Chris Bailey
Java Support, Monitoring and Serviceability

AVP Expert Call Series

Exclusively for AVP Clients

Java Runtime Memory Management

How the Java Runtime uses memory

Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

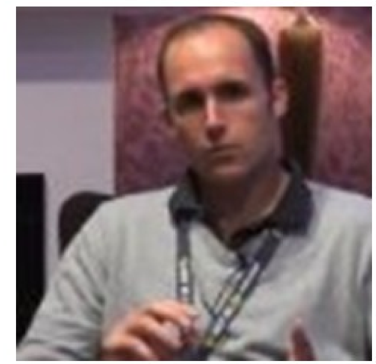
- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Introduction to the speaker

Chris Bailey

Java Support, Monitoring and Serviceability

13 years experience developing and deploying Java SDKs



Recent work focus:

- Java applications in PureApp and Bluemix
- Java monitoring and diagnostic tools and capabilities
- Highly resilient and scalable deployments
- Java usability and quality
- Requirements gathering

My contact information:

- baileyc@uk.ibm.com
- http://www.linkedin.com/in/chrisbaileyibm
- http://www.slideshare.net/cnbailey/

IBM developerWorks Professional Author

Goals of this talk

- Deliver an insight into the memory usage of Java code:
  - The overhead of Java Objects
  - The cost of delegation
  - The overhead of the common Java Collections

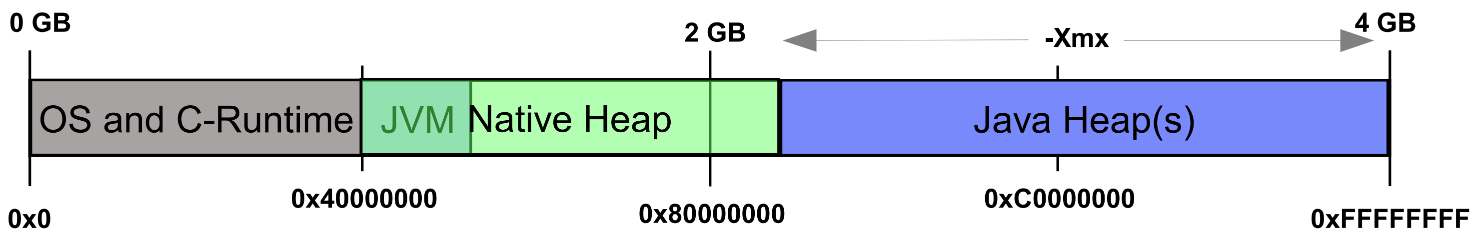
- Provide you with information to:
  - Choose the right collection types
  - Analyze your application for memory inefficiencies

Agenda

- Introduction to Memory Management
- Anatomy of a Java object
- Understanding Java Collections
- Analyzing your application

Understanding Java Memory Management

- Java runs as a Operating System (OS) level process, with the restrictions that the OS imposes:



- 32 bit architecture and/or OS gives 4GB of process address space
  - Much, much larger for 64bit
- Some memory is used by the OS and C-language runtime
  - Area left over is termed the “User Space”
- Some memory is used by the Java Virtual Machine (JVM) runtime
- Some of the rest is used for the Java Heap(s)

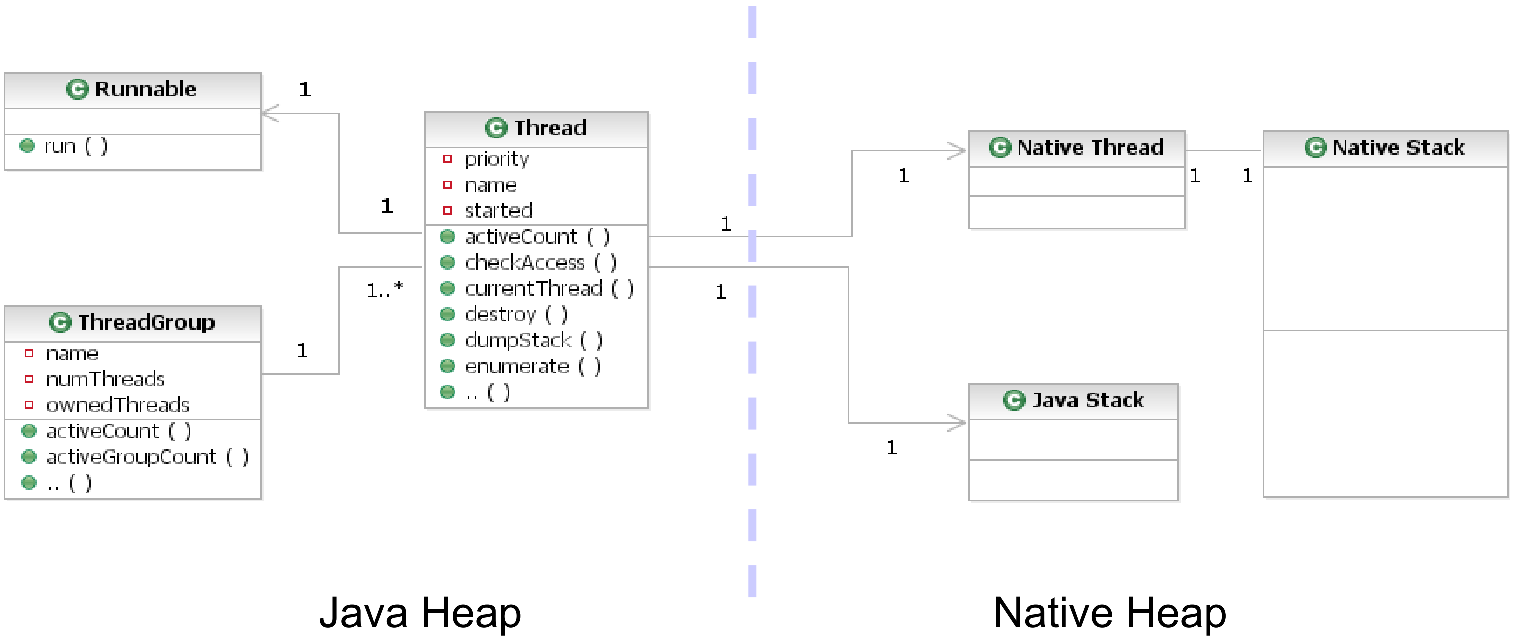
…and some is left over: the “native” heap

- Native heap is usually measured including the JVM memory usage

Java objects with “native” resources

- A number of Java objects are underpinned by OS level resources
  - Therefore have associated “native” heap memory

- Example: `java.lang.Thread`



Anatomy of a Java Object

```
public class CreateInteger {

    public static void main(String[] args) {
        Integer myInteger = new Integer(10);
    }

}
```

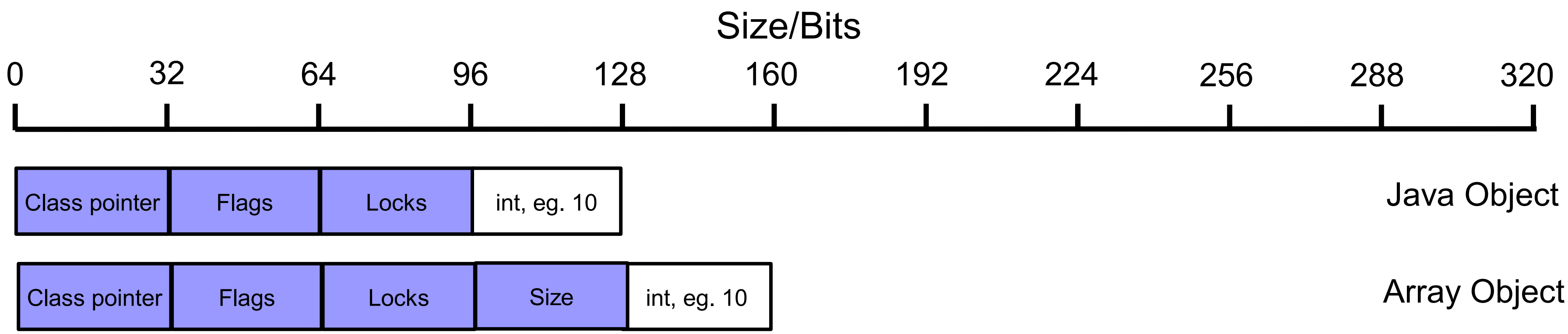
- Question: An *int* (eg. 10) is 32 bits, but how much bigger is an *Integer* object?
  (for a 32bit platform)
  (a) x1
  (b) x1.5
  (c) x2
  (d) x3

- **Answer is option (e) x4 !!**

Anatomy of a Java Object

```
public static void main(String[] args) {
    Integer myInteger = new Integer(10);
}
```

- Object Metadata: 3 slots of data (4 for arrays)
  - Class: pointer to class information
  - Flags: shape, hash code, etc
  - Lock: flatlock or pointer to inflated monitor
  - Size: the length of the array *(arrays only)*

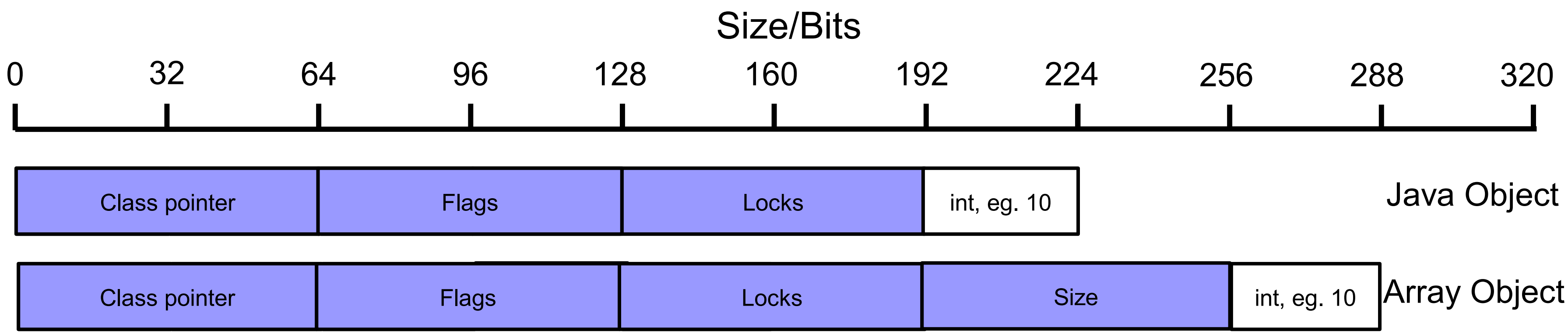


- Additionally, all Objects are 8 byte aligned (16 byte for CompressedOops with large heaps)

Anatomy of a 64bit Java Object

```
public static void main(String[] args) {
    Integer myInteger = new Integer(10);
}
```

- Object Metadata: 3 slots of data (4 for arrays)
  - Class: pointer to class information
  - Flags: shape, hash code, etc
  - Lock: flatlock or pointer to inflated monitor
  - Size: the length of the array *(arrays only)*



- Size ratio of an *Integer* object to an *int* value becomes x9 !!

Object Field Sizes

Field Type	Field size/bits			
	32bit Process		64bit Process	
	Object	Array	Object	Array
boolean	32	8	32	8
byte	32	8	32	8
char	32	16	32	16
short	32	16	32	16
int	32	32	32	32
float	32	32	32	32
long	64	64	64	64
double	64	64	64	64
Objects	32	32	64*	64

***32bits if Compressed References / Compressed Oops enabled**

Compressed References and CompressedOOPs

- Migrating an application from 32bit to 64bit Java increases memory usage:
  - Java heap usage increases by ~70%
  - “Native” heap usage increases by ~90%

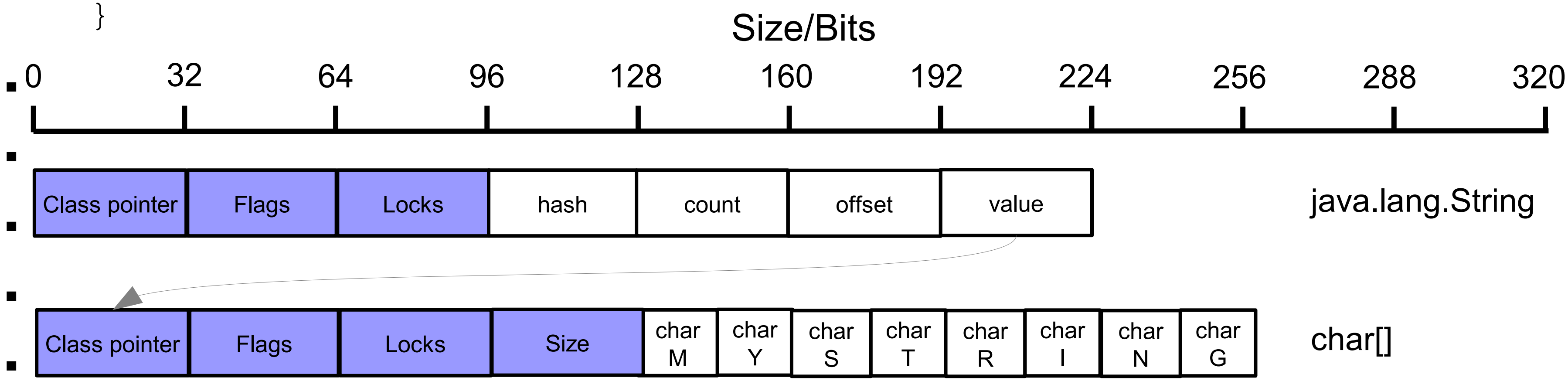
- Compressed References / Compressed Ordinary Object Pointers
  - Use bit shifted, relative addressing for 64bit Java heaps
  - Object metadata and Objects references become 32bits

- Using compressed technologies ***does*** remove **Java heap** usage increase
- Using compressed technologies ***does not*** remove **“native” heap** usage increase

Allocating (slightly) more complex objects

- Good object orientated design encourages encapsulation and delegation
- Simple example: java.lang.String containing “MyString”:

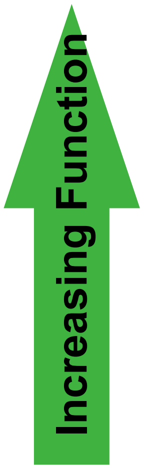
```
public static void main(String[] args) {
    String myString = new String("MyString");
}
```



- 128 bits of char data, stored in 480 bits of memory, size ratio of x3.75
  - Maximum overhead would be x24 for a single character!

Java Collections

- Each Java Collection has a different level of function, and memory overhead



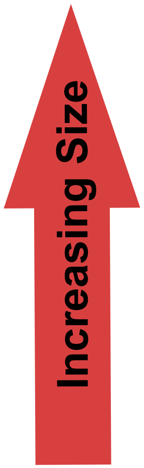
java.util.HashSet

java.util.HashMap

java.util.Hashtable

java.util.LinkedList

java.util.ArrayList



- Using the wrong type of collection can incur significant additional memory overhead

HashSet

```
public static void main(String[] args) {
    HashSet myHashSet = new HashSet();
}
```

- Implementation of the Set interface
  - *"A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction. "*
    - *Java Platform SE 6 API doc*

- Implementation is a wrapper around a HashMap:

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashSet @ 0x10a6d908	16	144
java.util.HashMap @ 0x10a6d918	48	128

- Default capacity for HashSet is 16
- Empty size is 144 bytes
- Additional 16 bytes / 128 bits overhead for wrappering over HashMap

HashMap

```
public static void main(String[] args) {
    HashMap myHashMap = new HashMap();
}
```

- Implementation of the Map interface:
  - *“An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.“*
    - *Java Platform SE 6 API doc*

- Implementation is an array of HashMap$Entry objects:

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.HashMap @ 0x10a6d918	48	128
java.util.HashMap$Entry[16] @ 0x10a6d948	80	80

- Default capacity is 16 entries
- Empty size is 128 bytes
- Overhead is 48 bytes for HashMap, plus (16 + (entries * 4bytes)) for array
  - Plus overhead of HashMap$Entry objects

HashMap$Entry

- Each HashMap$Entry contains:
  - int KeyHash
  - Object next
  - Object key
  - Object value

- Additional 32bytes per key ↔ value entry
- Overhead of HashMap is therefore:
  - 48 bytes, plus 36 bytes per entry

- For a 10,000 entry HashMap, the overhead is ~360K

Hashtable

```
public static void main(String[] args) {
    Hashtable myHashtable = new Hashtable();
}
```

- Implementation of the Map interface:
  - *“This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.“*
    - *Java Platform SE 6 API doc*

- Implementation is an array of Hashtable$Entry objects:

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1bae9290	40	104
java.util.Hashtable$Entry[11] @ 0x1bae92b8	64	64

- Default capacity is 11 entries
- Empty size is 104 bytes
- Overhead is 40 bytes for Hashtable, plus (16 + (entries * 4bytes)) for array
  - Plus overhead of Hashtable$Entry objects

Hashtable$Entry

- Each Hashtable$Entry contains:
  - int KeyHash
  - Object next
  - Object key
  - Object value

- Additional 32bytes per key ↔ value entry

- Overhead of Hashtable is therefore:
  - 40 bytes, plus 36 bytes per entry

- For a 10,000 entry Hashtable, the overhead is ~360K

LinkedList

```java
public static void main(String[] args) {
    LinkedList myLinkedList = new LinkedList();
}
```

- Linked list implementation of the List interface:
  - *"An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.*
  - *Unlike sets, lists typically allow duplicate elements. "*
    - *Java Platform SE 6 API doc*

- Impleme

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.LinkedList @ 0x11624d50 Thread	24	48
java.util.LinkedList$Link @ 0x11624d68	24	24

nk):

- Default capacity is 1 entry
- Empty size is 48 bytes
- Overhead is 24 bytes for LinkedList, plus overhead of LinkedList$Entry/Link objects

LinkedList$Entry / Link

- Each LinkedList$Entry contains:
  - Object previous
  - Object next
  - Object entry

- Additional 24bytes per entry

- Overhead of LinkedList is therefore:
  - 24 bytes, plus 24 bytes per entry

- For a 10,000 entry LinkedList, the overhead is ~240K

ArrayList

```
public static void main(String[] args) {
    ArrayList myArrayList = new ArrayList();
}
```

- A resizeable array instance of the List interface:
  - *"An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.*
  - *Unlike sets, lists typically allow duplicate elements. "*
    - *Java Platform SE 6 API doc*

- Implem

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.ArrayList @ 0x1fc279e0	32	88
java.lang.Object[10] @ 0x1fc27a00	56	56

- Default capacity is 10 entries
- Empty size is 88 bytes
- Overhead is 32bytes for ArrayList, plus (16 + (entries * 4bytes)) for array
- For a 10,000 entry ArrayList, the overhead is ~40K

Other types of “Collections”

```java
public static void main(String[] args) {
    StringBuffer myStringBuffer = new StringBuffer();
}
```

- StringBuffers can be considered to be a type of collection
  - *“A thread-safe, mutable sequence of characters...*

    ....
  - *Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.”*
    - *Java Platform SE 6 API doc*

- Implementation is an array of char

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.StringBuffer @ 0x2898eb0 buffer text	24	72
char[16] @ 0x2898ec8 buffer text\u0000\u0000\u0000\u0000\u0000	48	48

- Default capacity is 16 characters
- Empty size is 72 bytes
- Overhead is just 24bytes for StringBuffer

Collections Summary

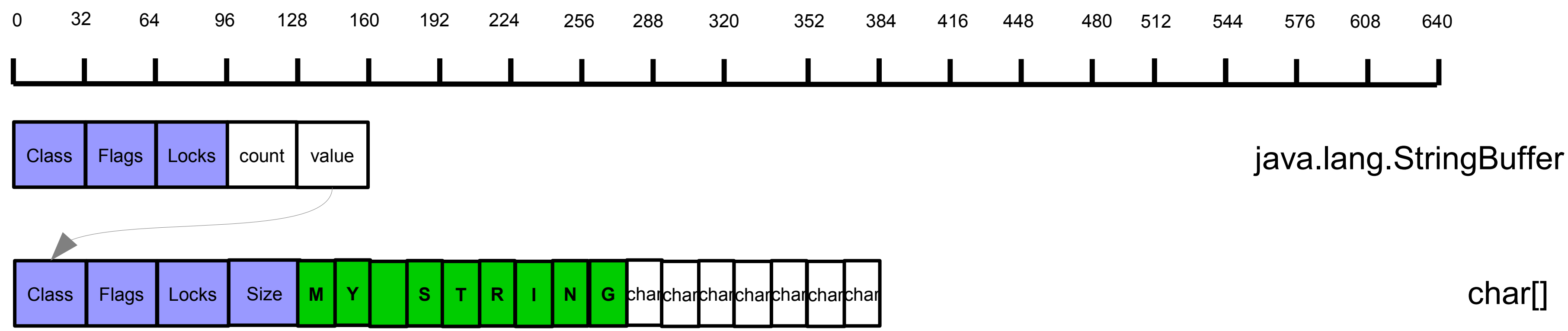
Collection	Default Capacity	Empty Size	10K Overhead
HashSet	16	144	360K
HashMap	16	128	360K
Hashtable	11	104	360K
LinkedList	1	48	240K
ArrayList	10	88	40K
StringBuffer	16	72	24

Hash* collections vs others

- Hash* collections are much larger
  - x9 the size of an ArrayList
- Additional size helps search/insert/delete performance
  - Constant for Hash collections
  - Linear for Array collections
    - If there is no other index
- Using the larger collection *may* be the right thing to do
  - Important to ***know*** it is the right thing to do!

Empty space in collections

- Collections that contain empty space introduce additional overhead
- Default collection size may not be appropriate for the amount of data being held
- ```
  public static void main(String[] args) {
      StringBuffer myStringBuffer = new StringBuffer("MyString");
  }
  ```

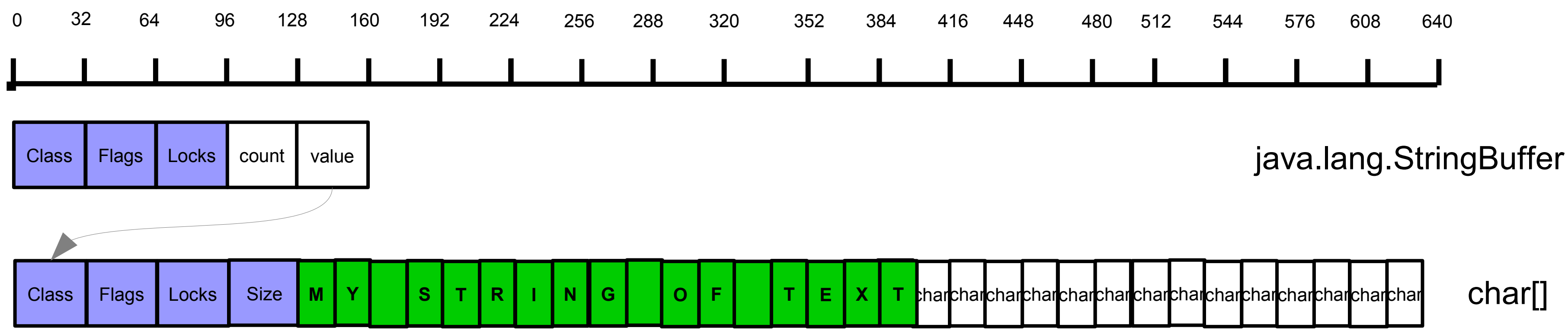


- StringBuffer default of 16 is inappropriate to hold a 9 character string
  - 7 additional entries in char[]
  - 112 byte additional overhead
  ```

Expansion of collections

- When collections hit the limit of their capacity, they expand
  - Greatly increases capacity
  - Greatly reduces "fill ratio"

- Introduces additional collection overhead:



- Additional 16 char[] entries to hold single extra character
  - 240 byte additional overhead

Collections Summary

Collection	Default Capacity	Empty Size	10K Overhead	Expansion
HashSet	16	144	360K	x2
HashMap	16	128	360K	x2
Hashtable	11	104	360K	x2 + 1
LinkedList	1	48	240K	+1
ArrayList	10	88	40K	x1.5
StringBuffer	16	72	24	x2

Collections Summary

- Collections exist in large numbers in many Java applications

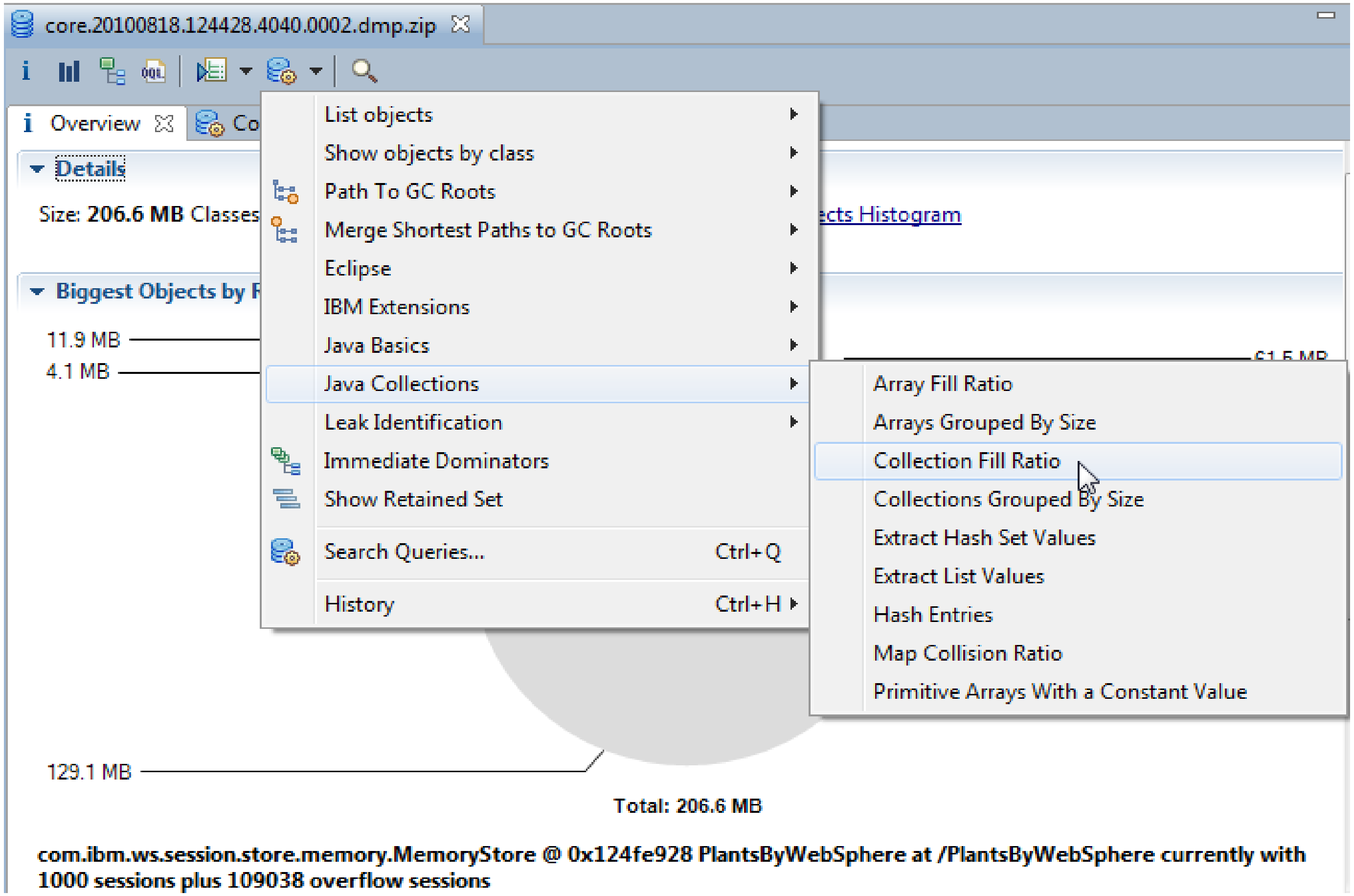
- Example: IBM WebSphere Application Server running PlantsByWebSphere
  - When running a 5 user test load, and using 206MB of Java heap:

HashTable	262,234	instances,	26.5MB	of Java heap
WeakHashMap	19,562	instances	12.6MB	of Java heap
HashMap	10,600	instances	2.3MB	of Java heap
ArrayList	9,530	instances	0.3MB	of Java heap
HashSet	1,551	instances	1.0MB	of Java heap
Vector	1,271	instances	0.04MB	of Java heap
LinkedList	1,148	instances	0.1MB	of Java heap
TreeMap	299	instances	0.03MB	of Java heap
	306,195		**42.9MB**	

- 16% of the Java heap used just for the collection objects !!

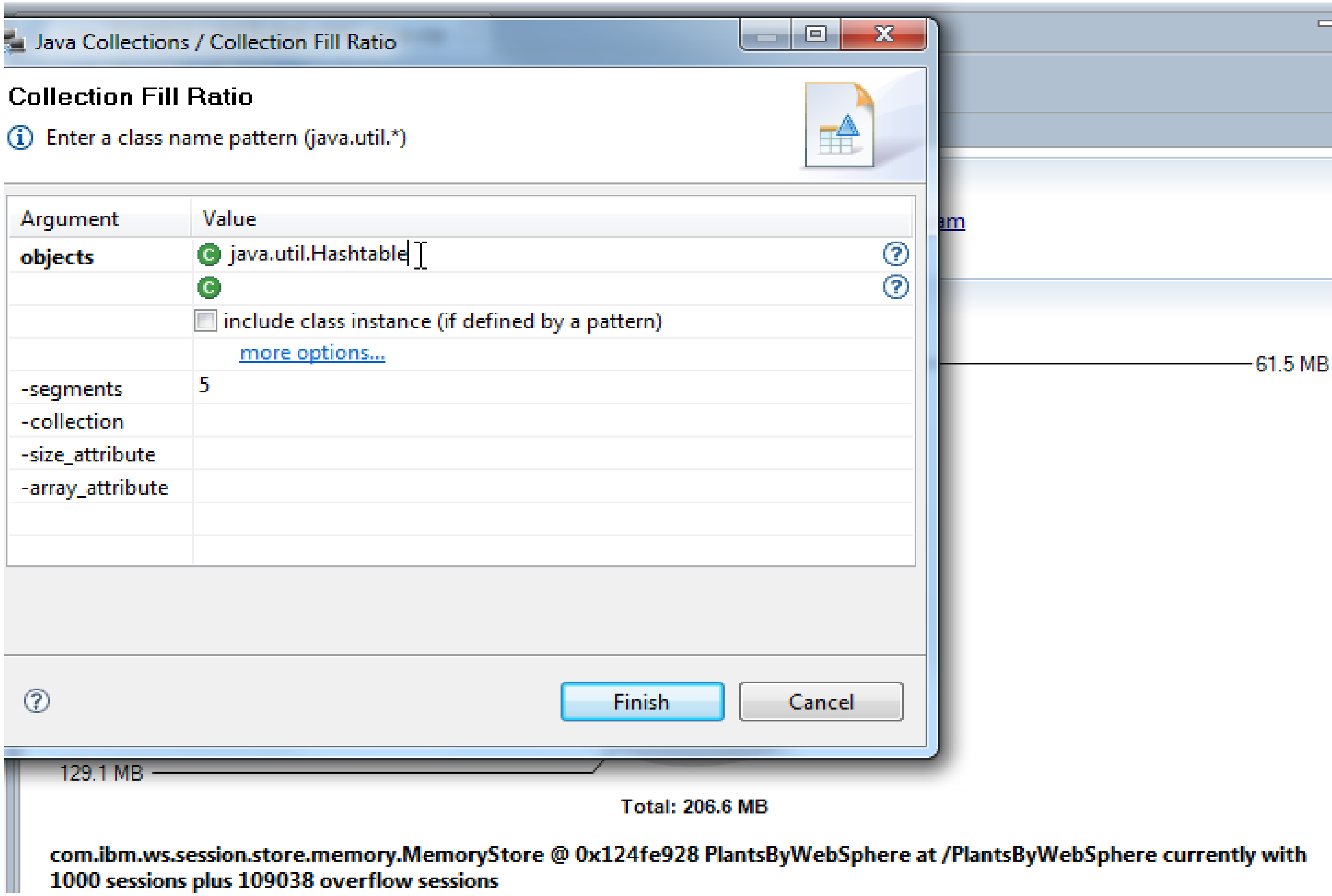
Analyzing your Collections

- Eclipse Memory Analyzer Tool (MAT) provides Collection analysis:



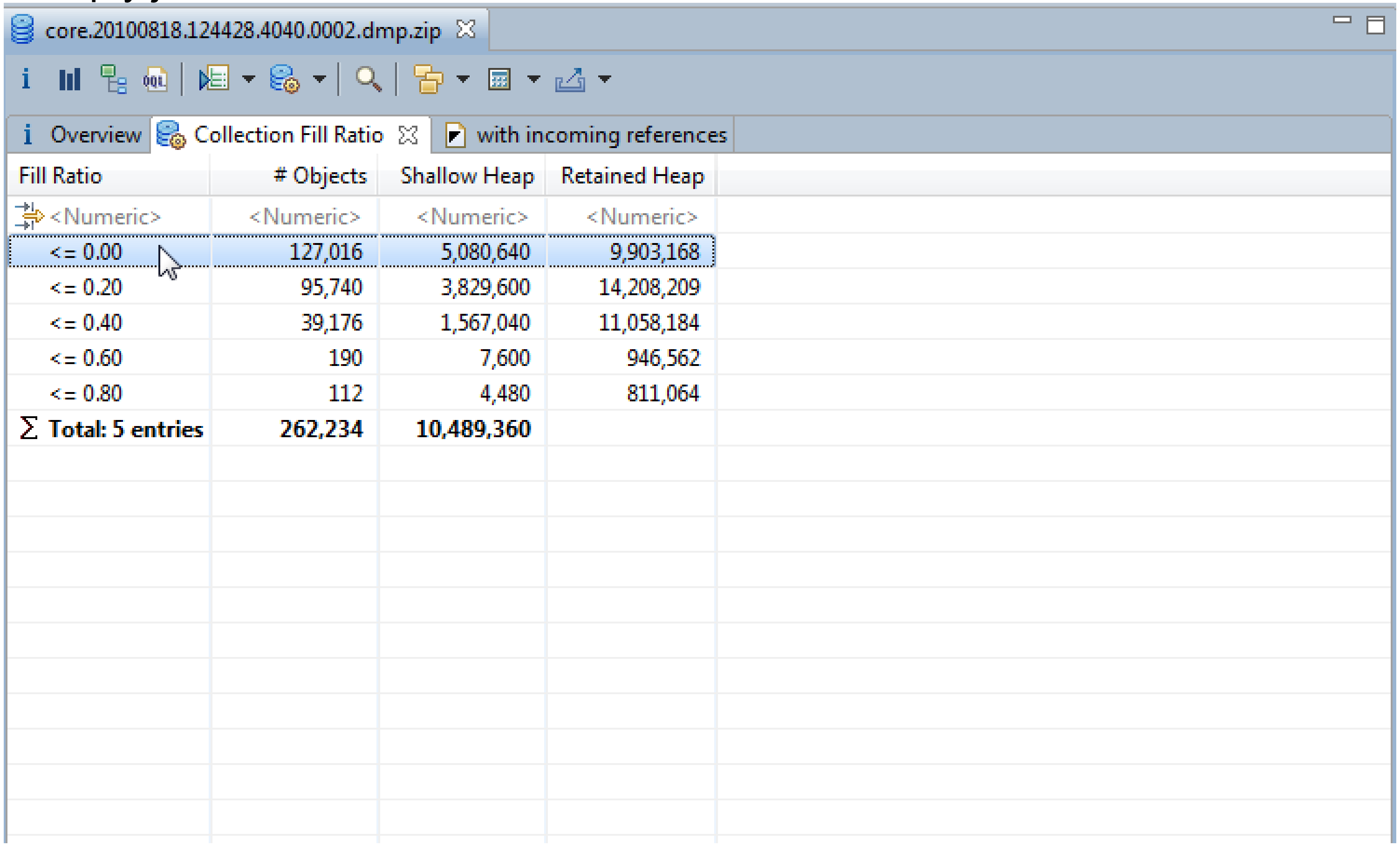
Analyzing your Collections

- Can select a specific Collection (java.util.Hashtable) or any



Analyzing your Collections

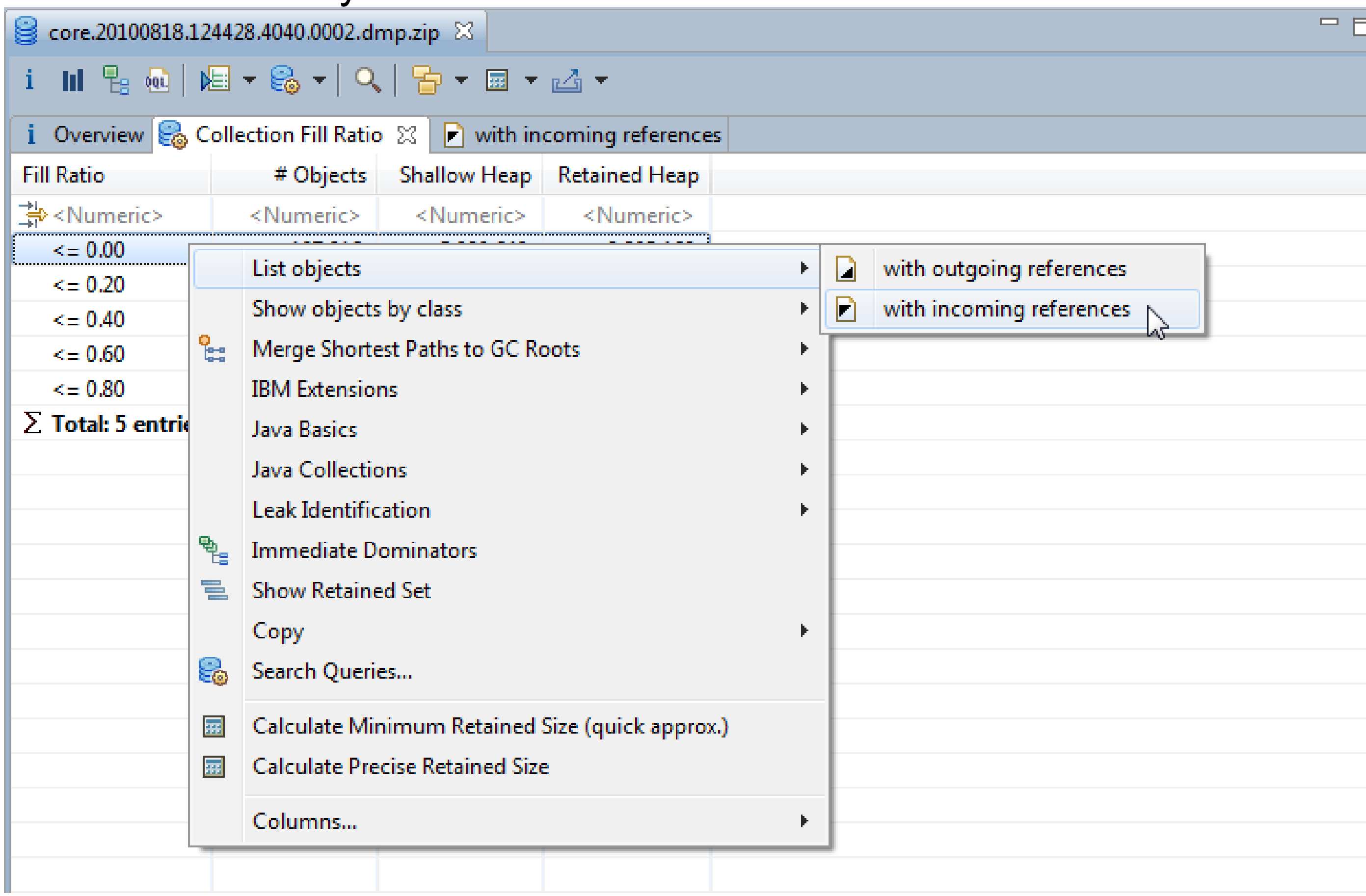
- Shows 127,016 empty java.util.Hashtable instances!



Fill Ratio	# Objects	Shallow Heap	Retained Heap
<Numeric>	<Numeric>	<Numeric>	<Numeric>
<= 0.00	127,016	5,080,640	9,903,168
<= 0.20	95,740	3,829,600	14,208,209
<= 0.40	39,176	1,567,040	11,058,184
<= 0.60	190	7,600	946,562
<= 0.80	112	4,480	811,064
Σ Total: 5 entries	**262,234**	**10,489,360**	

Analyzing your Collections

- You can “List objects” to see what they are



Analyzing your Collections

- java.util.Hashtable objects being used to store session data!

core.20100818.124428.4040.0002.dmp.zip

Overview | Collection Fill Ratio | with incoming references

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Hashtable @ 0x1ff06bf8	40	80
_attributes com.ibm.ws.session.store.memory.MemorySession @ 0x1ff06b38 for memo	112	352
value java.util.HashMap$Entry @ 0x1ff06c48	32	1,080
_iSession com.ibm.ws.session.WsSessionData @ 0x1ff06c68	64	80
Σ **Total: 2 entries**		
java.util.Hashtable @ 0x1ff06ba8	40	80
_attributeNames com.ibm.ws.session.store.memory.MemorySession @ 0x1ff06b38 for	112	352
value java.util.HashMap$Entry @ 0x1ff06c48	32	1,080
_iSession com.ibm.ws.session.WsSessionData @ 0x1ff06c68	64	80
Σ **Total: 2 entries**		
java.util.Hashtable @ 0x1fd66d48	40	80
_attributes com.ibm.ws.session.store.memory.MemorySession @ 0x1fd66c88 for memo	112	352
value java.util.HashMap$Entry @ 0x1fd66d98	32	480
_iSession com.ibm.ws.session.WsSessionData @ 0x1fd66db8	64	80
Σ **Total: 2 entries**		
java.util.Hashtable @ 0x1fd66cf8	40	80
_attributeNames com.ibm.ws.session.store.memory.MemorySession @ 0x1fd66c88 for	112	352
value java.util.HashMap$Entry @ 0x1fd66d98	32	480
_iSession com.ibm.ws.session.WsSessionData @ 0x1fd66db8	64	80
Σ **Total: 2 entries**		
java.util.Hashtable @ 0x1fd4de78	40	80
java.util.Hashtable @ 0x1fd4de28	40	80

Collection Analysis for PlantsByWebSphere Example

Collection	Number	Empty	% Empty
Hashtable	262,234	127,016	48.8
WeakHashMap	19,562	19,456	99.5
HashMap	10,600	7,599	71.7
ArrayList	9,530	4,588	48.1
HashSet	1,551	866	55.8
Vector	1,271	622	48.9
Total	304,748	160,156	52.6

- Over 50% of collections are empty in our example

Improvements in the JDK: WeakHashMap

WeakHashMap	19,562	19,456	99.5

- 12.5MB of memory being used for 19,456 empty instances of WeakHashMap

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.WeakHashMap @ 0x1fcf6d30	48	688
<class> class java.util.WeakHashMap @ 0x1007f748 System Class	8,863	8,863
elementData java.util.WeakHashMap$Entry[16] @ 0x1fcf6d60	80	80
referenceQueue java.lang.ref.ReferenceQueue @ 0x1fcf6db0	32	560
Σ **Total: 3 entries**		

- 560 bytes per instance used for java.lang.ref.ReferenceQueue
  - ReferenceQueue only required is there are elements in the WeakHashMap

- Lazy allocation of ReferenceQueue saves 10.9MB in our example

Techniques for minimizing memory

- Lazy allocation of collections
  - Don't create a collection until you have something to put into it
- Don't create collections for a single Object!
  - Just store the Object itself
- Correct sizing of collections
  - If only 2 entries will be stored, create with size 2:

```
HashMap myHashMap = new HashMap(2);
```

- Avoid expansion of large collections due to x2 algorithm
  - 32MB used to store 17MB of data
- Collections do not shrink once expanded
  - May need to reallocate if collection uses drops significantly

Summary

- There is significant overhead to your data!
  - Some of which is on the "native" heap

- Applications often have:
  - The wrong collection types in use
  - Empty or sparsely populated collections

- Careful use of:
  - Data structure layout
  - Collection type selection
  - Collection type default sizing

  Can improve your memory efficiency

- Eclipse Memory Analyzer Tool can identify inefficiencies in your application
  - As well as show you the wider memory usage for code

Questions?

Upcoming Java Expert Call Series

Performance: Garbage Collection Tuning & Heap Sizing	This presentation provides a brief description of the garbage collection policies available in the IBM JVM, along with a guide for configuring the new default policy. The session also provide an understanding of the IBM JDK Generational Collector mechanics, how to tune the collector for best behavior, and how to distinguish between generational and non-generational workloads
Overview of IBM Java Runtime Tools (GCMV, Memory Analyzer & Health Center)	This session will give an overview of the IBM Monitoring and Diagnostic Tools for Java, and hints & tips on how the tools can help speed up application development and deployment
Java Application Performance - Tools for identifying performance bottlenecks	The presentation will cover debugging Java performance problems including resource contention, application code performance & external delays. Java tracing options will be covered.
Debugging Java OOM issues using Eclipse Memory Analyzer	This session will show developers, architects and operations engineers how to use Memory Analyzer Tool to debug Java Heap Out Of Memory Errors and explore the application to find the source of problem
Where does all the native memory go? Best practices for debugging native memory problems.	This session would cover how Java runtime uses native memory, what happens when an application runs out of memory and best practices for debugging native memory problems

Backup

Process Memory Monitoring

- Monitoring of native heap carried out by monitoring the process size
  - Java heap and VM usage are static, so process size growth is the native heap

- Exhaustion of the process address space shows native heap exhaustion
  - Leads to OutOfMemoryError as would Java heap exhaustion

- Native heap is managed using OS malloc/free routines
- Therefore OS tools are best place to monitor memory usage

- GC and Memory Visualiser (GCMV) in ISA can visualize some OS tool output:
  - AIX, Linux, Windows, z/OS

- Health Center can do live monitoring of native memory usage:
  - AIX, Linux, Windows, z/OS

Process Memory Monitoring: Windows

- Recommended tool is “perfmon”
  - In Control Panel -> Admin Tools -> Performance
  - Can also be started using “perfmon” on the command line

- Displays a number of counters for a given process
- Relevant counter is “Virtual Bytes”
  - memory that has been allocated, ie. a malloc() request has been made
- “Working Set” may also be on interest
  - memory that is committed to, ie. has been written to and is actively in use
- NB: Maximum Java heap size is allocated at start up
  - but only the Minimum heap size is written to (committed)

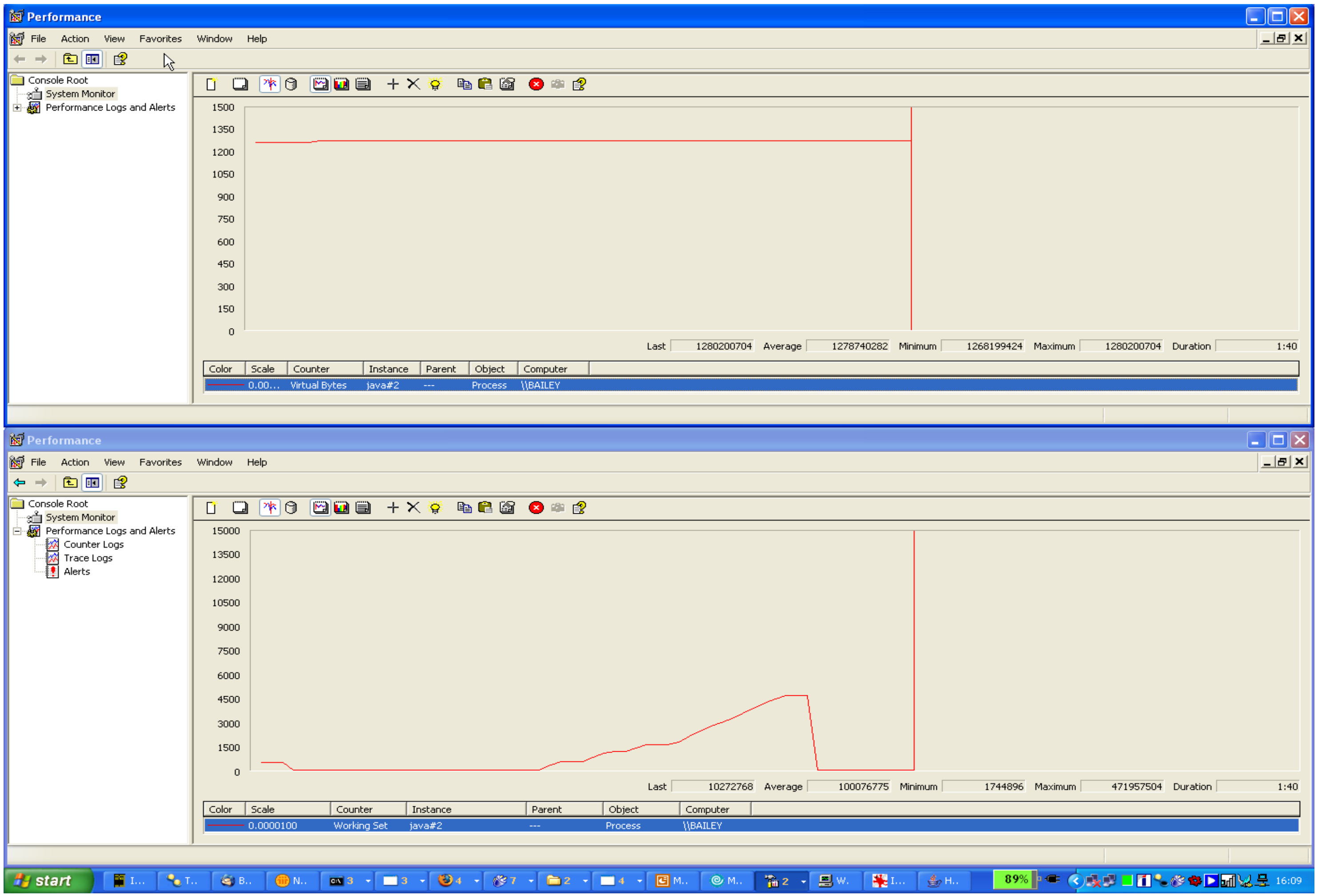
Perfmon log

- Perfmon can log to text (.csv) file or binary
- CSV file format is as follows:

```
"(PDH-CSV 4.0) (GMT Daylight Time)(-60)","\\MY_COMP\Process(java)\Virtual Bytes"
"05/08/2008 16:33:56.859","1198592000"
"05/08/2008 16:34:11.859","1198592000"
"05/08/2008 16:34:26.859","1198592000"
"05/08/2008 16:34:41.859","1198592000"
"05/08/2008 16:34:56.859","1198592000"
```

- Can be imported into other tooling:
  - **GCMV!**
  - Spreadsheet
  - Database
  - etc

Process Memory: PerfMon View



Process Memory Monitoring: AIX

- Recommended tool is “svmon”
  - Available on the AIX install image
  - Started using “svmon –P {pid} –m –r –i {interval}”

- Displays a per segment breakdown of memory
  - Relevant value is “Addr Range” for heap segments
  - memory that has been allocated, ie. a malloc() request has been made
  - Relevant heap segments according to AIX memory layout

Process Memory: Svmon Output

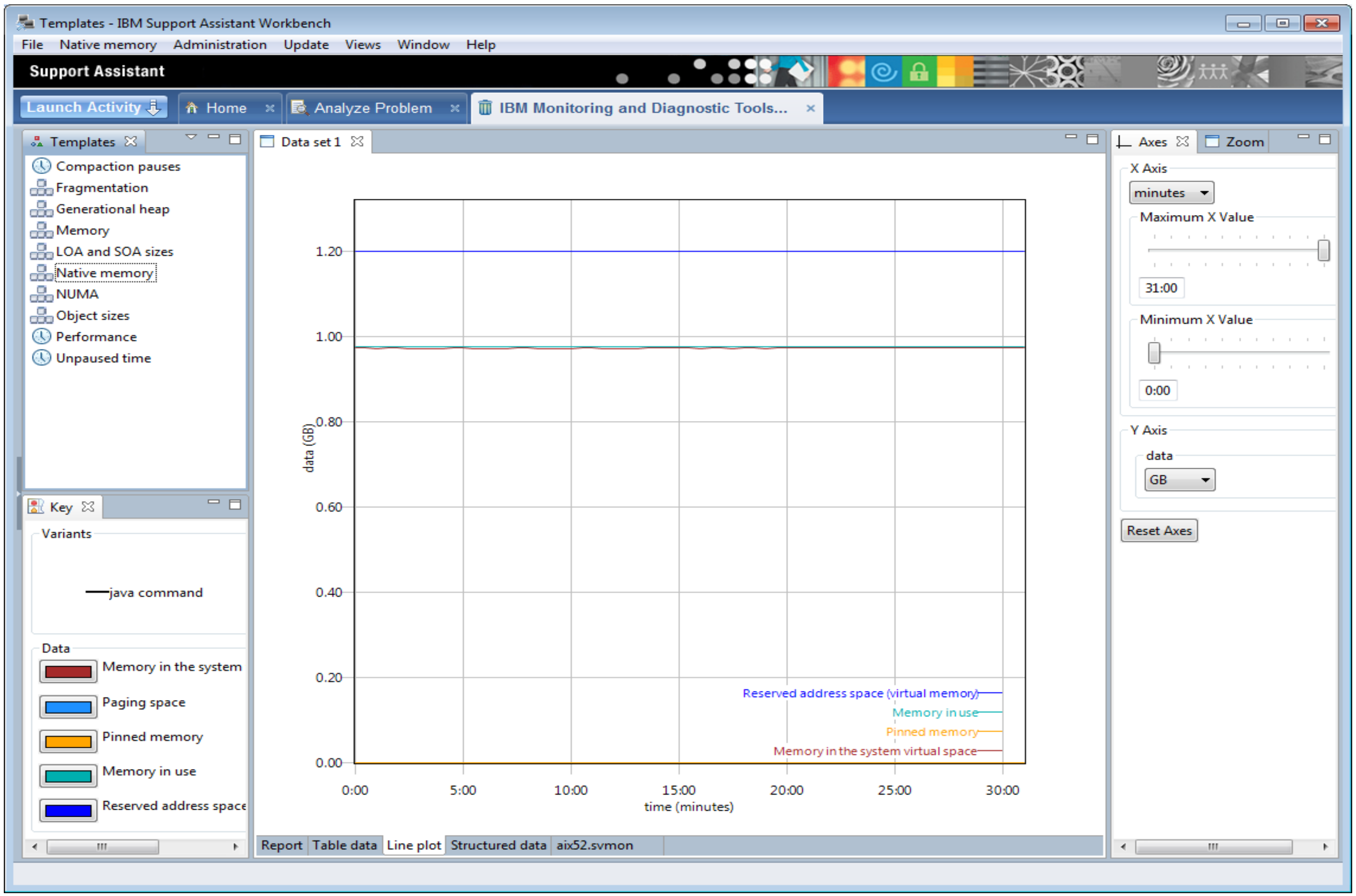
```
Pid Command          Inuse      Pin      Pgsp  Virtual   64-bit    Mthrd
   25084 AppS            78907      1570       182     67840         N        Y
    Vsid      Esid Type Description           Inuse    Pin Pgsp Virtual Addr Range
   2c7ea         3 work shmat/mmap            36678      0    0 36656   0..65513
   3c80e         4 work shmat/mmap             7956      0    0  7956   0..65515
   5cd36         5 work shmat/mmap             7946      0    0  7946   0..65517
   14e04         6 work shmat/mmap             7151      0    0  7151   0..65519
   7001c         d work shared library text    6781      0    0   736   0..65535
       0         0 work kernel seg             4218   1552  182  3602   0..22017 :
                                                                        65474..65535
   6cb5a         7 work shmat/mmap             2157      0    0  2157   0..65461
   48733         c work shmat/mmap             1244      0    0  1244   0..1243
    cac3         - pers /dev/hd2:176297        1159      0    -     -   0..1158
   54bb5         - pers /dev/hd2:176307         473      0    -     -   0..472
   78b9e         - pers /dev/hd2:176301         454      0    -     -   0..453
   58bb6         - pers /dev/hd2:176308         254      0    -     -   0..253
    cee2         - work                         246     17    0   246   0..49746
   4cbb3         - pers /dev/hd2:176305         226      0    -     -   0..225
   7881e         - pers /dev/e2axa702-1:2048    186      0    -     -   0..1856
   68f5b         - pers /dev/e2axa702-1:2048    185      0    -     -   0..1847
   28b8a         - pers /dev/hd2:176299         119      0    -     -   0..118
```

Analysing Native Memory with GCMV

- Garbage Collection and Memory Visualizer (GCMV) is available as part of ISA
- GCMV provides scripts to capture the data in the help file

- Visualization makes it easier to see trends over time
  - Look for memory leak
  - Look for native heap footprint issues

Analysing Process Memory in GCMV



Monitoring GC activity

- Monitor GC live using Health Center from ISA
  - Very low (<1%) cost live monitoring of a single Java instance

- Use of Verbose GC logging
  - Activated using command line options:

```
-verbose:gc
-Xverbosegclog:[DIR_PATH][FILE_NAME]
-Xverbosegclog:[DIR_PATH][FILE_NAME],X,Y
```

  - where:

`[DIR_PATH]`	is the directory where the file should be written
`[FILE_NAME]`	is the name of the file to write the logging to
`X`	is the number of files to
`Y`	is the number of GC cycles a file should contain

  - Performance Cost:
    - Very, very small – cost of I/O to stderr/file only
    - basic testing shows a <1ms overhead per GC cycle

Analysing Verbose GC output

- A number of tools exist for plotting verbose:gc output

- **Recommendation**: Garbage Collection and Memory Visualizer (GCMV)
  - Developed, maintained and supported by IBM Java Tools team
  - https://www.ibm.com/developerworks/java/jdk/tools/gcmv/

- Available with in ISA:
  - http://www-306.ibm.com/software/support/isa/

- GCMV will visualise verbose:gc data from the following JVMs
  - IBM, Oracle* and HP
  - 1.4.2, 5.0, 6.0, 7.0
  - WebSphere RealTime 1.0, 2.0 and 3.0

  - *No support for Oracle G1GC yet.

Sizing Deployments

- Check native heap usage before increasing Java heap size
  - Especially for large Java heaps on 32bit

- Ensure enough physical memory is available for all running processes
  - Use “Reserved Address Space (Virtual Memory)” value in GCMV

- Ensure additional physical memory is available for filesystem/IO caching
  - Typically a minimum of 10% of RAM is assigned to file caching

References

- **Get Products and Technologies:**
  - IBM Monitoring and Diagnostic Tools for Java:
    - https://www.ibm.com/developerworks/java/jdk/tools/
  - Eclipse Memory Analyzer Tool:
    - http://eclipse.org/mat/downloads.php
    -
- **Learn:**
  - Debugging from Dumps:
    - http://www.ibm.com/developerworks/java/library/j-memoryanalyzer/index.html
  - Why the Memory Analyzer (with IBM Extensions) isn't just for memory leaks anymore:
    - http://www.ibm.com/developerworks/websphere/techjournal/1103_supauth/1103_supauth.html
- **Discuss:**
  - IBM on Troubleshooting Java Applications Blog:
    - https://www.ibm.com/developerworks/mydeveloperworks/blogs/troubleshootingjava/
  - IBM Java Runtimes and SDKs Forum:
    - http://www.ibm.com/developerworks/forums/forum.jspa?forumID=367&start=0

Copyright and Trademarks