**JVM Memory management**

The JVM manages memory for us. This removes the memory management burden from the developers, so **we don't need to manipulate object pointers manually**, which is proven to be time consuming and error-prone.

Under the hood, the JVM incorporates a lot of nifty tricks to optimize the memory management process. **One trick is the use of *Compressed Pointers****,*which we're going evaluate in this article. First off, let's see how the JVM represents objects at runtime.

## **2. Runtime Object Representation**

The HotSpot JVM uses a data structure called [oops](https://www.baeldung.com/java-memory-layout) or Ordinary Object Pointers to represent objects. These oops are equivalent to native C pointers. **The *[instanceOop](https://github.com/openjdk/jdk15/blob/master/src/hotspot/share/oops/instanceOop.hpp)*s are a special kind of oop that represents the object instances in Java**. Moreover, the JVM also supports a handful of other oops that are kept in the [OpenJDK source tree](http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/oops/).

Let's see how the JVM lays out instanceOops in memory.

### ****2.1. Object Memory Layout****

The memory layout of an instanceOop is simple: it's just the object header immediately followed by zero or more references to instance fields.

The JVM representation of an object header consists of:

* **One mark word** serves many purposes such as Biased Locking, Identity Hash Values, and GC. It's not an oop, but for historical reasons, it resides in the [**OpenJDK's oop**](http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/oops/markOop.hpp) source tree. Also, the mark word state only contains a ***[uintptr\_t](https://github.com/openjdk/jdk15/blob/e208d9aa1f185c11734a07db399bab0be77ef15f/src/hotspot/share/oops/markWord.hpp" \l "L96)***, therefore, **its size varies between 4 and 8 bytes in 32-bit and 64-bit architectures, respectively**
* **One, possibly compressed, Klass word**, which represents a pointer to class metadata. Before Java 7, they were pointing to the [**Permanent Generation**](https://www.baeldung.com/native-memory-tracking-in-jvm), but from the Java 8 onward, they are pointing to the ***[Metaspace](https://www.baeldung.com/native-memory-tracking-in-jvm)***
* **A 32-bit gap**to enforce object alignment. This makes the layout more hardware friendly, as we will see later

**Immediately after the header, there are to be zero or more references to instance fields.**In this case, a word is a native machine word, so 32-bit on legacy 32-bit machines and 64-bit on more modern systems.

The object header of arrays, in addition to mark and klass words, contains a [32-bit word](https://github.com/openjdk/jdk15/blob/e208d9aa1f185c11734a07db399bab0be77ef15f/src/hotspot/share/oops/arrayOop.hpp#L35) to represent its length.

### ****2.2.****Anatomy of Waste

Suppose we're going to switch from a legacy 32-bit architecture to a more modern 64-bit machine. At first, we may expect to get an immediate performance boost. However, that's not always the case when the JVM is involved.

**The main culprit for this possible performance degradation is 64-bit object references.** 64-bit references take up twice the space of 32-bit references, so this leads to more memory consumption in general and more frequent GC cycles. The more time dedicated to GC cycles, the fewer CPU execution slices for our application threads.

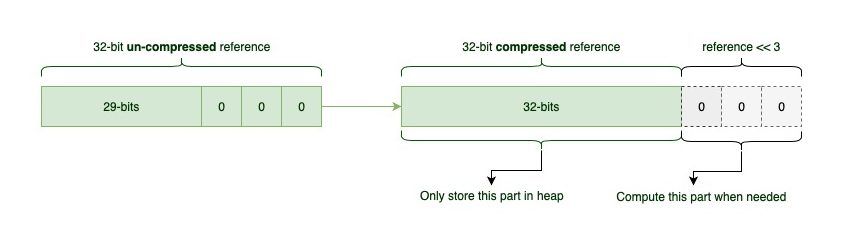
So, should we switch back and use those 32-bit architectures again? Even if this were an option, we couldn't have more than 4 GB of heap space in 32-bit process spaces without a bit more work.

## **3. Compressed OOPs**

As it turns out, the JVM can avoid wasting memory by compressing the object pointers or oops, so we can have the best of both worlds: **allowing more than 4 GB of heap space with 32-bit references in 64-bit machines!**

### ****3.1.****Basic Optimization

As we saw earlier, the JVM adds padding to the objects so that their size is a multiple of 8 bytes.**With these paddings, the last three bits in oops are always zero.**This is because numbers that are a multiple of 8 always end in 000 in binary.



Since the JVM already knows that the last three bits are always zero, there's no point in storing those insignificant zeros in the heap. Instead, it assumes they are there and stores 3 other more significant bits that we couldn't fit into 32-bits previously. Now, we have a 32-bit address with 3 right-shifted zeros, so we're compressing a 35-bit pointer into a 32-bit one. **This means that we can use up to 32 GB –  232+3=235=32 GB – of heap space without using 64-bit references.**

In order to make this optimization work, when the JVM needs to find an object in memory **it shifts the pointer to left by 3 bits** (basically adds those 3-zeros back on to the end). On the other hand, when loading a pointer to the heap, the JVM shifts the pointer to right by 3 bits to discard those previously added zeros. **Basically, the JVM performs a little bit more computation to save some space.**Luckily, bit shifting is a really trivial operation for most CPUs.

To enable oop compression, we can use the -XX:+UseCompressedOops tuning flag. The oop compression is the default behavior from Java 7 onwards whenever the maximum heap size is less than 32 GB.**When the maximum heap size is more than 32 GB, the JVM will automatically switch off the oop compression.** So memory utilization beyond a 32 Gb heap size needs to be managed differently.

**3.2. Beyond 32 GB**

It's also possible to use compressed pointers when Java heap sizes are greater than 32GB. **Although the default object alignment is 8 bytes, this value is configurable using the *-XX:ObjectAlignmentInBytes*tuning flag. The specified value should be a power of two and must be within the range of 8 and 256**.

We can calculate the maximum possible heap size with compressed pointers as follows:

|  |  |
| --- | --- |
|  | 4 GB \* ObjectAlignmentInBytes |

For example, when the object alignment is 16 bytes, we can use up to 64 GB of heap space with compressed pointers.

Please note that as the alignment value increases, the unused space between objects might also increase. As a result, we may not realize any benefits from using compressed pointers with large Java heap sizes.

### ****3.3.****Futuristic GCs

[ZGC](https://www.baeldung.com/jvm-zgc-garbage-collector), a new addition in [Java 11](https://openjdk.java.net/jeps/333), was an experimental and scalable low-latency garbage collector.

It can handle different ranges of heap sizes while keeping the GC pauses under 10 milliseconds. Since ZGC needs to use [64-bit colored pointers](https://youtu.be/kF_r3GE3zOo?t=643), **it does not support compressed references**. So, using an ultra-low latency GC like ZGC has to be weighed against using more memory.

As of Java 15, ZGC supports the compressed class pointers but still lacks the support for Compressed OOPs.

All new GC algorithms, however, won't trade off memory for being low-latency. For instance, [Shenandoah GC](https://openjdk.java.net/jeps/379) supports compressed references in addition to being a GC with low pause times.