Do you know your data size?

This article represents a list of web pages which can help one understand the **memory usage of Java objects** and **arrays** — along with examples. Please feel free to comment/suggest any other cool pages. Also, sorry for the typos.

The in-memory size of the object depends on the architecture, mainly on whether the VM is 32 or 64-bit. The actual VM implementation also matters.

- How to calculate memory usage of a Java object?: Very simplified explanation of how one could calculate a memory of any Java object. For example, lets say, you want to calculate the memory of a Java object which holds two int variables, one boolean variable, one **Long** object, and a **reference to other objects**. The memory would turn out to be following:
    - **8 bytes** for the **object header**
    - 2 x **4** = 8 bytes for two **int variables**
    - **1 byte** for a **boolean** variable
    - **8 bytes (object reference header?) + 8 bytes for long data type** = 16 bytes for **Long object**
    - **4 bytes** for **reference to some other object**
    - The total size of the above mentioned object will be 8 + 8 + 1 + 16 + 4 = 37 bytes + 3 bytes (for padding) = 40 bytes.

Memory usage of Java objects: general guide

On this page, we take a general look at how to calculate the memory usage of a Java object, or at least an *estimate* of its usage. (Note that using the Classmexer agent from this site— or VM insturmentation generally— you can query the size of a Java object from within your program.)

We'll generally be talking about the memory taken up on the **heap** by a given object under "normal circumstances". A couple of minor complications that we'll gloss over are that:

- in some cases, a JVM **may not put an object on the heap** at all: for example, a small thread-local object could in principle be held entirely on the stack or in registers and not "exist" strictly speaking as a Java heap object;
- the overall memory impact of an object can depend on its **current state**: for example, whether its synchronization lock is **contended**, or whether it is being garbage collected (such extra "system" data is not necessarily allocated on the Java heap, however).

On this page, we look at the memory usage of a Java object generally. On the next pages, we'll look specifically at the memory usage of Strings and related objects.

General formula for calculating memory usage

In general, the heap memory used by a Java object in Hotspot consists of:

- an **object header**, consisting of a few bytes of "housekeeping" information;
- memory for **primitive fields**, according to their size (see below);
- memory for **reference fields** (4 bytes each);

- **padding**: potentially a few "wasted" unused bytes after the object data, to make every object start at an address that is a convenient multiple of bytes and reduce the number of bits required to represent a pointer to an object.

Sizes of primitive types

In case you're not familiar with the byte size of the different Java primitive data types, here is the complete list:

<table>
<tr><th>Java type</th><th>Bytes required</th></tr>
<tr><td>boolean</td><td rowspan="2">1</td></tr>
<tr><td>byte</td></tr>
<tr><td>char</td><td rowspan="2">2</td></tr>
<tr><td>short</td></tr>
<tr><td>int</td><td rowspan="2">4</td></tr>
<tr><td>float</td></tr>
<tr><td>long</td><td rowspan="2">8</td></tr>
<tr><td>double</td></tr>
</table>

You may have expected a boolean to take up a single bit, or an eighth of a byte, especially if an object had 8 boolean fields. In practice, Hotspot (and, I believe, VMs generally) allocate a whole byte to each boolean[*].

* The reason is simply ease and efficiency of implementation: generally, we want to assign a "byte offset" to each field of a class and use a simple instruction to read/write an individual byte. It would be awkward if we had to cope with sub-byte offsets for certain fields, and it would require extra logic to read/write individual bits at a given position rather than just the whole byte each time the boolean was accessed.

Object overhead for "housekeeping" information

Instances of an object on the Java heap don't just take up memory for their actual fields. Inevitably, they also require some "housekeeping" information, such as recording an object's class, ID and status flags such as whether the object is currently reachable, currently synchronization-locked etc.

In Hotspot:

- a normal object requires **8 bytes** of "housekeeping" space;
- **arrays require 12 bytes** (the same as a normal object, plus 4 bytes for the array length).

Other JVMs probably have a similar object overhead.

Object size granularity

In Hotspot, every object occupies a number of bytes that is a **multiple of 8**. If the number of bytes required by an object for its header and fields is not a multiple 8, then you **round up to the next multiple of 8**.

This means, for example, that:

- a bare Object takes up 8 bytes;
- an instance of a class with a single boolean field takes up 16 bytes: 8 bytes of header, 1 byte for the boolean and 7 bytes of "padding" to make the size up to a multiple of 8;
- an instance with *eight* boolean fields will also take up 16 bytes: 8 for the header, 8 for the booleans; since this is already a multiple of 8, no padding is needed;
- an object with a two long fields, three int fields and a boolean will take up:
    - 8 bytes for the header;
    - 16 bytes for the 2 longs (8 each);
    - 12 bytes for the 3 ints (4 each);
    - 1 byte for the boolean;
    - a further 3 bytes of padding, to round the total up from 37 to 40, a multiple of 8.

- How to calculate memory usage of a Java array?:

Following on from our discussion of the memory usage of Java objects, on this page, we consider the special case of **arrays**. Recall that:

- in Java, an array is a **special type of object**;
- a **multi-dimensional** array is simple an **array of arrays**; for example, in a **two-dimensional array**, every **row is a separate array object**.

(To query the memory usage of an array from within a Java program, you can use the memory utility class of the Classmexer agent available from this site.)

Memory usage of a single-dimension array

A single-dimension array is a single object. As expected, the array has the usual **object header**. However, this object head is **12 bytes** to accommodate a **four-byte array length**. Then comes the actual array data which, as you might expect, consists of the number of elements multiplied by the number of bytes required for one element, depending on its type. The memory usage for one element is 4 bytes for an object reference; for a list of the memory usage of primitive types, see the page on Java data types. If the total memory usage of the array is not a multiple of 8 bytes, then the size is rounded up to the next mutlitple of 8 (just as for any other object).

Note that a **boolean array requires one byte per element**, even though each element actually only stores a single bit of useful information. (If you need to store a series of bits more compactly, see the BitSet class.)

Memory usage of a two-dimensional array

In a language such as C, a two-dimensional array (or indeed any multidimensional array) is essentially a one-dimensional array with judicious pointer manipulation. This is not the case in Java, where a multidimensional array is actually a set of **nested arrays**. This means that **every row of a two-dimensional array has the overhead of an object**, since it actually *is* a separate object!

For example, let's consider a 10x10 int array. Firstly, the "outer" array has its 12-byte object header followed by space for the 10 elements. Those elements are object references to the 10 arrays making up the rows. That comes to 12+4*10=52 bytes, which must then be rounded up to the next multiple of 8, giving 56. Then, each of the 10 rows has its own 12-byte object header, 4*10=40 bytes for the actual row of ints, and again, 4 bytes of padding to bring the total for that row to a multiple of 8. So in total, that gives 11*56=616 bytes. That's a bit bigger than if you'd just counted on 10*10*4=400 bytes for the hundred "raw" ints themselves.

Multidimensional arrays

For arrays of more than 2 dimensions, the above logic repeats: each row of the "outer" array is now an *array of references* to a further array, which contains the actual primitive data (or references if it is an object array).

The page presents examples on how to calculate size of a Java array object. For example, lets say a Java array consisting of 20 Integer objects. Following is the detail on the size:

- 12 bytes for array header object (**8 bytes for header** and **4 bytes for storing length** of the array)
- 20 x 16 bytes (object reference + an Integer object) = 320 bytes for Integer objects.
- The total size of the said Java array object = 12 + 320 bytes = 332 bytes + 4 bytes (**padding**) = 336 bytes.
- Memory usage of both Java Objects and Array: This article presents more examples on memory usage of objects types such as String.

Don't pay the price for hidden class fields

Recently, I helped design a Java server application that resembled an in-memory database. That is, we biased the design toward caching tons of data in memory to provide super-fast query performance.

Once we got the prototype running, we naturally decided to profile the data memory footprint after it had been parsed and loaded from disk. The unsatisfactory initial results, however, prompted me to search for explanations.

Note: You can download this article's source code from Resources.

The tool

Since Java purposefully hides many aspects of memory management, discovering how much memory your objects consume takes some work. You could use the `Runtime.freeMemory()` method to measure heap size differences before and after several objects have been allocated. Several articles, such as Ramchander Varadarajan's "Question of the Week No. 107" (Sun Microsystems, September 2000) and Tony Sintes's "Memory Matters" (*JavaWorld,* December 2001), detail that idea. Unfortunately, the former article's solution fails because the implementation employs a wrong `Runtime` method, while the latter article's solution has its own imperfections:

- A single call to `Runtime.freeMemory()` proves insufficient because a **JVM may decide to increase its current heap size at any time** (especially when it runs garbage collection). Unless the total heap size is already at the -Xmx maximum size, we should use `Runtime.totalMemory()-Runtime.freeMemory()` as the used heap size.
- Executing a single `Runtime.gc()` call **may not prove sufficiently aggressive for requesting garbage collection.** We could, for example, **request object finalizers to run as well**. And since `Runtime.gc()` is not documented to block until collection completes**, it is a good idea to wait until the perceived heap size stabiliz**es.
- If the profiled class creates any static data as part of its per-class class initialization (including static class and field initializers), the heap memory used for the first class instance may include that data. We should ignore heap space consumed by the first class instance.

Considering those problems, I present `Sizeof`, a tool with which I snoop at various Java core and application classes:

```
public class Sizeof
{
    public static void main (String [] args) throws Exception
    {
        // Warm up all classes/methods we will use
        runGC ();
        usedMemory ();
        // Array to keep strong references to allocated objects
        final int count = 100000;
        Object [] objects = new Object [count];

        long heap1 = 0;
        // Allocate count+1 objects, discard the first one
        for (int i = -1; i < count; ++ i)
        {
            Object object = null;

            // Instantiate your data here and assign it to object

            object = new Object ();
            //object = new Integer (i);
            //object = new Long (i);
            //object = new String ();
            //object = new byte [128][1]

            if (i >= 0)
                objects [i] = object;
            else
            {
                object = null; // Discard the warm up object
                runGC ();
                heap1 = usedMemory (); // Take a before heap snapshot
            }
        }
        runGC ();
        long heap2 = usedMemory (); // Take an after heap snapshot:

        final int size = Math.round (((float)(heap2 - heap1))/count);
        System.out.println ("'before' heap: " + heap1 +
                            ", 'after' heap: " + heap2);
        System.out.println ("heap delta: " + (heap2 - heap1) +
            ", {" + objects [0].getClass () + "} size = " + size + " bytes");
        for (int i = 0; i < count; ++ i) objects [i] = null;
        objects = null;
    }
    private static void runGC () throws Exception
    {
        // It helps to call Runtime.gc()
        // using several method calls:
        for (int r = 0; r < 4; ++ r) _runGC ();
    }
    private static void _runGC () throws Exception
    {
```

```
        long usedMem1 = usedMemory (), usedMem2 = Long.MAX_VALUE;
        for (int i = 0; (usedMem1 < usedMem2) && (i < 500); ++ i)
        {
            s_runtime.runFinalization ();
            s_runtime.gc ();
            Thread.currentThread ().yield ();

            usedMem2 = usedMem1;
            usedMem1 = usedMemory ();
        }
    }
    private static long usedMemory ()
    {
        return s_runtime.totalMemory () - s_runtime.freeMemory ();
    }

    private static final Runtime s_runtime = Runtime.getRuntime ();
} // End of class
```

`Sizeof`'s key methods are `runGC()` and `usedMemory()`. I use a `runGC()` wrapper method to call `_runGC()` several times because it appears to make the method more aggressive. (I am not sure why, but it's possible creating and destroying a method call-stack frame causes a change in the reachability root set and prompts the garbage collector to work harder. Moreover, consuming a large fraction of the heap space to create enough work for the garbage collector to kick in also helps. In general, it is hard to ensure everything is collected. The exact details depend on the JVM and garbage collection algorithm.)

Note carefully the places where I invoke `runGC()`. You can edit the code between the `heap1` and `heap2` declarations to instantiate anything of interest.

Also note how `Sizeof` prints the object size: the transitive closure of data required by all `count` class instances, divided by `count`. For most classes, the result will be memory consumed by a single class instance, including all of its owned fields. That memory footprint value differs from data provided by many commercial profilers that report shallow memory footprints (for example, if an object has an `int[]` field, its memory consumption will appear separately).

The results

Let's apply this simple tool to a few classes, then see if the results match our expectations.

Note: The following results are based on Sun's JDK 1.3.1 for Windows. Due to what is and is not guaranteed by the Java language and JVM specifications, you cannot apply these specific results to other platforms or other Java implementations.

java.lang.Object

Well, the root of all objects just had to be my first case. For `java.lang.Object`, I get:

```
'before' heap: 510696, 'after' heap: 1310696
heap delta: 800000, {class java.lang.Object} size = 8 bytes
```

So, a plain `Object` takes 8 bytes; of course, no one should expect the size to be 0, as every instance must carry around fields that support base operations like `equals()`, `hashCode()`, `wait()/notify()`, and so on.

java.lang.Integer

My colleagues and I frequently wrap native `ints` into `Integer` instances so we can store them in Java collections. How much does it cost us in memory?

```
'before' heap: 510696, 'after' heap: 2110696
heap delta: 1600000, {class java.lang.Integer} size = 16 bytes
```

The 16-byte result is a little worse than I expected because an `int` value can fit into just 4 extra bytes. Using an `Integer` costs me a 300 percent memory overhead compared to when I can store the value as a primitive type.

java.lang.Long

`Long` should take more memory than `Integer`, but it does not:

```
'before' heap: 510696, 'after' heap: 2110696
heap delta: 1600000, {class java.lang.Long} size = 16 bytes
```

Clearly, actual object size on the heap is subject to low-level memory alignment done by a particular JVM implementation for a particular CPU type. It looks like a `Long` is 8 bytes of `Object` overhead, plus 8 bytes more for the actual long value. In contrast, `Integer` had an unused 4-byte hole, most likely because the JVM I use forces object alignment on an 8-byte word boundary.

Arrays

Playing with primitive type arrays proves instructive, partly to discover any hidden overhead and partly to justify another popular trick: wrapping primitive values in a size-1 array to use them as objects. By modifying `Sizeof.main()` to have a loop that increments the created array length on every iteration, I get for `int` arrays:

```
length: 0, {class [I} size = 16 bytes
length: 1, {class [I} size = 16 bytes
length: 2, {class [I} size = 24 bytes
length: 3, {class [I} size = 24 bytes
length: 4, {class [I} size = 32 bytes
length: 5, {class [I} size = 32 bytes
length: 6, {class [I} size = 40 bytes
length: 7, {class [I} size = 40 bytes
length: 8, {class [I} size = 48 bytes
length: 9, {class [I} size = 48 bytes
length: 10, {class [I} size = 56 bytes
```

and for `char` arrays:

```
length: 0, {class [C} size = 16 bytes
length: 1, {class [C} size = 16 bytes
length: 2, {class [C} size = 16 bytes
length: 3, {class [C} size = 24 bytes
length: 4, {class [C} size = 24 bytes
length: 5, {class [C} size = 24 bytes
length: 6, {class [C} size = 24 bytes
length: 7, {class [C} size = 32 bytes
```

```
length: 8, {class [C} size = 32 bytes
length: 9, {class [C} size = 32 bytes
length: 10, {class [C} size = 32 bytes
```

Above, the evidence of 8-byte alignment pops up again. Also, in addition to the inevitable `Object` 8-byte overhead, a primitive array adds another 8 bytes (out of which at least 4 bytes support the `length` field). And using `int[1]` appears to not offer any memory advantages over an `Integer` instance, except maybe as a mutable version of the same data.

Multidimensional arrays

Multidimensional arrays offer another surprise. Developers commonly employ constructs like `int[dim1][dim2]` in numerical and scientific computing. In an `int[dim1][dim2]` array instance, every nested `int[dim2]` array is an `Object` in its own right. Each adds the usual 16-byte array overhead. When I don't need a triangular or ragged array, that represents pure overhead. The impact grows when array dimensions greatly differ. For example, a `int[128][2]` instance takes 3,600 bytes. Compared to the 1,040 bytes an `int[256]` instance uses (which has the same capacity), 3,600 bytes represent a 246 percent overhead. In the extreme case of `byte[256][1]`, the overhead factor is almost 19! Compare that to the C/C++ situation in which the same syntax does not add any storage overhead.

java.lang.String

Let's try an empty `String`, first constructed as `new String()`:

```
'before' heap: 510696, 'after' heap: 4510696
heap delta: 4000000, {class java.lang.String} size = 40 bytes
```

The result proves quite depressing. An empty `String` takes 40 bytes—enough memory to fit 20 Java characters.

Before I try `String`s with content, I need a helper method to create `String`s guaranteed not to get interned. Merely using literals as in:

```
    object = "string with 20 chars";
```

will not work because all such object handles will end up pointing to the same `String` instance. The language specification dictates such behavior (see also the `java.lang.String.intern()` method). Therefore, to continue our memory snooping, try:

```
    public static String createString (final int length)
    {
        char [] result = new char [length];
        for (int i = 0; i < length; ++ i) result [i] = (char) i;

        return new String (result);
    }
```

After arming myself with this `String` creator method, I get the following results:

```
length: 0, {class java.lang.String} size = 40 bytes
length: 1, {class java.lang.String} size = 40 bytes
length: 2, {class java.lang.String} size = 40 bytes
```

```
length: 3, {class java.lang.String} size = 48 bytes
length: 4, {class java.lang.String} size = 48 bytes
length: 5, {class java.lang.String} size = 48 bytes
length: 6, {class java.lang.String} size = 48 bytes
length: 7, {class java.lang.String} size = 56 bytes
length: 8, {class java.lang.String} size = 56 bytes
length: 9, {class java.lang.String} size = 56 bytes
length: 10, {class java.lang.String} size = 56 bytes
```

The results clearly show that a `String`'s memory growth tracks its internal `char` array's growth. However, the `String` class adds another 24 bytes of overhead. For a nonempty `String` of size 10 characters or less, the added overhead cost relative to useful payload (2 bytes for each `char` plus 4 bytes for the length), ranges from 100 to 400 percent.

Of course, the penalty depends on your application's data distribution. Somehow I suspected that 10 characters represents the typical `String` length for a variety of applications. To get a concrete data point, I instrumented the SwingSet2 demo (by modifying the `String` class implementation directly) that came with JDK 1.3.x to track the lengths of the `String`s it creates. After a few minutes playing with the demo, a data dump showed that about 180,000 `Strings` were instantiated. Sorting them into size buckets confirmed my expectations:

```
[0-10]:  96481
[10-20]: 27279
[20-30]: 31949
[30-40]: 7917
[40-50]: 7344
[50-60]: 3545
[60-70]: 1581
[70-80]: 1247
[80-90]: 874

...
```

That's right, more than 50 percent of all `String` lengths fell into the 0-10 bucket, the very hot spot of `String` class inefficiency!

In reality, `String`s can consume even more memory than their lengths suggest: `String`s generated out of `StringBuffer`s (either explicitly or via the '+' concatenation operator) likely have `char` arrays with lengths larger than the reported `String` lengths because `StringBuffer`s typically start with a capacity of 16, then double it on `append()` operations. So, for example, `createString(1) + ' '` ends up with a `char`array of size 16, not 2.

What do we do?

"This is all very well, but we don't have any choice but to use `String`s and other types provided by Java, do we?" I hear you ask. Let's find out.

Wrapper classes

Wrapper classes like `java.lang.Integer` seem a bad choice for storing large data amounts in memory. If you strive to be memory-economic, avoid them altogether. Rolling your own vector class for primitive `int`s isn't difficult. Of course, it would be great if the Java core API already contained such libraries. Perhaps the situation will improve when Java has generic types.

Multidimensional arrays

For large data structures built with multidimensional arrays, you can oftentimes reduce the extra dimension overhead by an easy indexing change: convert every `int[dim1][dim2]` instance to an `int[dim1*dim2]` instance and change all expressions like `a[i][j]` to `a[i*dim1 + j]`. Of course, you pay a price from the lack of index-range checking on `dim1` dimension (which also boosts performance).

java.lang.String

You can try a few simple tricks to reduce your application's `String` static memory size.

First, you can try one common technique when an application loads and caches many `String`s from a data file or a network connection, and the `String` value range proves limited. For example, if you want to parse an XML file in which you frequently encounter a certain attribute, but the attribute is limited to just two possible values. Your goal: filter all `String`s through a hash map and reduce all equal but distinct `String`s to identical object references:

```
    public String internString (String s)
    {
        if (s == null) return null;

        String is = (String) m_strings.get (s);
        if (is != null)
            return is;
        else
        {
            m_strings.put (s, s);
            return s;
        }
    }

    private Map m_strings = new HashMap ();
```

When applicable, that trick can decrease your static memory requirements by hundreds of percent. An experienced reader may observe that the trick duplicates `java.lang.String.intern()`'s functionality. Numerous reasons exist to avoid the `String.intern()` method. One is that few modern JVMs can intern large amounts of data.

What if your `String`s are all different? For the second trick, recollect that for small `String`s the underlying `char` array takes half the memory occupied by the `String` that wraps it. Thus, when my application caches many distinct `String` values, I can just keep the arrays in memory and convert them to `String`s as needed. That works well if each such `String` then serves as a transient, quickly discarded object. A simple experiment with caching 90,000 words taken from a sample dictionary file shows that this data takes about 5.6 MB in `String` form and only 3.4 MB in `char[]` form, a 65 percent reduction.

The second trick contains one obvious disadvantage: you cannot convert a `char[]` back to a `String` through a constructor that would take ownership of the array without cloning it. Why? Because the entire public `String` API ensures that every `String` is immutable, so every `String` constructor defensively clones input data passed through its parameters.

Still, you can try a third trick when the cost of converting from `char` arrays to `String`s proves too high. The trick exploits `java.lang.String.substr()`'s ability to avoid data copying: the method implementation exploits `String` immutability and creates a shallow `String` object that shares the `char` content array with the original `String` but has its internal start and end indices adjusted correspondingly. To make an example, `new String("smiles").substring(1,5)` is a `String` configured to start at index 1 and end at index 4 within a `char` buffer "smiles" shared by reference with the originally constructed `String`. You can exploit that fact as follows: given a large `String` set, you can merge its `char` content into one large `char` array, create a `String` out of it, and recreate the original `String`s as sub`Strings` of this master `String`, as the following method illustrates:

```
    public static String [] compactStrings (String [] strings)
    {
        String [] result = new String [strings.length];
        int offset = 0;

        for (int i = 0; i < strings.length; ++ i)
            offset += strings [i].length ();

        // Can't use StringBuffer due to how it manages capacity
        char [] allchars = new char [offset];

        offset = 0;
        for (int i = 0; i < strings.length; ++ i)
        {
            strings [i].getChars (0, strings [i].length (), allchars,
offset);
            offset += strings [i].length ();
        }

        String allstrings = new String (allchars);

        offset = 0;
        for (int i = 0; i < strings.length; ++ i)
            result [i] = allstrings.substring (offset,
                                              offset += strings [i].length
());

        return result;
    }
```

The above method returns a new set of `String`s equivalent to the input set but more compact in memory. Recollect from earlier measurements that every `char[]` adds 16 bytes of overhead; effectively removed by this method. The savings could be significant when cached data comprises mostly short `String`s. When you apply this trick to the same 90,000-word dictionary mentioned above, the memory size drops from 5.6 MB to 4.2 MB, a 30 percent reduction. (An astute reader will observe in that particular example the `String`s tend to share many prefixes and the `compactString()` method could be further optimized to reduce the merged `char` array's size.)

As a side effect, `compactString()` also removes `StringBuffer`-related inefficiencies mentioned earlier.

Is it worth the effort?

To many, the techniques I presented may seem like micro-optimizations not worth the time it takes to implement them. However, remember the applications I had in mind: server-side applications that cache massive amounts of data in memory to achieve performance impossible when data comes from a disk or database. Several hundred megabytes of cached data represents a noticeable fraction of maximum heap sizes of today's 32-bit JVMs. Shaving 30 percent or more off is nothing to scoff at; it could push an application's scalability limits quite noticeably. Of course, these tricks cannot substitute for beginning with well-designed data structures and profiling your application to determine its actual hot spots. In any case, you're now more aware of how much memory your objects consume.

Vladimir Roubtsov has programmed in a variety of languages for more than 12 years, including Java since 1995. Currently, he develops enterprise software as a senior developer for Trilogy in Austin, Texas. When coding for fun, Vladimir develops software tools based on Java byte code or source code instrumentation.