Chris Bailey – IBM Java Service Architect
3rd October 2012

# From Java Code to Java Heap

## Understanding the Memory Usage of Your Application

# 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

- 12 years experience developing and deploying Java SDKs
-
-
- Recent work focus:
  - Java applications in the cloud
  - Java usability and quality
  - Debugging tools and capabilities
  - Requirements gathering
  - Highly resilient and scalable deployments
-
-
- My contact information:
  - baileyc@uk.ibm.com
  - http://www.linkedin.com/in/chrisbaileyibm
  - http://www.slideshare.net/cnbailey/

# Goals of this talk

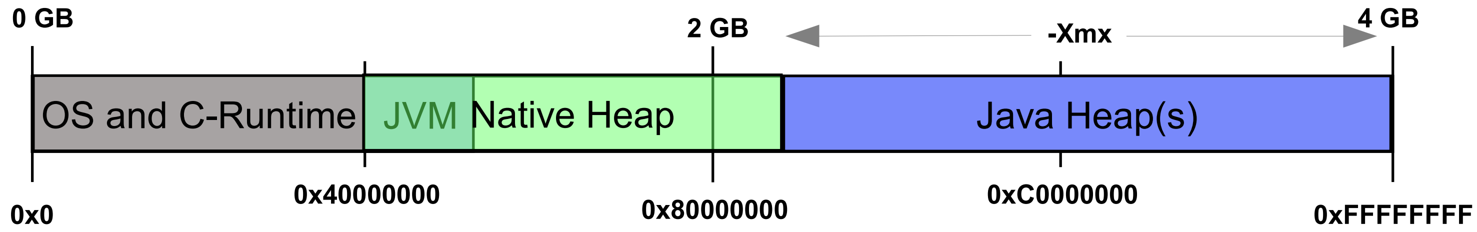
- Deliver an insight into the memory usage of Java code:
  - The overhead of Java Objects
  - The cost of delegation
  - The overhead of the common Java Collections
  -
  -
- Provide you with information to:
  - Choose the right collection types
  - Analyze your application for memory inefficiencies

# 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:



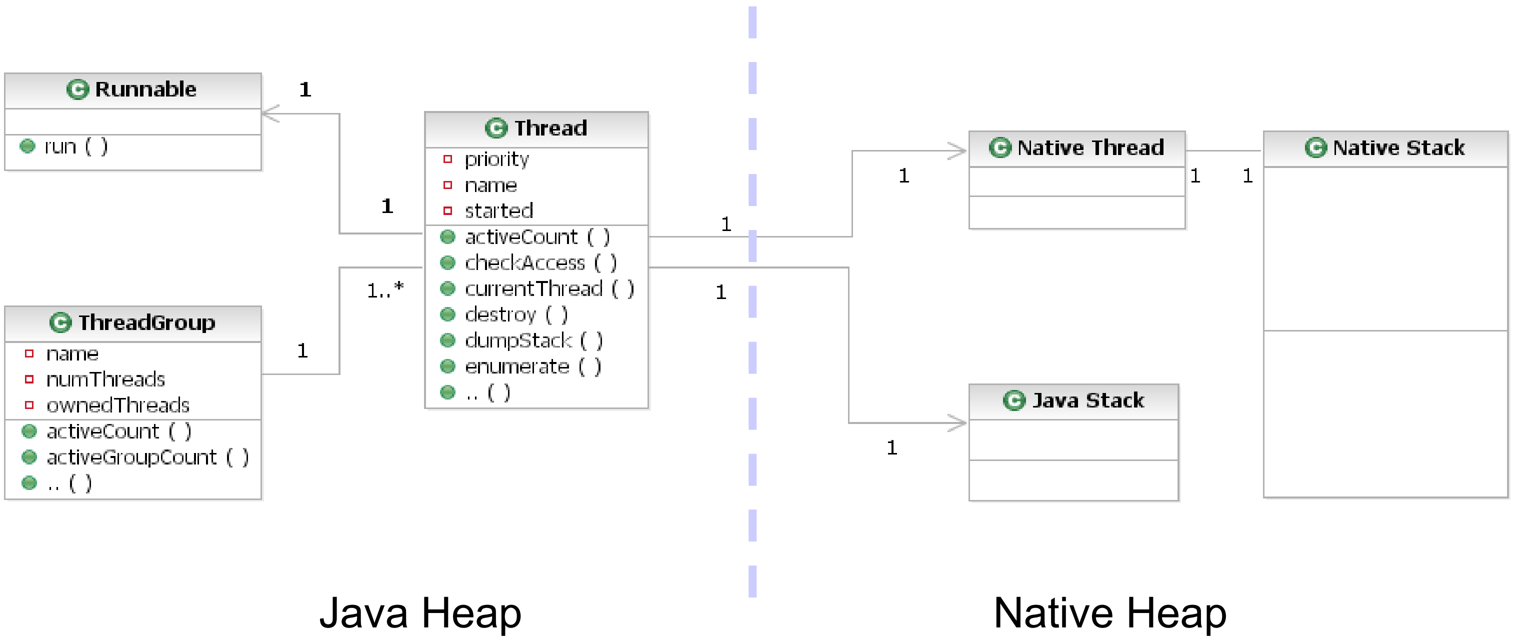

- 32 bit architecture and/or OS gives 4GB of process address space
  - Much, much larger for 64bit
- Some memory is used by the OS and C-language runtime
  - Area left over is termed the “User Space”
- Some memory is used by the Java Virtual Machine (JVM) runtime
- Some of the rest is used for the Java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

}
```

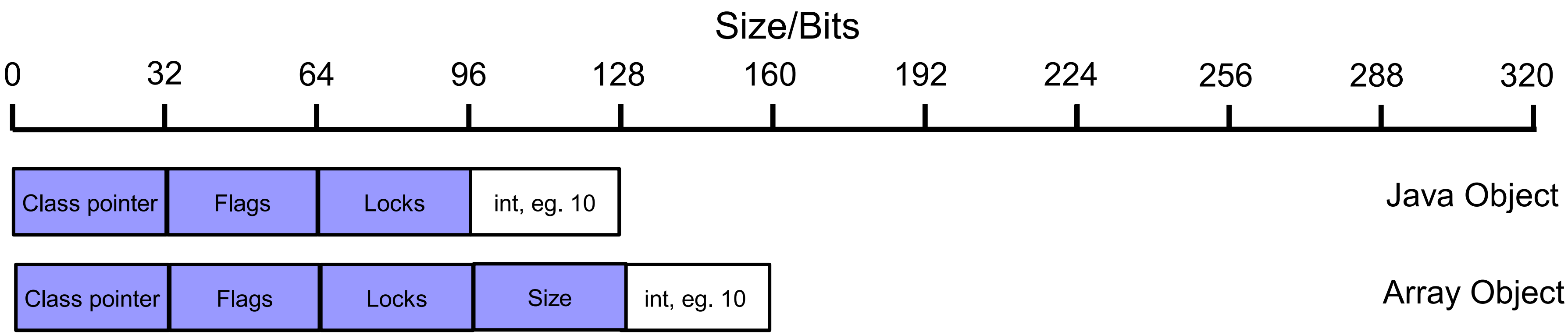
- Question: An *int* (eg. 10) is 32 bits, but how much bigger is an *Integer* object? (for a 32bit platform)
  (a) x1
  (b) x1.5
  (c) x2
  (d) x3

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

# Anatomy of a Java Object

```java
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)*

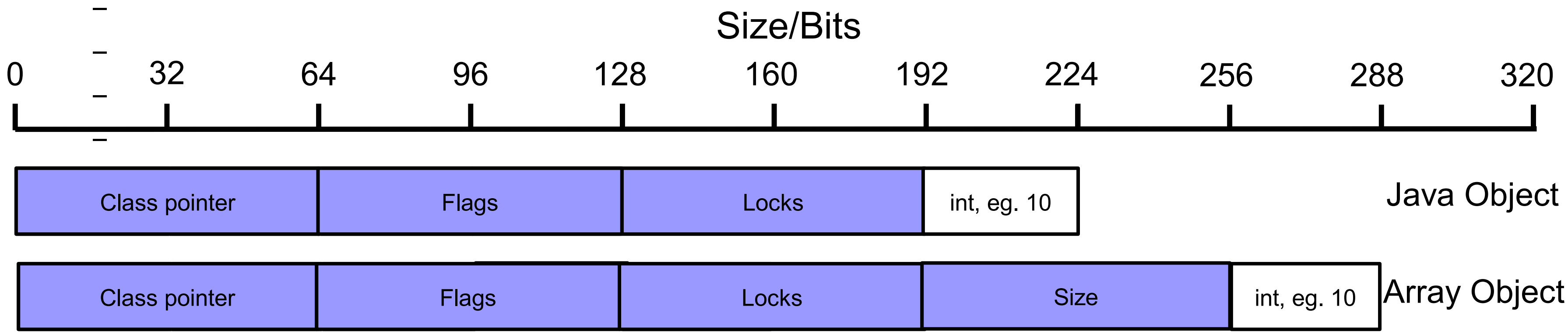



- Additionally, all Objects are 8 byte aligned (16 byte for CompressedOops with large heaps)

# 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**

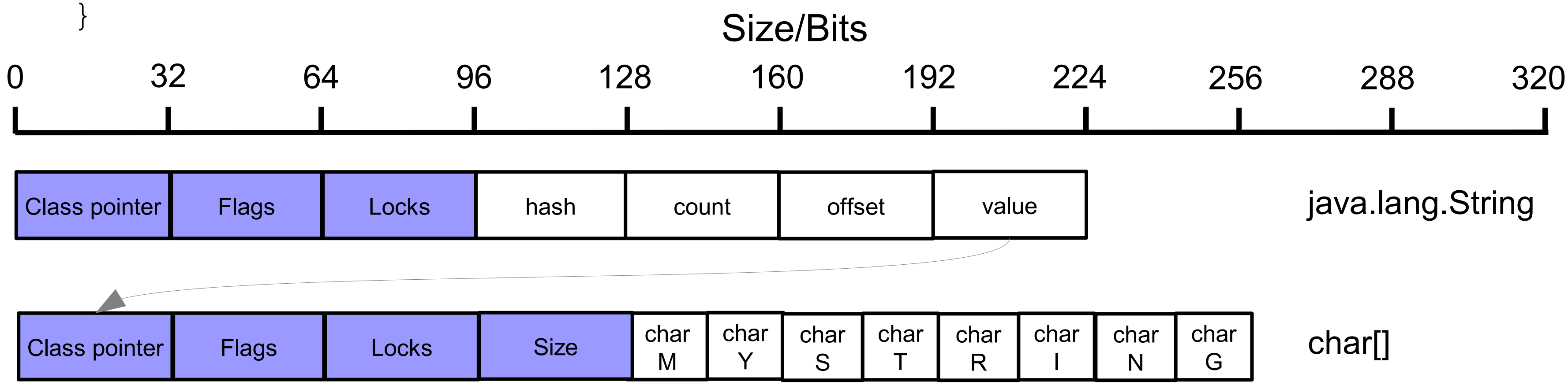
# Compressed References and CompressedOOPs

- 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”:

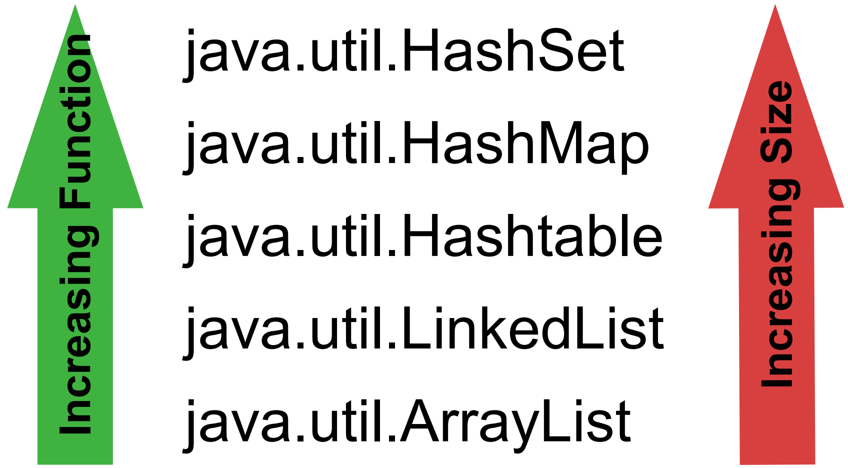
```java
public static void main(String[] args) {
     String myString = new String("MyString");
}
```




- 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




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

# HashSet

```java
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

```java
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

```
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 nk):

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

- 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

```java
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"

```
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

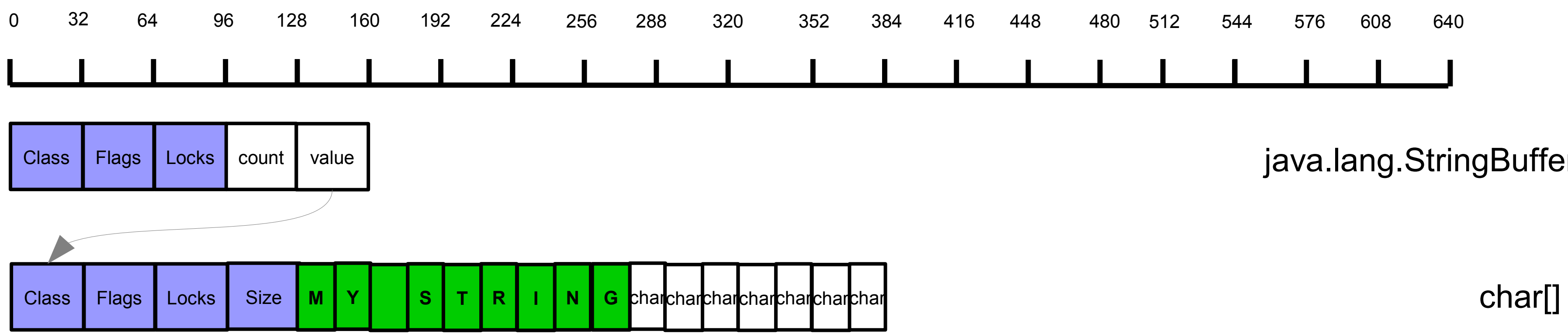
| Collection | Default Capacity | Empty Size | 10K Overhead |
|---|---|---|---|
| HashSet | 16 | 144 | 360K |
| HashMap | 16 | 128 | 360K |
| Hashtable | 11 | 104 | 360K |
| LinkedList | 1 | 48 | 240K |
| ArrayList | 10 | 88 | 40K |
| StringBuffer | 16 | 72 | 24 |

# 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

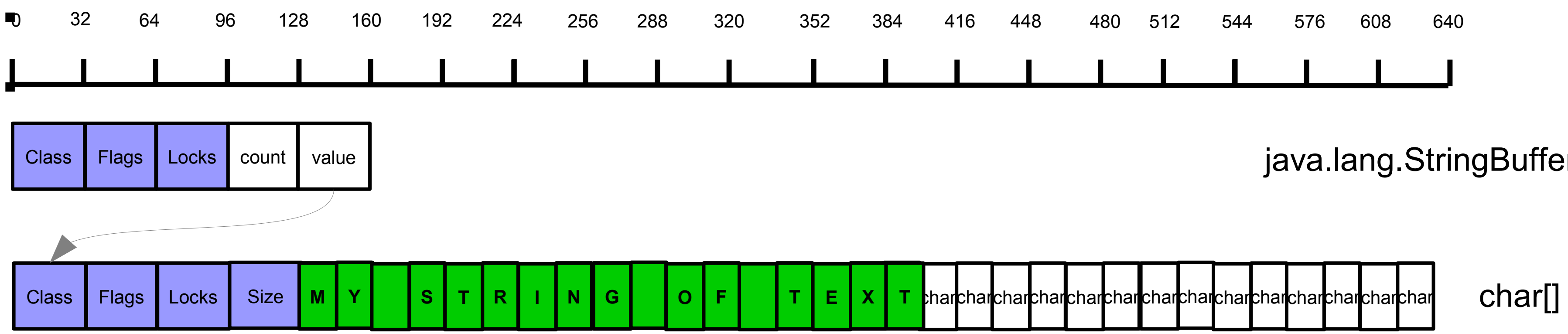
- Collections that contain empty space introduce additional overhead
- Default collection size may not be appropriate for the amount of data being held
- 
```
public static void main(String[] args) {
    StringBuffer myStringBuffer = new StringBuffer("MyString");
}
```




- 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

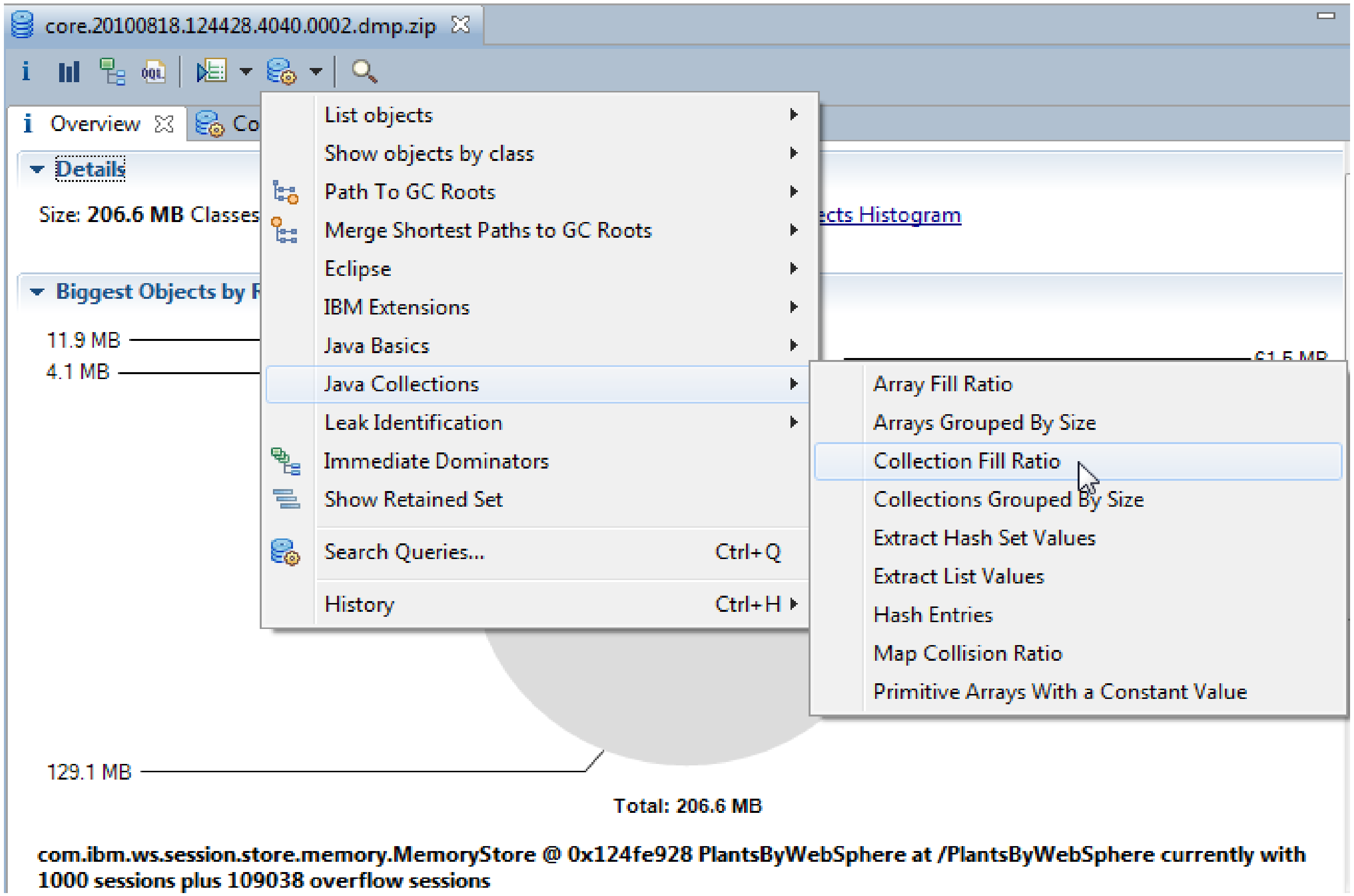
- Collections exist in large numbers in many Java applications
-
- Example: IBM WebSphere Application Server running PlantsByWebSphere
  - When running a 5 user test load, and using 206MB of Java heap:

| | | | | |
|---|---|---|---|---|
| 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 !!

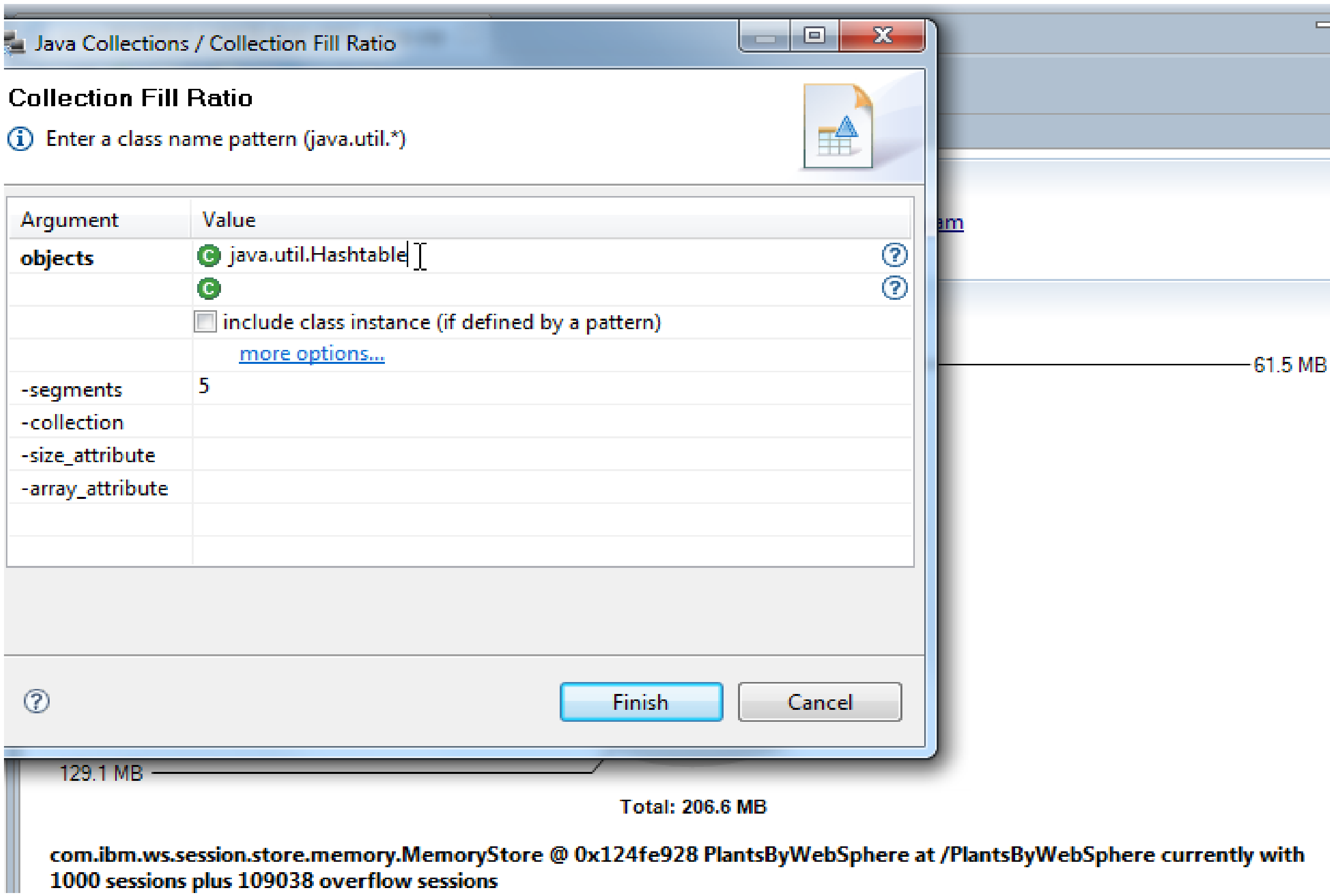
# Analyzing your Collections

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



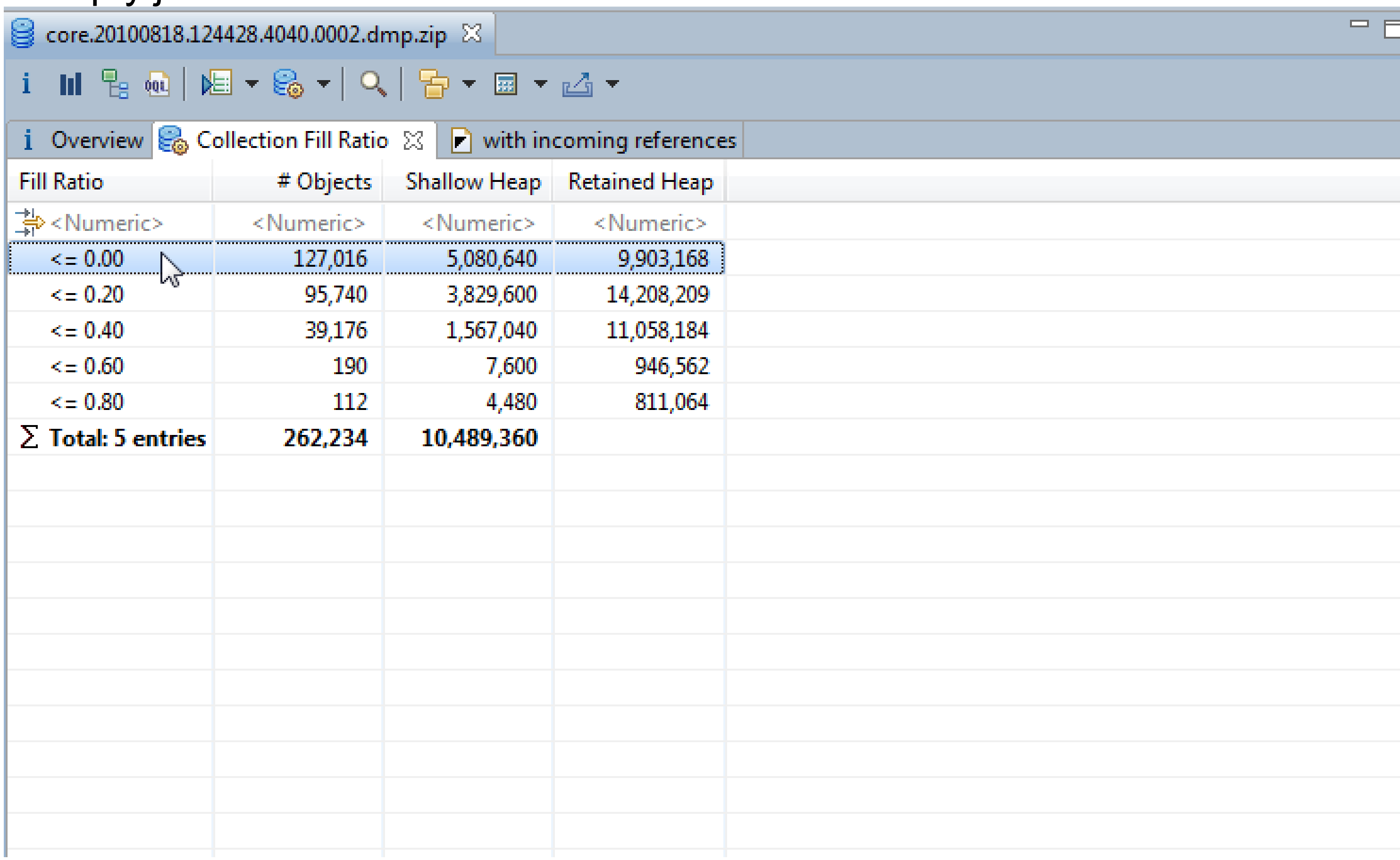
# Analyzing your Collections

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



# Analyzing your Collections

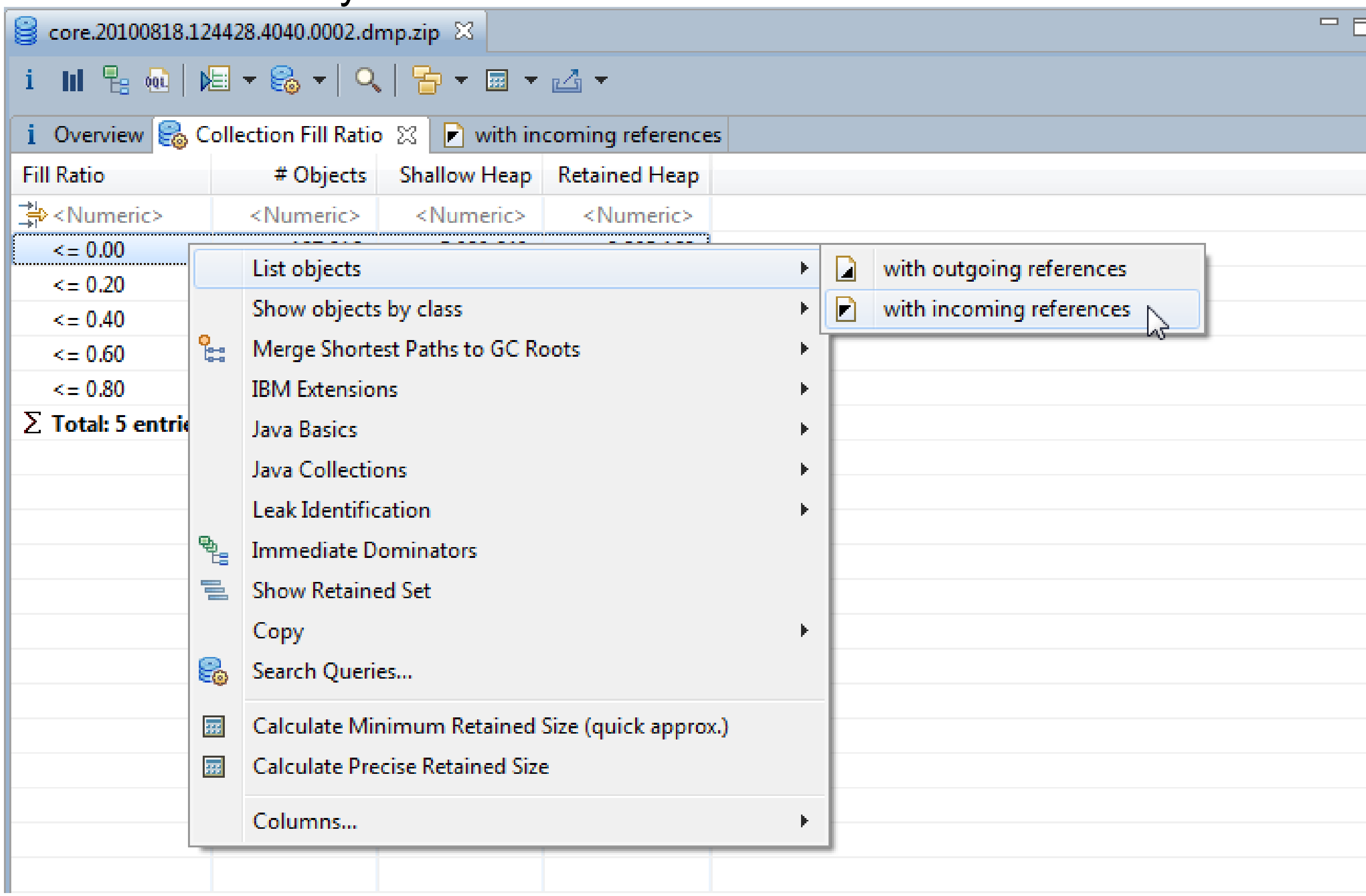
- Shows 127,016 empty java.util.Hashtable instances!




| 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** | |

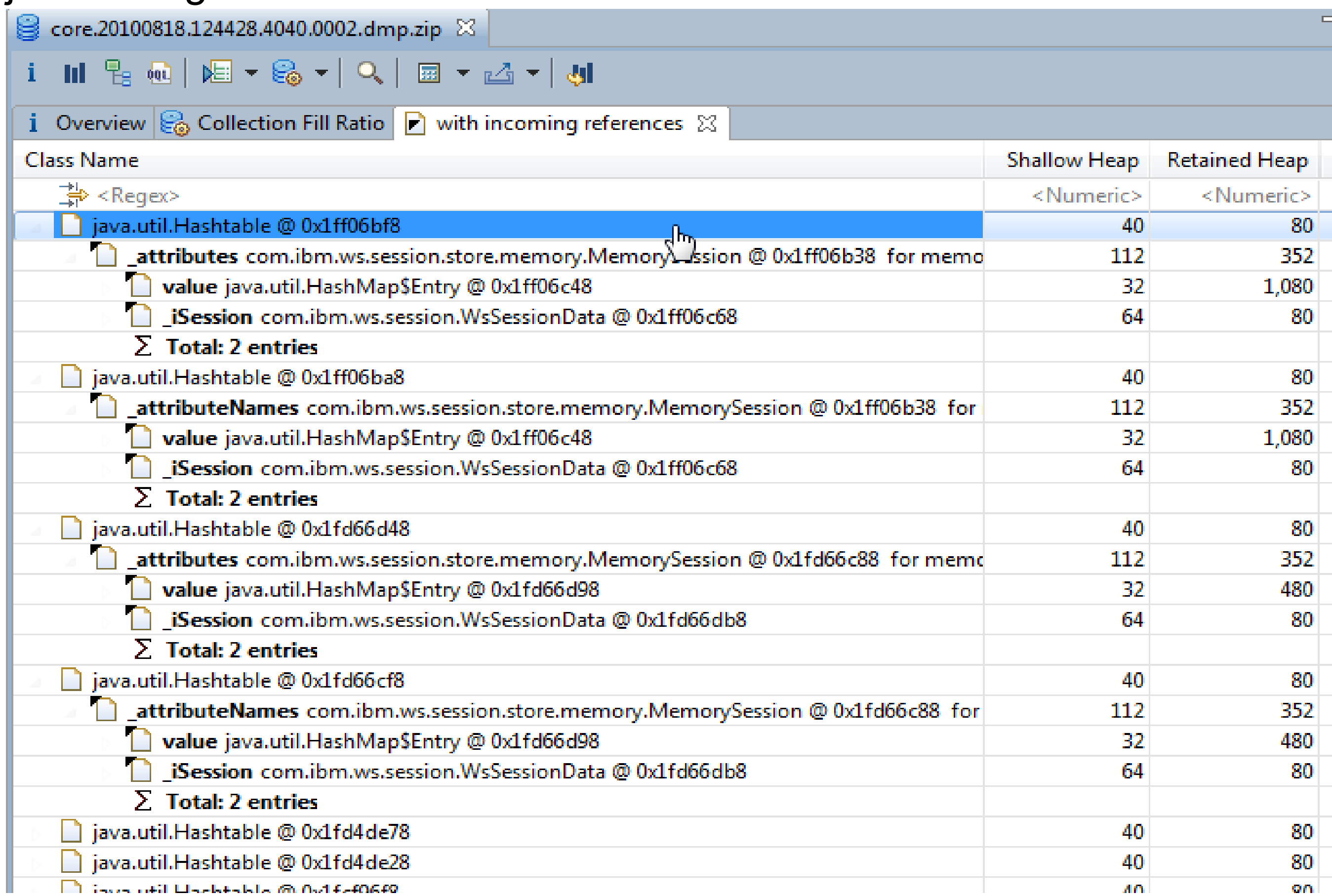
# Analyzing your Collections

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



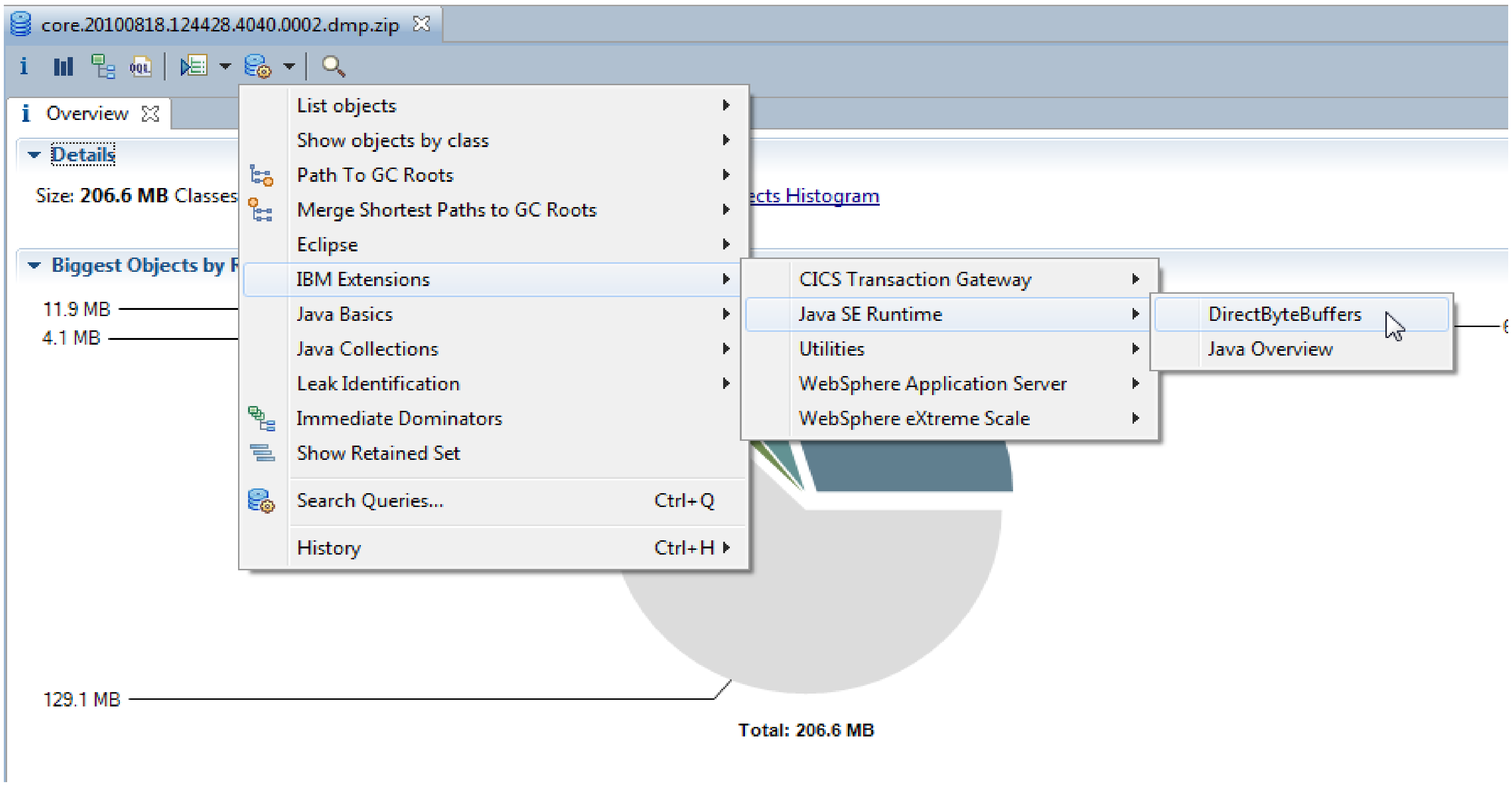
# Analyzing your Collections

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



# Analyzing your DirectByteBuffers

- IBM Extensions for Memory Analyzer (IEMA) provides DirectByteBuffer analysis:



# 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

# Read the Article:

From Java Code to Java Heap on IBM developerWorks:

http://www.ibm.com/developerworks/java/library/j-codetoheap/index.html

▪

# Get the Slides:

From Java Code to Java Heap on Slideshare.net:

http://www.slideshare.net/cnbailey/java-code-to-java-heap

▪

# 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_sup

- **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