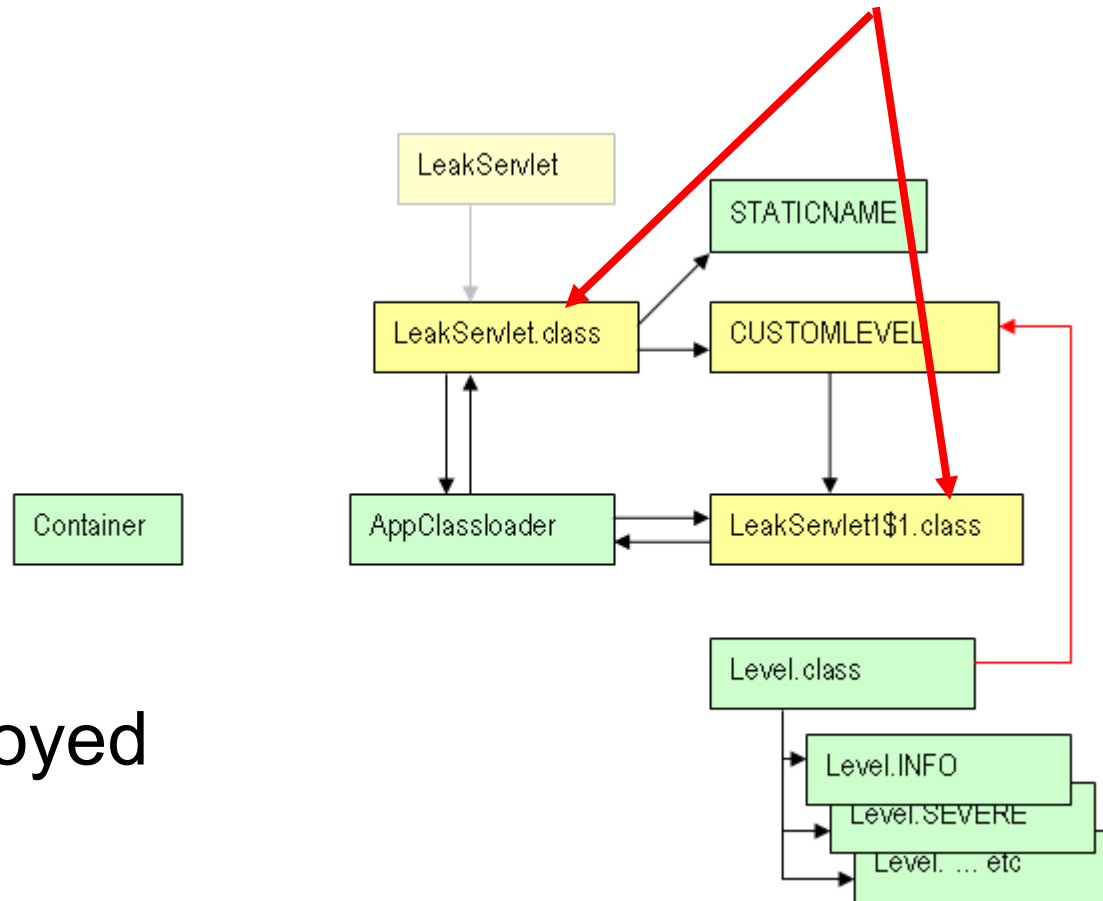
Deployed

A Leak...
These 2 are taking up space in PermGen

Undeployed

Reality:

- **Hundreds or *Thousands* of leaked classes**

- **Thousands of leaked objects**

- **Bafflement...**

Java Profilers

- **Take memory snapshots**

- **Find reference chains to GC root objects**

- **Most see class objects as GC root objects – so they are not very helpful**

DEMO

Inspecting the Heap With jhat

Agenda

What's the Problem?

Observing the Problem

Walking the Heap

Using Instrumentation

Lessons Learned

An Additional Problem...

Q&A

A Common Question

- **Are there any proactive techniques that will eliminate these problems?**

  - **Static Analysis tools: Might help, but in general not very useful....**

  - **Weak References: Can be very helpful – http://weblogs.java.net/blog/enicholas/ archive/2006/05/understanding_w.html**

Resources

Software Test & Performance

April, 2007 issue

May, 2007 issue

By Gregg Sporar and A. Sundararajan

Does your Java application become slower as it runs? The cause could be a memory leak. In Java applications, memory leaks usually end up causing performance problems. At the least, they decrease the CPU time available for running your application, slowing its ability to respond. In a worst-case scenario, your application stops responding altogether.

Solving memory leak problems in Java applications requires a variety of tools and techniques. There is no single solution—different techniques are appropriate for different situations.

What Is a Memory Leak?

The Java programming language doesn't require the developer to directly manage memory allocations—it doesn't even allow it. Instead, programs use the *new* operator to allocate objects on

an OutOfMemoryError will stop responding to requests.

A common approach for resolving an OutOfMemoryError is to restart the application and use a JVM option to specify a larger heap. This is reasonable during development, when you're determining the heap requirements of the application. The heap should grow as your application processes requests. As the load declines, the heap usage should

Brain Drain In Your Java Apps?

It's Not Just The Younger Generations

By Gregg Sporar, A. Sundararajan and Frank Kieviet

Like the rest of us, you probably don't think much about aging parts under the hood— until a hose starts to leak. Then it suddenly becomes that trip's urgent issue, sometimes forcing you off the road with disastrous results. So what

operator; the term *heap* is commonly used to refer to the combination of the young and tenured generations.

The permanent generation, however, is very different. The JVM employs it to hold the classes that your application uses. Your application's classes are loaded by a class loader, which the JVM provides so that you don't have to be concerned

refer to Sun's reference implementations, version 1.4.2 or higher.

What Causes PermGen Leaks?

If your application encounters an OutOfMemoryError, it could be the permanent generation that ran out of space. Starting with JDK 5, the OutOfMemoryError message includes

Both available at http://www.stpmag.com/

Q & A

gregg.sporar@sun.com