From Java code to Java heap

Understanding and optimizing your application's memory usage

Chris Bailey (baileyc@uk.ibm.com)
Java Service Architect
IBM

Skill Level: Intermediate

Date: 29 Feb 2012

This article gives you insight into the memory usage of Java™ code, covering the memory overhead of putting an `int` value into an `Integer` object, the cost of object delegation, and the memory efficiency of the different collection types. You'll learn how to determine where inefficiencies occur in your application and how to choose the right collections to improve your code.

Although the subject of optimizing your application code's memory usage isn't new, it's not one that is generally well understood. This article briefly covers the memory usage of a Java process, then digs in depth into the memory usage of the Java code that you write. Finally, it shows ways to make your application code more memory-efficient, particularly in the area of using Java collections such as `HashMap`s and `ArrayList`s.

Background: Memory usage of a Java process

Native intelligence

For a more in-depth understanding of the process-memory usage of a Java application, read Andrew Hall's "Thanks for the memory" articles on developerWorks. They cover the layout and user space available on Windows® and Linux® and on AIX®, and the interaction between the Java heap and native heap.

When you run a Java application by executing `java` on the command line or by starting some Java-based middleware, the Java runtime creates an operating-system process — just as if you were running a C-based program. In fact, most JVMs are written largely in C or C++. As an operating-system process, the Java runtime faces the same restrictions on memory as any other process: the addressability provided by the architecture, and the user space provided by the operating system.

The memory addressability provided by the architecture depends on the bit size of the processor — for example, 32 or 64 bits, or 31 bits in the case of the mainframe. The number of bits the process can handle determines the range of memory that the processor is capable of addressing: 32 bits provides an addressable range of 2^32, which is 4,294,967,296 bits, or 4GB. The addressable range for a 64-bit processor is significantly larger: 2^64 is 18,446,744,073,709,551,616, or 16 exabytes.

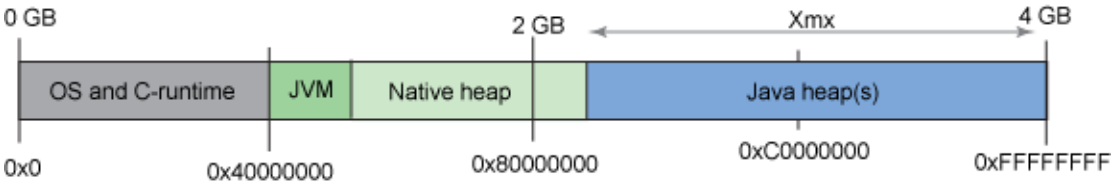
Recommended Resources

- "IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer"
- "Debugging from dumps"
- "IBM on troubleshooting Java applications"

Some of the addressable range provided by the processor architecture is used by the OS itself for its kernel and (for JVMs written in C or C++) for the C runtime. The amount of memory used by the OS and C runtime depends on the OS being used, but it is usually significant: the default usage by Windows is 2GB. The remaining addressable space — termed the *user space* — is the memory available to the actual process that's running.

For Java applications, then, the user space is the memory used by the Java process, effectively consisting of two pools: the Java heap(s) and the *native* (non-Java) heap. The size of the Java heap is controlled by the JVM's Java heap settings: `-Xms` and `-Xmx` set the minimum and maximum Java heap, respectively. The native heap is the user space left over after the Java heap has been allocated at the maximum size setting. Figure 1 shows an example of what this might look like for a 32-bit Java process:

Figure 1. Example memory layout for a 32-bit Java process



In Figure 1, the OS and C runtime use about 1GB of the 4GB of addressable range, the Java heap uses almost 2GB, and the native heap uses the rest. Note that the JVM itself uses memory — the same way the OS kernel and C runtime do — and that the memory the JVM uses is a subset of the native heap.

Anatomy of a Java object

When your Java code uses the `new` operator to create an instance of a Java object, much more data is allocated than you might expect. For example, it might surprise you to know that the size ratio of an `int` value to an `Integer` object — the smallest object that can hold an `int` value — is typically 1:4. The additional overhead is metadata that the JVM uses to describe the Java object, in this case an `Integer`.

The amount of object metadata varies by JVM version and vendor, but it typically consists of:

- **Class** : A pointer to the class information, which describes the object type. In the case of a `java.lang.Integer` object, for example, this is a pointer to the `java.lang.Integer` class.
- **Flags** : A collection of flags that describe the state of the object, including the hash code for the object if it has one, and the *shape* of the object (that is, whether or not the object is an array).
- **Lock** : The synchronization information for the object — that is, whether the object is currently synchronized.

The object metadata is then followed by the object data itself, consisting of the fields stored in the object instance. In the case of a `java.lang.Integer` object, this is a single `int`.

So, when you create an instance of a `java.lang.Integer` object when running a 32-bit JVM, the layout of the object might look like Figure 2:

Figure 2. Example layout of a `java.lang.Integer` object for a 32-bit Java process

As Figure 2 shows, 128 bits of data are used to store the 32 bits of data in the `int` value, because the object metadata uses the rest of those 128 bits.

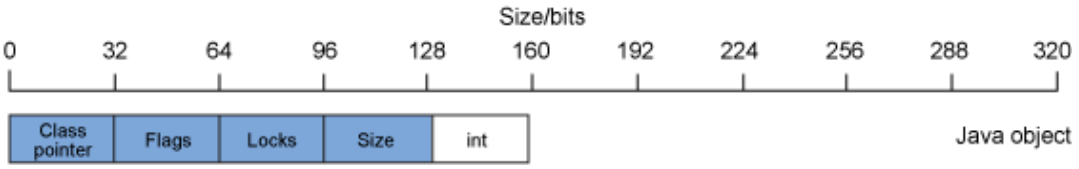
Anatomy of a Java array object

The shape and structure of an array object, such as an array of `int` values, is similar to that of a standard Java object. The primary difference is that the array object has an additional piece of metadata that denotes the array's size. An array object's metadata, then, consists of:

- **Class** : A pointer to the class information, which describes the object type. In the case of an array of `int` fields, this is a pointer to the `int[]` class.
- **Flags** : A collection of flags that describe the state of the object, including the hash code for the object if it has one, and the shape of the object (that is, whether or not the object is an array).
- **Lock** : The synchronization information for the object — that is, whether the object is currently synchronized.
- **Size** : The size of the array.

Figure 3 shows an example layout for an `int` array object:

Figure 3. Example layout of an `int` array object for a 32-bit Java process

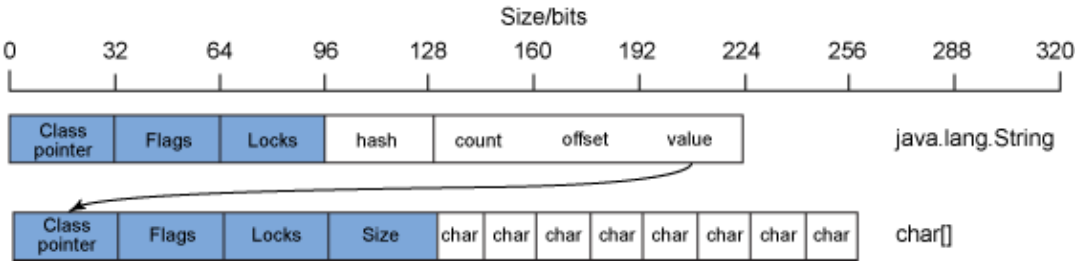


In Figure 3, 160 bits of data store the 32 bits of data in the `int` value, because the array metadata uses the rest of those 160 bits. For primitives such as `byte`, `int`, and `long`, a single-entry array is more expensive in terms of memory than the corresponding wrapper object (`Byte`, `Integer`, or `Long`) for the single field.

Anatomy of more-complex data structures

Good object-oriented design and programming encourage the use of *encapsulation* (providing interface classes that control access to data) and *delegation* (the use of helper objects to carry out tasks). Encapsulation and delegation cause the representation of most data structures to involve multiple objects. A simple example is a `java.lang.String` object. The data in a `java.lang.String` object is an array of characters that is encapsulated by a `java.lang.String` object that manages and controls access to the character array. The layout of a `java.lang.String` object for a 32-bit Java process might look like Figure 4:

Figure 4. Example layout of a `java.lang.String` object for a 32-bit Java process



As Figure 4 shows, a `java.lang.String` object contains — in addition to the standard object metadata — some fields to manage the string data. Typically, these fields are a hash value, a count of the size of the string, the offset into the string data, and an object reference to the character array itself.

This means that to have a string of 8 characters (128 bits of `char` data), 256 bits of data are for the character array and 224 bits of data are for the `java.lang.String` object that manages it, making a total of 480 bits (60 bytes) to represent 128 bits (16 bytes) of data. This is an overhead ratio of 3.75:1.

In general, the more complex a data structure becomes, the greater its overhead. This is discussed in more detail in the next section.

32-bit and 64-bit Java objects

The sizes and overhead for the objects in the preceding examples apply to a 32-bit Java process. As you know from the Background: Memory usage of a Java process section, a 64-bit processor has a much higher level of memory addressability than a 32-bit processor. With a 64-bit process, the size of some of the data fields in the Java object — specifically, the object metadata and any field that refers to another object — also need to increase to 64 bits. The other data-field types — such as `int`, `byte`, and `long` — do not change in size. Figure 5 shows the layout for a 64-bit `Integer` object and for an `int` array:

Figure 5. Example layout of a `java.lang.Integer` object and an `int` array for a 64-bit Java process

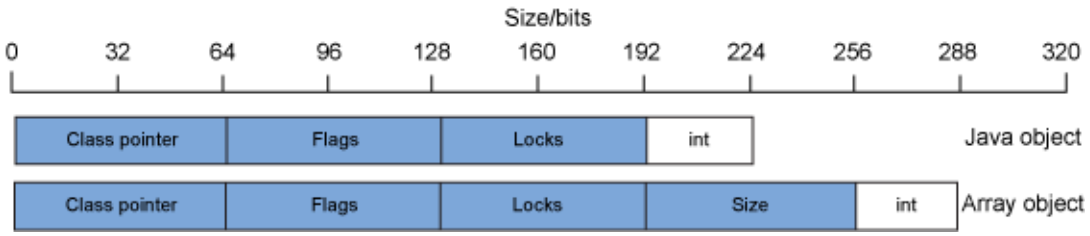


Figure 5 shows that for a 64-bit `Integer` object, 224 bits of data are now being used to store the 32 bits used for the `int` field — an overhead ratio of 7:1. For a 64-bit single-element `int` array, 288 bits of data are used to store the 32-bit `int` entry — an overhead of 9:1. The effect of this on real applications is that the Java heap memory usage of an application that previously ran on a 32-bit Java runtime increases dramatically when it's moved to a 64-bit Java runtime. Typically, the increase is on the order of 70 percent of the original heap size. For example, a Java application using 1GB of Java heap with the 32-bit Java runtime will typically use 1.7GB of Java heap with the 64-bit Java runtime.

Note that this memory increase is not limited to the Java heap. The native-heap memory-area usage will also increase, sometimes by as much as 90 percent.

Table 1 shows the field sizes for objects and arrays when an application runs in 32-bit and 64-bit mode:

Table 1. Field sizes in objects for 32-bit and 64-bit Java runtimes

Field type	Field size (bits)			
	Object		Array	
	32-bit	64-bit	32-bit	64-bit
`boolean`	32	32	8	8
`byte`	32	32	8	8
`char`	32	32	16	16
`short`	32	32	16	16

`int`	32	32	32	32
`float`	32	32	32	32
`long`	64	64	64	64
`double`	64	64	64	64
Object fields	32	64 (32*)	32	64 (32*)
Object metadata	32	64 (32*)	32	64 (32*)

*The size of the object fields and of the data used for the each of the object-metadata entries can be reduced to 32 bits via the Compressed References or Compressed OOPs technologies.

Compressed References and Compressed Ordinary Object Pointers (OOPs)

IBM and Oracle JVMs both provide object-reference compression capabilities via the Compressed References (`-Xcompressedrefs`) and Compressed OOPs (`-XX:+UseCompressedOops`) options, respectively. Use of these options enables the object fields and the object metadata values to be stored in 32 bits rather than 64 bits. This has the effect of negating the 70 percent Java-heap memory increase when an application is moved from a 32-bit Java runtime to a 64-bit Java runtime. Note that the options have no effect on the memory usage of the native heap; it is still higher with the 64-bit Java runtime than with the 32-bit Java runtime.

Memory usage of Java collections

In most applications, a large amount of data is stored and managed using the standard Java Collections classes provided as part of the core Java API. If memory-footprint optimization is important for your application, it's especially useful to understand the function each collection provides and the associated memory overhead. In general, the higher the level of a collection's functional capabilities, the higher its memory overhead — so using collection types that provide more function than you require will incur unnecessary additional memory overhead.

Some of the commonly used collections are:

- `HashSet`
- `HashMap`
- `Hashtable`
- `LinkedList`
- `ArrayList`

With the exception of `HashSet`, this list is in decreasing order of both function and memory overhead. (A `HashSet`, being a wrapper around a `HashMap` object, effectively provides less function than `HashMap` whilst being slightly larger.)

Java Collections: `HashSet`

A `HashSet` is an implementation of the `Set` interface. The Java Platform SE 6 API documentation describes `HashSet` as:

> 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.

A `HashSet` has fewer capabilities than a `HashMap` in that it cannot contain more than one null entry and cannot have duplicate entries. The implementation is a wrapper around a `HashMap`, with the `HashSet` object managing what is allowed to be put into the `HashMap` object. The additional function of restricting the capabilities of a `HashMap` means that `HashSet`s have a slightly higher memory overhead.

Figure 6 shows the layout and memory usage of a `HashSet` on a 32-bit Java runtime:

Figure 6. Memory usage and layout of a `HashSet` on a 32-bit Java runtime

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

Figure 6 shows the *shallow heap* (memory usage of the individual object) in bytes, along with the *retained heap* (memory usage of the individual object and its child objects) in bytes for a `java.util.HashSet` object. The shallow heap size is 16 bytes, and the retained heap size is 144 bytes. When a `HashSet` is created, its *default capacity* — the number of entries that can be put into the set — is 16 entries. When a `HashSet` is created at the default capacity and no entries are put into the set, it occupies 144 bytes. This is an extra 16 bytes over the memory usage of a `HashMap`. Table 2 shows the attributes of a `HashSet`:

Table 2. Attributes of a `HashSet`

Default capacity	16 entries
Empty size	144 bytes
Overhead	16 bytes plus `HashMap` overhead
Overhead for a 10K collection	16 bytes plus `HashMap` overhead
Search/insert/delete performance	O(1) — Time taken is constant time, regardless of the number of elements (assuming no hash collisions)

Java Collections: `HashMap`

A `HashMap` is an implementation of the `Map` interface. The Java Platform SE 6 API documentation describes `HashMap` as:

> An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

`HashMap` provides a way of storing key/value pairs, using a hashing function to transform the key into an index into the collection where the key/value pair is stored.

This allows for fast access to the data location. Null entries and duplicate entries are allowed; as such, a `HashMap` is a simplification of a `HashSet`.

The implementation of a `HashMap` is as an array of `HashMap$Entry` objects. Figure 7 shows the memory usage and layout of a `HashMap` on a 32-bit Java runtime:

Figure 7. Memory usage and layout of a `HashMap` on a 32-bit Java runtime

As Figure 7 shows, when a `HashMap` is created, the result is a `HashMap` object and an array of `HashMap$Entry` objects at its default capacity of 16 entries. This gives a `HashMap` a size of 128 bytes when it is completely empty. Any key/value pairs inserted into the `HashMap` are wrapped by a `HashMap$Entry` object, which itself has some overhead.

Most implementations of `HashMap$Entry` objects contain the following fields:

- `int KeyHash`
- `Object next`
- `Object key`
- `Object value`

A 32-byte `HashMap$Entry` object manages the key/value pairs of data that are put into the collection. This means that the total overhead of a `HashMap` consists of the `HashMap` object, a `HashMap$Entry` array entry, and a `HashMap$Entry` object for each entry. This can be expressed by the formula:

> `HashMap` object + Array object overhead + (number of entries * (`HashMap$Entry` array entry + `HashMap$Entry` object))

For a 10,000-entry `HashMap`, the overhead of just the `HashMap`, `HashMap$Entry` array, and `HashMap$Entry` objects is approximately 360K. This is before the size of the keys and values being stored is taken into account.

Table 3 shows `HashMap`'s attributes:

Table 3. Attributes of a `HashMap`

Default capacity	16 entries
Empty size	128 bytes
Overhead	64 bytes plus 36 bytes per entry
Overhead for a 10K collection	~360K
Search/insert/delete performance	O(1) — Time taken is constant time, regardless of the number of elements (assuming no hash collisions)

Java Collections: `Hashtable`

`Hashtable`, like `HashMap`, is an implementation of the `Map` interface. The Java Platform SE 6 API documentation's description of `Hashtable` is:

> This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

`Hashtable` is very similar to `HashMap`, but it has two limitations. It cannot accept null values for either the key or the value entries, and it is a synchronized collection. In contrast, `HashMap` can accept null values and is not synchronized but can be made synchronized using the `Collections.synchronizedMap()` method.

The implementation of `Hashtable` — also similar to `HashMap`'s — is as an array of entry objects, in this case `Hashtable$Entry` objects. Figure 8 shows the memory usage and layout of a `Hashtable` on a 32-bit Java runtime:

Figure 8. Memory usage and layout of a `Hashtable` on a 32-bit Java runtime

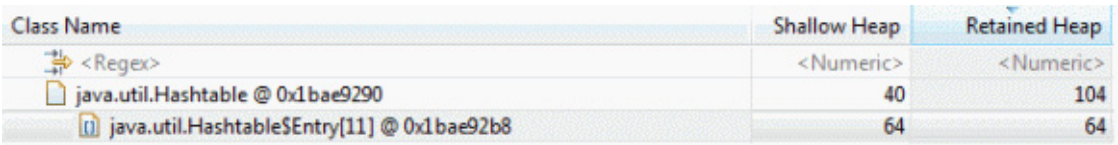


Figure 8 shows that when a `Hashtable` is created, the result is a `Hashtable` object using 40 bytes of memory along with an array of `Hashtable$entry`s with a default capacity of 11 entries, totaling a size of 104 bytes for an empty `Hashtable`.

`Hashtable$Entry` stores effectively the same data as `HashMap`:

- `int KeyHash`
- `Object next`
- `Object key`
- `Object value`

This means that the `Hashtable$Entry` object is also 32 bytes for key/value entry in the `Hashtable`, and the calculation for `Hashtable` overhead and size of a 10K entry collection (approximately 360K) is similar to `HashMap`'s.

Table 4 shows the attributes of a `Hashtable`:

Table 4. Attributes of a `Hashtable`

Default capacity	11 entries
Empty size	104 bytes
Overhead	56 bytes plus 36 bytes per entry
Overhead for a 10K collection	~360K
Search/insert/delete performance	O(1) — Time taken is constant time, regardless of the number of elements (assuming no hash collisions)

As you can see, `Hashtable` has a slightly smaller default capacity than `HashMap` (11 vs. 16). Otherwise, the main differences are `Hashtable`'s inability to accept null keys and values, and its default synchronisation, which may not be needed and reduces the collection's performance.

Java Collections: `LinkedList`

A `LinkedList` is a linked-list implementation of the `List` interface. The Java Platform SE 6 API documentation describes `LinkedList` as:

> 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.

The implementation is a linked list of `LinkedList$Entry` objects. Figure 9 shows the memory usage and layout of `LinkedList` on a 32-bit Java runtime:

Figure 9. Memory usage and layout of a `LinkedList` on a 32-bit Java runtime

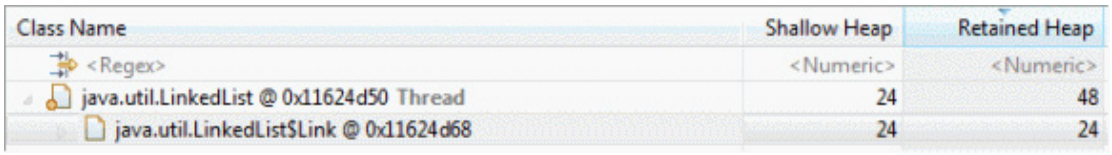


Figure 9 shows that when a `LinkedList` is created, the result is a `LinkedList` object using 24 bytes of memory along with a single `LinkedList$Entry` object, totaling 48 bytes of memory for an empty `LinkedList`.

One of the advantages of linked lists is that they are accurately sized and do not need to be resized. The default capacity is effectively one entry, and this grows and shrinks dynamically as more entries are added or removed. There is still an overhead for each `LinkedList$Entry` object, whose data fields are:

- `Object previous`
- `Object next`
- `Object value`

But this is smaller than the overhead of `HashMap`s and `Hashtable`s, because linked lists store only a single entry rather than a key/value pair, and there's no need to store a hash value because array-based lookups are not used. On the negative side, lookups into a linked list can be much slower, because the linked list must be traversed in order for the correct entry to be found. For large linked lists, that can result in long lookup times.

Table 5 shows the attributes of a `LinkedList`:

Table 5. Attributes of a `LinkedList`

Default capacity	1 entry
Empty size	48 bytes
Overhead	24 bytes, plus 24 bytes per entry
Overhead for a 10K collection	~240K
Search/insert/delete performance	O(n) — Time taken is linearly dependent on the number of elements

Java Collections: `ArrayList`

An `ArrayList` is a resizeable array implementation of the `List` interface. The Java Platform SE 6 API documentation describes `ArrayList` as:

> 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.

Unlike `LinkedList`, `ArrayList` is implemented using an array of `Objects`. Figure 10 shows the memory usage and layout of an `ArrayList` on a 32-bit Java runtime:

Figure 10. Memory usage and layout of an `ArrayList` on a 32-bit Java runtime

Figure 10 shows that when an `ArrayList` is created, the result is an `ArrayList` object using 32 bytes of memory, along with an `Object` array at a default size of 10, totaling 88 bytes of memory for an empty `ArrayList` This means that the `ArrayList` is not accurately sized and therefore has a default capacity, which happens to be 10 entries.

Table 6 shows attributes of an `ArrayList`:

Table 6. Attributes of an `ArrayList`

Default capacity	10
Empty size	88 bytes
Overhead	48 bytes plus 4 bytes per entry
Overhead for 10K collection	~40K
Search/insert/delete performance	O(n) — Time taken is linearly dependent to the number of elements

Other types of "collections"

In addition to the standard collections, `StringBuffer` can also be considered a collection in that it manages character data and is similar in structure and capabilities

to the other collections. The Java Platform SE 6 API documentation describes `StringBuffer` as:

> 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.

The implementation of a `StringBuffer` is as an array of `char`s. Figure 11 shows the memory usage and layout of a `StringBuffer` on a 32-bit Java runtime:

Figure 11. Memory usage and layout of a `StringBuffer` on a 32-bit Java runtime

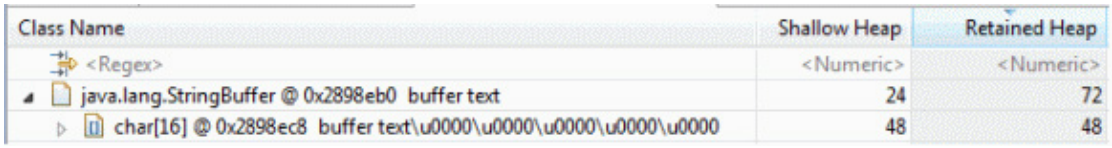


Figure 11 shows that when a `StringBuffer` is created, the result is a `StringBuffer` object using 24 bytes of memory, along with a character array with a default size of 16, totaling 72 bytes of data for an empty `StringBuffer`.

Like the collections, `StringBuffer` has a default capacity and a mechanism for resizing. Table 7 shows the attributes of `StringBuffer`:

Table 7. Attributes of a `StringBuffer`

Default capacity	16
Empty size	72 bytes
Overhead	24 bytes
Overhead for 10K collection	24 bytes
Search/Insert/Delete performance	NA

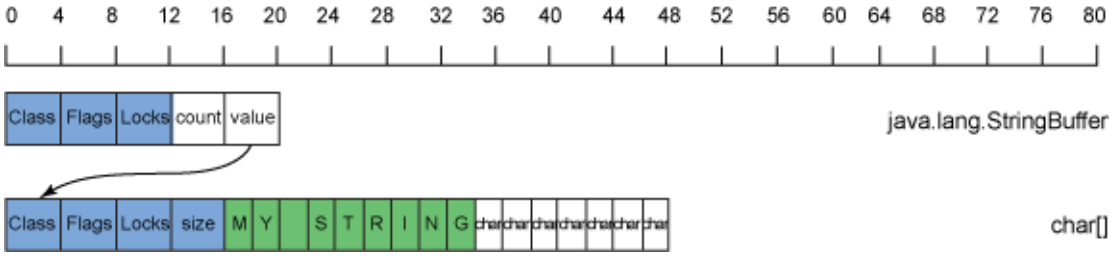
Empty space in collections

The overhead of the various collections with a given number of objects is not the whole memory-overhead story. The measurements in the preceding examples assume that the collections have been accurately sized. But for most collections, this is unlikely to be true. Most collections are created with a given initial capacity, and data is put into the collection. This means that it is common for collections to have a capacity that is greater than the data being stored in the collection, which introduces additional overhead.

Consider the example of a `StringBuffer`. Its default capacity is 16 character entries, with a size of 72 bytes. Initially, no data is being stored in the 72 bytes. If you put some characters into the character array — for example `"MY STRING"` — then you

are storing 9 characters in the 16-character array. Figure 12 shows the memory usage of a `StringBuffer` containing `"MY STRING"` on a 32-bit Java runtime:

Figure 12. Memory usage of a `StringBuffer` containing `"MY STRING"` on a 32-bit Java runtime



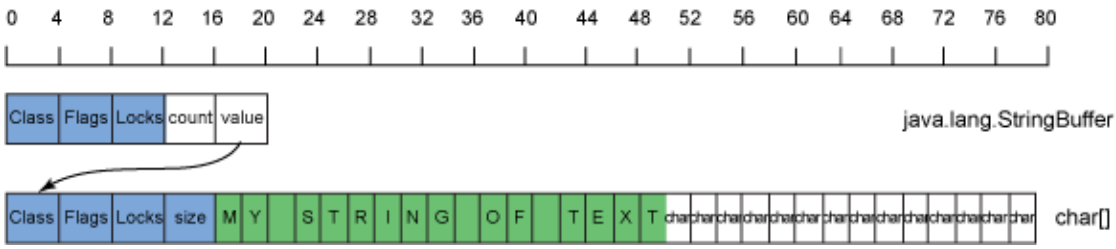
As Figure 12 shows, 7 additional character entries available in the array are not being used but are consuming memory — in this case an additional overhead of 112 bytes. For this collection you have 9 entries in a capacity of 16, which gives you a *fill ratio* of 0.56. The lower a collection's fill ratio, the greater the overhead that is due to spare capacity.

Expansion and resizing of collections

After a collection reaches its capacity and a request is made to put additional entries into the collection, the collection is resized and expanded to accommodate new entries. This increases the capacity but often lowers the fill ratio and introduces greater memory overhead.

The expansion algorithm used differs amongst the collections, but a common approach is to double the capacity of the collection. This is the approach taken for `StringBuffer`. In the case of the preceding example's `StringBuffer`, if you wanted to append `" OF TEXT"` to the buffer to produce `"MY STRING OF TEXT"`, you need to expand the collection, because your new collection of characters has 17 entries against a current capacity of 16. Figure 13 shows the resulting memory usage:

Figure 13. Memory usage of a `StringBuffer` containing `"MY STRING OF TEXT"` on a 32-bit Java runtime



Now, as Figure 13 shows, you have a 32-entry character array and 17 used entries, giving you a fill ratio of 0.53. The fill ratio hasn't dropped dramatically, but you now have an overhead of 240 bytes for the spare capacity.

In the case of small strings and collections, the overheads for low fill ratios and spare capacity might not seem to be too much of a problem, but they become much more

apparent and expensive at greater sizes. For example, if you create a `StringBuffer` that contains just 16MB of data, it will be (by default) using a character array that is sized to hold up to 32MB of data — creating 16MB of additional overhead in the form of spare capacity.

Java Collections: Summary

Table 8 summarizes the attributes of the collections:

Table 8. Summary of collections attributes

Collection	Performance	Default capacity	Empty size	10K entry overhead	Accurately sized?	Expansion algorithm
`HashSet`	O(1)	16	144	360K	No	x2
`HashMap`	O(1)	16	128	360K	No	x2
`Hashtable`	O(1)	11	104	360K	No	x2+1
`LinkedList`	O(n)	1	48	240K	Yes	+1
`ArrayList`	O(n)	10	88	40K	No	x1.5
`StringBuffer`	O(1)	16	72	24	No	x2

The performance of the `Hash` collections is much better than that of either of the `List`s, but at a much greater per-entry cost. Because of the access performance, if you are creating large collections (for example, to implement a cache), it's better to use a `Hash`-based collection, regardless of the additional overhead.

For smaller collections for which the access performance is less of an issue, `List`s become an option. The performance of the `ArrayList` and the `LinkedList` collections is approximately the same, but their memory footprints differ: the per-entry size of the `ArrayList` is much smaller than the `LinkedList`, but it is not accurately sized. Whether an `ArrayList` or a `LinkedList` is the right implementation of `List` to use depends on how predictable the length of the `List` is likely to be. If the length is unknown, a `LinkedList` may be the right option, because the collection will contain less empty space. If the size is known, an `ArrayList` will have much less memory overhead.

Choosing the correct collection type enables you to select the right balance between collection performance and memory footprint. In addition, you can minimize the memory footprint by correctly sizing the collection to maximize fill ratio and to minimize unused space.

Collections in use: PlantsByWebSphere and WebSphere Application Server Version 7

In Table 8, the overhead of creating a 10,000-entry `Hash`-based collection is shown to be 360K. Given that it's not uncommon for complex Java applications to run with Java heaps sized in gigabytes, this does not seem like a large overhead — unless, of course, a large number of collections are being used.

Table 9 shows the collection-object usage as part of the 206MB of Java heap usage when the PlantsByWebSphere sample application supplied with WebSphere® Application Server Version 7 runs under a five-user load test:

Table 9. Collection usage by PlantsByWebSphere on WebSphere Application Server v7

Collection type	Number of instances	Total collection overhead (MB)
`Hashtable`	262,234	26.5
`WeakHashMap`	19,562	12.6
`HashMap`	10,600	2.3
`ArrayList`	9,530	0.3
`HashSet`	1,551	1.0
`Vector`	1,271	0.04
`LinkedList`	1,148	0.1
`TreeMap`	299	0.03
Total	**306,195**	**42.9**

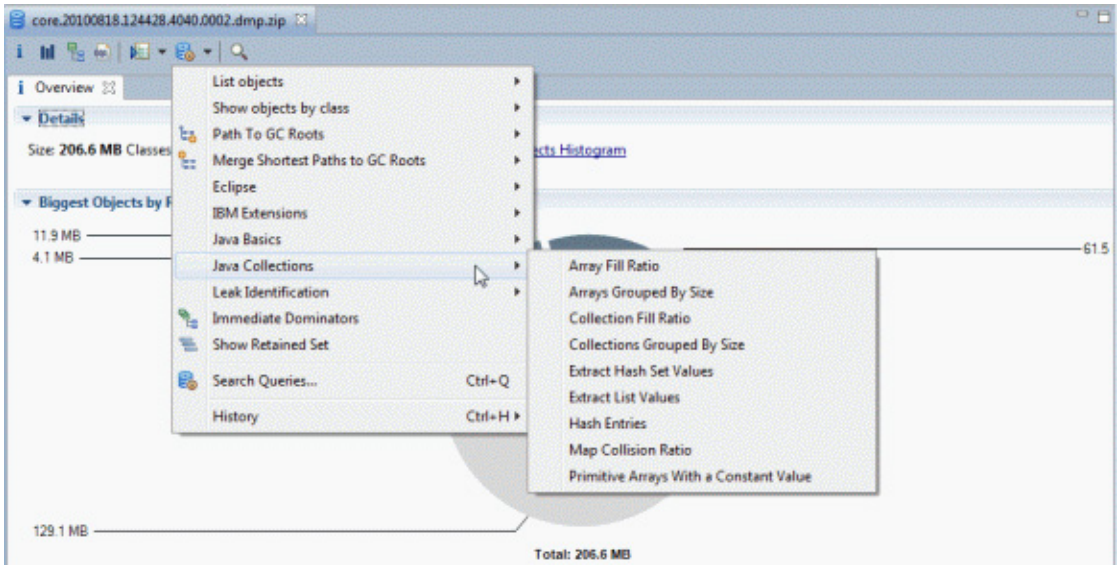
You can see from Table 9 that more than 300,000 different collections are being used — and that the collections themselves, not counting the data they contain, account for 42.9MB (21 percent) of the 206MB Java heap usage. This means that substantial potential memory savings are available if you either change collection types or ensure that the sizes of the collections are more accurate.

Looking for low fill ratios with Memory Analyzer

The IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer tool (Memory Analyzer) that is available as part of the IBM Support Assistant can analyze Java collections' memory usage (see Resources). Its capabilities include analysis of fill ratios and the sizes of collections. You can use this analysis to identify any collections that are candidates for optimization.

The collection-analysis capabilities in Memory Analyzer are located under the Open Query Browser -> Java Collections menu, as shown in Figure 14:

Figure 14. Analysis of the fill ratio of Java collections in Memory Analyzer

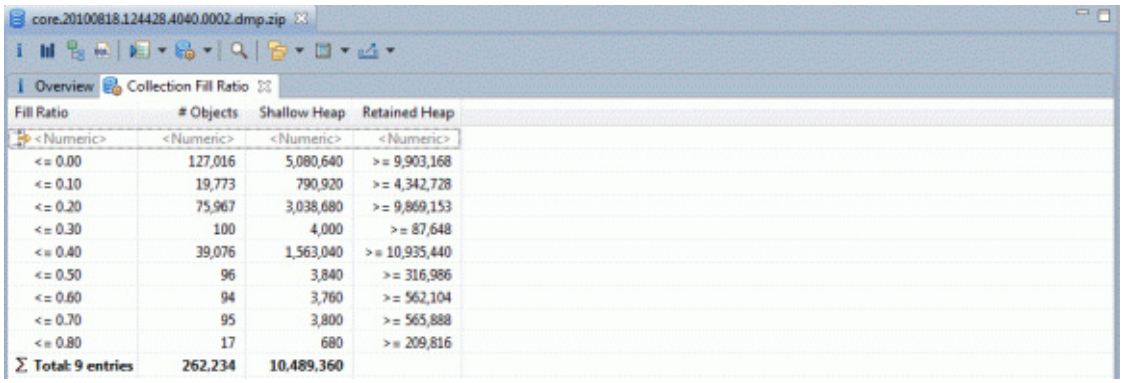


The Collection Fill Ratio query selected in Figure 14 is the most useful for identifying collections that are much larger than currently required. You can specify a number of options for this query, including:

- **objects** : The types of objects (collections) you are interested in
- **segments** : The fill ratio ranges to group the objects into

Running the query with the objects options set to "java.util.Hashtable" and the segments option set to "10" produces the output shown in Figure 15:

Figure 15. Analysis in Memory Analyzer of the fill ratio of `Hashtables`



Fill Ratio	# Objects	Shallow Heap	Retained Heap
<Numeric>	<Numeric>	<Numeric>	<Numeric>
<= 0.00	127,016	5,080,640	>= 9,903,168
<= 0.10	19,773	790,920	>= 4,342,728
<= 0.20	75,967	3,038,680	>= 9,869,153
<= 0.30	100	4,000	>= 87,648
<= 0.40	39,076	1,563,040	>= 10,935,440
<= 0.50	96	3,840	>= 316,986
<= 0.60	94	3,760	>= 562,104
<= 0.70	95	3,800	>= 565,888
<= 0.80	17	680	>= 209,816
Σ Total: 9 entries	262,234	10,489,360	

Figure 15 shows that of the 262,234 instances of `java.util.Hashtable`, 127,016 (48.4 percent) of them are completely empty, and that almost all of them only have a small number of entries.

It's then possible to identify these collections by selecting a row of the results table and right-clicking to select either **list objects -> with incoming references** to see what objects own those collections or **list objects -> with outgoing references** to

see what is inside those collections. Figure 16 shows the results of looking at the incoming references for the empty `Hashtable`s and expanding a couple of the entries:

Figure 16. Analysis of the incoming references to empty `Hashtable`s in Memory Analyzer

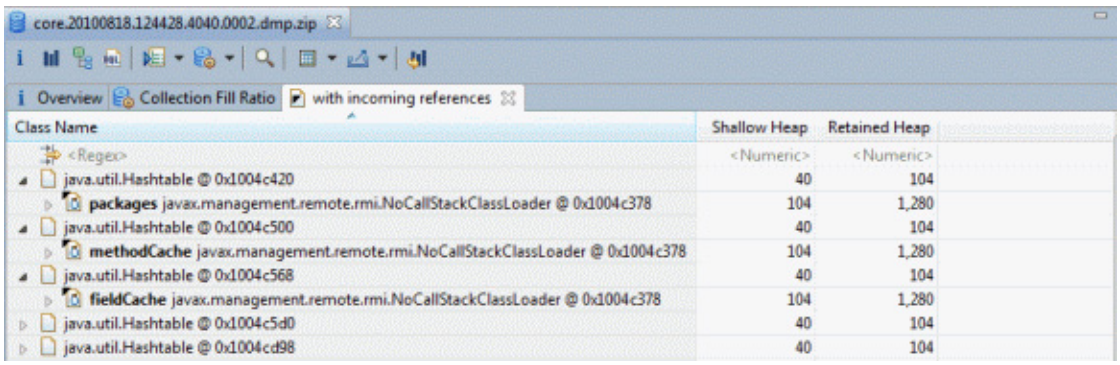


Figure 16 shows that some of the empty `Hashtable`s are owned by the `javax.management.remote.rmi.NoCallStackClassLoader` code.

By looking at the **Attributes** view in the left-hand panel of Memory Analyzer, you can see specific details about the `Hashtable` itself, as shown in Figure 17:

Figure 17. Inspection of the empty `Hashtable` in Memory Analyzer

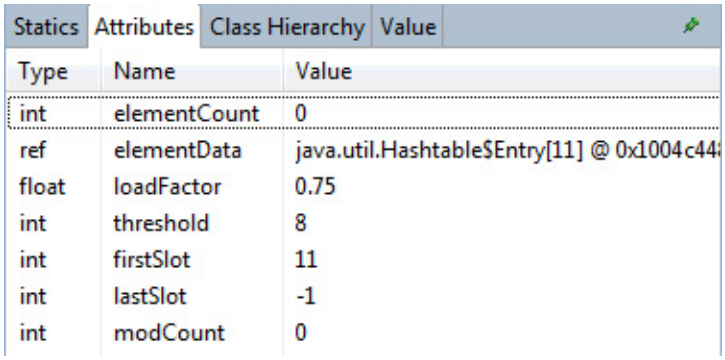


Figure 17 shows that the `Hashtable` has a size of 11 (the default size) and that it is completely empty.

For the `javax.management.remote.rmi.NoCallStackClassLoader` code, it might be possible to optimize the collection usage by:

- Lazily allocating the `Hashtable`: If it is common for the `Hashtable` to be empty, then it may make sense for the `Hashtable` to be allocated only when there is data to store inside it.
- Allocating the `Hashtable` to an accurate size: Because the default size has been used, it's possible that a more accurate initial size could be used.

Whether either or both of these optimizations are applicable depends on how the code is commonly used, and what data is commonly stored inside it.

Empty collections in the PlantsByWebSphere example

Table 10 shows the result of analyzing collections in the PlantsByWebSphere example to identifying those that are empty:

Table 10. Empty-collection usage by PlantsByWebSphere on WebSphere Application Server v7

Collection type	Number of instances	Empty instances	% Empty
`Hashtable`	262,234	127,016	48.4
`WeakHashMap`	19,562	19,465	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**

Table 10 shows that on average, over 50 percent of the collections are empty, implying that significant memory-footprint savings could be gained by optimization of collection usage. It could be applied to various levels of the application: in the PlantsByWebSphere example code, in the WebSphere Application Server, and in the Java collections classes themselves.

Between WebSphere Application Server version 7 and version 8, some work has been done to improve memory efficiency in the Java collections and middleware layers. For example, a large percentage of the overhead of instances of `java.util.WeahHashMap` is due to the fact that it contains an instance of `java.lang.ref.ReferenceQueue` to handle the weak references. Figure 18 shows the memory layout of a `WeakHashMap` for a 32-bit Java runtime:

Figure 18. Memory layout of a `WeakHashMap` for a 32-bit Java runtime

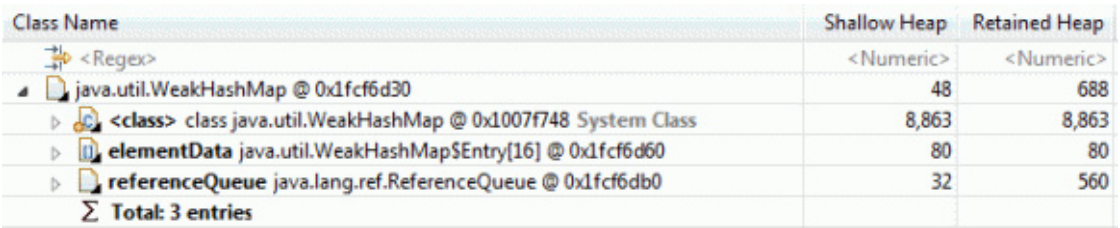


Figure 18 shows that the `ReferenceQueue` object is responsible for retaining 560 bytes' worth of data, even if the `WeakHashMap` is empty and `ReferenceQueue` is therefore not required. For the PlantsByWebSphere example case with 19,465 empty `WeakHashMaps`, the `ReferenceQueue` objects are adding an additional 10.9MB of data that is not required. In WebSphere Application Server version 8 and the Java 7 release of the IBM Java runtimes, the `WeakHashMap` has undergone some optimization: It contains a `ReferenceQueue`, which in turn contains an array of

`Reference` objects. That array has been changed to be allocated lazily — that is, only when objects are added to the `ReferenceQueue`.

Conclusion

A large and perhaps surprising number of collections exist in any given application, and more so for complex applications. Use of a large number of collections often provides scope for achieving sometimes significant memory-footprint savings by selecting the right collection, sizing it correctly, and potentially by allocating it lazily. These decisions are best made during design and development, but you can also use the Memory Analyzer tool to analyze your existing applications for potential memory-footprint optimization.

Resources

Learn

- "Debugging from dumps" (Chris Bailey, Andrew Johnson, and Kevin Grigorenko, developerWorks, March 2011): Learn how to generate dumps with Memory Analyzer and use them to examine the state of your application.
- "The Support Authority: Why the Memory Analyzer (with IBM Extensions) isn't just for memory leaks anymore" (Chris Bailey, Kevin Grigorenko, and Dr. Mahesh Rathi, developerWorks, March 2011): This article shows you how to use Memory Analyzer combined with the IBM Extensions for Memory Analyzer plug-in to examine the states of both WebSphere Application Server and your application.
- "*5 things you didn't know about* ... the Java Collections API, Part 1" (Ted Neward, developerWorks, April 2010): Read five tips for doing more with Collections. Get five more in Part 2.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer: Memory Analyzer brings the diagnostic capabilities of the Eclipse Memory Analyzer Tool (MAT) to the IBM Virtual Machines for Java.
- IBM Extensions for Memory Analyzer: The IBM Extensions for Memory Analyzer offer additional capabilities for debugging generic Java applications, and capabilities for debugging specific IBM software products.
- Eclipse Memory Analyzer Tool (MAT): MAT helps find memory leaks and identify high-memory-consumption issues.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- IBM on troubleshooting Java applications: Read this blog, written by Chris Bailey and his colleagues, for news and information on IBM tooling for troubleshooting your Java applications.
- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Chris Bailey

Chris Bailey is a member of the IBM Java Technology Center team at the Hursley Park Development Lab in the United Kingdom. As the technical architect for the IBM Java service and support organization, he is responsible for enabling users of the IBM SDK for Java to deliver successful application deployments. Chris is also involved in gathering and assessing new requirements, delivering new debugging capabilities and tools, making improvements in documentation, and improving the quality of the IBM SDK for Java.