



# Precise Identification of Memory Leaks in Java Production Environments

**Business White Paper**

February, 2011

## Have you ever seen this situation?



If the answer is yes, you may have a case of memory leak induced insomnia, but fortunately we've got a cure for what ails you.

This whitepaper addresses your concern for everything you need to know to ease your suffering including what memory leaks are, why they happen, and how to fix them.

### Anatomy of memory leaks

It is true that Java code does not require the programmer to be responsible for memory management cleanup, and that it automatically garbage collects unused objects. The key point to remember is that an object is only counted as being unused when it is no longer referenced. All objects in Java are references and they form an intricate web of associations. These associations can become unmanageable and could eventually lead to memory leaks.

### Mea Culpa: The three deadly sins causing memory leak

- 1. Unknown or unwanted object references:** These objects are no longer needed, but the garbage collector cannot reclaim the memory because another object still refers to it.
- 2. Long-living (static) objects:** These objects stay in the memory for the application's full lifetime. Objects tagged to the session may also have the same lifetime as the session, which are created per user and remain until the user logs out of the application.
- 3. Unnecessarily high memory usage:** This is caused by implementations consuming too much memory. Large amount of unused state information (managed for "user comfort") or unbound or inefficiently configured caches are the typical causes for high memory usage.

### Magic word: Reduced MTTR (Mean-Time-To-Resolution)

Availability has become a critical component for organizations because business processes can quickly come to a halt when a computer system is down. Therefore, shorter Mean Time To Resolution (MTTR) is critical. MTTR includes problem diagnosis and problem repair. Typically, 80% of MTTR is spent in the diagnosis phase. However, when potential problem areas are better managed, the time required to successfully diagnose the problem is much lower, and consequently, MTTR is lower as well.

### Practical problems that lengthen MTTR

With traditional approaches like heap dumps, profilers etc. problem diagnosis and repair for a memory leak will take a long time to fix resulting in a long MTTR. In our opinion, the main reasons for a long MTTR are:

- It can be difficult to reliably reproduce an issue, and a lot of time is typically required to reproduce an issue before you can really start addressing it.
- Heap dump generates a huge file size and you might not be able to load it because you do not have enough memory in your development box.
- While profilers are great, they also induce performance overhead, which is not acceptable in a production environment.
- Information abundance problem increases diagnosis time. Additionally it becomes very tedious to analyze information from thousands of classes making it easier to find a memory leak but making the isolation of the leaking component equally difficult.
- Even if you are lucky and you locate the leaking component (at the cost of time spent in diagnosis), finding out the code path or business transaction would still take considerable amount of time. And we are still not considering the time spent by your resources in identifying these leaks.
- A common route of escape is to “throw hardware at the problem”. Hardware is thought by many to be cheap, but scalability is not cheap at all.

### Recommended practice - Smart capacity planning

It is a common assumption that “OutOfMemory” errors in production are the result of a leakage. But the reality is that a significant number of errors are due to memory capacity issues. These are typical in situations with high workload concurrency and deep/prolonged call chains.

It is always a dream situation when perfect capacity planning helps avoid memory leaks in production. For this you need to have already obtained object allocation counts and sizes per activity. Naturally, this data should not be obtained in production (just like complete memory heap dumps should not be taken in production) but during testing for the purpose of capacity planning.

### Memory leak resolution in five minutes

AppDynamics’ memory management suite provides a panacea for all your memory leak problems with the help of three powerful features:

- (i) Automatic leak detection
- (ii) Object instance tracking
- (iii) Custom memory structures (CMS) monitoring

These three features serve to resolve any and every memory leak or memory capacity planning problem in as little as five minutes time-frame.

Listed below are the compelling reasons why AppDynamics is a perfect solution to all your memory leak woes:

1. Requires no configurations (the pre-built intelligence automatically tracks different application and system classes).
2. Provide histogram of every object in the JVM.
3. Monitors memory intensive data structures like third-party caches.
4. Helps you plan memory capacity in test environment, so that you can have the “Smart Capacity Planning” for your resources.
5. Low overhead of running Appdynamics (overhead is just 2% in production environment).
6. Finally, you don’t have to manage the APM solution; you can now focus only on your applications.

### AppDynamics’ memory management solution in action

Following snapshots show how AppDynamics captures potentially leaking Collection object and identify memory thrash problems in few steps.

#### Identify and troubleshoot memory leak

Automatic leak detection not only identifies potential memory leaks but also allows the user to isolate and troubleshoot the memory leak by finding the actual code path and business transaction causing the problem. Automatic leak detection automatically tracks the Java Collections object and can also be customized to track custom memory structures like third-party caches.

#### 1. Memory Trend

Average Utilization **66%**  
Current Utilization **66%**  
last 15 minutes

#### 2. Identify Leaks

Class	Collection Size	Potentially Leaking
java.util.HashMap	10,034	X Yes
java.util.HashMap	14,321	X Yes
java.util.HashSet	11,501	X Yes

#### 3. Isolate Leak Detection

Content Summary

Classname	Count
java.lang.String	2650164
com.appdynamics.qa.model.Address	330036
com.appdynamics.qa.model.Order	165018
java.lang.Object[]	1

#### 4. Pinpoint Root Cause

Code Path	Occurrences
<b>com.appdynamics.qa.manager.ordermanager.getOrder(ordermanager.java) at com.appdynamics.qa.manager.orderscheduler.executeInternal(orderscheduler.java:23)</b>	<b>314</b>
at org.springframework.scheduling.quartz.quartzjobbean.execute(quartzjobbean.java:66) at org.quartz.core.jobrunshell.run(jobrunshell.java:216) at org.quartz.simpl.simplethreadpool\$workerthread.run(simplethreadpool.java:549)	
<b>com.appdynamics.qa.manager.ordermanager.getOrder(ordermanager.java) at com.appdynamics.qa.manager.orderscheduler.checkStatus(orderscheduler.java:45)</b>	<b>307</b>
at com.appdynamics.qa.manager.orderscheduler.getOrderStatus(orderscheduler.java:27) at org.springframework.scheduling.quartz.quartzjobbean.execute(quartzjobbean.java:66)	

  

Business Transactions accessing this Collection

Transaction Name	Occurrences
<b>/ACMEBookstore/OrderProcessing/GetCart</b>	<b>314</b>
<b>/ACMEBookstore/OrderProcessing/GetOrderStatus</b>	<b>307</b>

**Better capacity planning**

Memory thrash problems are indicators of inefficient utilization of temporary memory, which often arise due to improper capacity planning. AppDynamics helps in identifying the memory footprint right up to the code and business transaction level. As a result, AppDynamics not only locates the code pattern—but also helps you understand the impact of the affected component on your business.

### 1. Memory Trend

Average Utilization **66%**  
Current Utilization **66%**  
last 15 minutes

### 2. Monitor Memory Usage

Class	Current Instance	Shallow Size	Instance Count Trend
org.apache.commons.httpclient.He	440	10,560	
org.apache.commons.httpclient.He	112	2,072	
org.apache.commons.httpclient.He	48	1,152	
org.apache.commons.httpclient.Na	32	512	
org.apache.commons.httpclient.pa	31	496	
org.apache.commons.httpclient.Ho	24	576	

### 3. Isolate Memory Thrash

### 4. Pinpoint Root Cause

Code Path	Occurrences
at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)	25700
at org.apache.catalina.core.applicationfilterchain.internaldoFilter(applicationfilterchain.java:269)	
at org.apache.catalina.core.applicationfilterchain.doFilter(applicationfilterchain.java:188)	
at org.apache.catalina.core.standardwrappervalue.invoke(standardwrappervalue.java:213)	
at org.apache.catalina.core.standardcontextvalue.invoke(standardcontextvalue.java:172)	
at org.apache.catalina.core.standardhostvalue.invoke(standardhostvalue.java:127)	
at org.apache.catalina.valves.errorreportvalue.invoke(errorreportvalue.java:147)	
<b>com.appdynamics.qa.model.order.&lt;init&gt;(order.java:16)</b>	<b>230179</b>
<b>at com.appdynamics.qa.model.order.getOrderInfo(order.java:45)</b>	
<b>at com.appdynamics.qa.manager.ordermanager.addUserOrder(ordermanager.java:45)</b>	

  

Business Transactions creating instances of this Class

Transaction Name	Occurrences
<b>/ACMEBookstore/GetItemsFromCart</b>	<b>230179</b>
/ACMEBookstore/GetOrderInformation	
/ACMEBookstore/GetUserOrders	



For more information, contact  
 SCL  
 Jubilee House  
 Jubilee walk  
 Crawley  
 RH10 1LQ  
 United Kingdom  
 Tel: +44 1293 403636  
 Web: www.scl.com/appdynamics