**Slide 2-3: So, let’s forget for a while about these snippents, we will turn to them in 2nd part of talk,**

**Slide 5:**

Let me give you brief overview of JVM components, which will be covered in more details on next slides.

**Stack** is used for storing primitives while heap keep objects. These areas are interconnected, because (we’ll talk a bit later about it) when objects are stored in heap, stack keeps references to them.

**Heap** is main area of JVM memory and is used for storing all instances of objects required for application **except instances of classes themselves. Explain that everything is object including class.**

**String pool** is special heap region where Strings are kept by JVM because due to Strings immutability JVM has ability to optimize amount of memory allocated to similar Strings. For example, if we have two String variables **a** and **b** and they both have value “Hello world”, then in fact JVM will keep only one object **“Hello world”** in String pool with two references pointing to it. This process is called interning and handled by **String.intern()** method.

**Metaspace** iskeeps objects of classes, e.g. while heap stores instances of objects, metaspace keeps object of classes(\*.class) which are used for creating new objects.

**Garbage collector** is component which is responsible for cleaning up objects which are not used anymore.

**Slide 6:**

Now it’s time to talk about how Stack and Heap work together.

**For example, we have following piece of code inside some method.**

Just for reference, do not give all these details during talk

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Stack Memory** | **Heap Space** |
| Order | It's accessed using Last-in First-out (LIFO) memory allocation system | Unlike Stack, where primitives can be only accessed from current method, objects in Heap can be accessed by reference from anywhere in the application.  Here we step into different garbage collectors’ algorithms  For better stability of the application, choosing of right Garbage Collection algorithm is critical. |
| Life | Stack memory only exists as long as the current method is running  The memory allocated of the heap remains until the program finishes or the memory is freed, whereas the memory allocated of the stack remains until the function returns. That’s the reason, why stack does not need garbage collection if compared with heap memory. | Heap space exists as long as the application runs |
| Efficiency | Stack is comparatively faster than heap because the data on a stack can only be accessed by the currently running thread, an­d synchronization is not necessary because each thread has its own stack, which is why stack memory is faster than heap memory. | Slower to allocate when compared to stack:   * We need to synchronize access to memory, because unlike in Stack, multiple objects can try to turn to similar region of memory; * Objects in heap are heavier than in stack; |
| Allocation/Deallocation | This Memory is automatically allocated and deallocated when a method is called and returned, respectively | Heap space is allocated when new objects are created and deallocated by Gargabe Collector when they're no longer referenced |

**Slide 7:**

[An object is considered garbage when it can no longer be reached from any pointer in the running program](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html). The most straightforward garbage collection algorithms iterate over every reachable object. Any objects left over are considered garbage. The time this approach takes is proportional to the number of live objects, which is prohibitive for large applications maintaining lots of live data.

That’s the reason for weak generational hypothesis, which states that most objects survive for only a short period of time.

According to this concept, Heap is separated into two generations: Young and Old.

Young generation – is a region, where all new objects are allocated.

Old generation – is a region, where long-living objects are placed in case if they were not cleaned by several garbage collection cycles.

**Slide 8:**

**Metaspace**

| **PermGen** | **MetaSpace** |
| --- | --- |
| It is removed from java 8. | It is introduced in Java 8. |
| PermGen always has a fixed maximum size. | Metaspace by default auto increases its size depending on the underlying OS. |
| Contiguous Java Heap Memory. | Native Memory (provided by underlying OS). |
| [GC can happen in PermGen and it is counted as major/full GC.](http://www.openkb.info/2014/07/garbage-collection-in-permgen.html) | Memory Management is a complex process which is, simplified, about:  Identifying the objects which are garbage, which is actually a process of determining which objects are reachable (read: not garbage) and consider everything not encountered to be garbage  Enqueuing object references to reference queues and/or trigger finalization, if necessary  Reclaiming memory formerly occupied by garbage, which might also be the other way around: sometimes the alive objects are moved to a different memory space instead  So for a memory space not consisting of Java objects, the first two points usually make not much sense which is what your question seems to be about. Algorithms addressing the first two points usually process the Java heap (defined as space containing ordinary Java object instances and similar structured data) only.  [The statement you have linked, saying “Metaspace is GCed” seems to address mainly the third point.](https://stackoverflow.com/questions/26462136/does-gc-collects-garbage-from-metaspace) It is about the fact that memory within the Metaspace might get reclaimed if not needed anymore. This does not imply that it requires a traversal of live references within the Metaspace or something similar. Obviously, class metadata are obsolete when their associated Class and ClassLoader have become unreachable, which are both ordinary (well, almost) objects living on the Java heap.  So, when the Metaspace size reaches a limit, a garbage collection will be triggered, but regarding the first two bullet above, it will not process the Metaspace as it is not the Metaspace which can tell you whether a Class has become unused. It will be an ordinary garbage collection, but it will be a “Full GC” or whatever term the currently used GC algorithm has for the mode that includes collecting garbage within the memory segment (aka “generation”) which contains classes and class loaders.  Once Class and ClassLoader heap instances have been collected, their associated Metaspace data can be reclaimed as well during the cleanup. |

**Metaspace** is a space for keeping loaded class metadata. This space is separated from the main memory heap.

This native memory region grows automatically by default (up to what the underlying OS provides).

With this improvement in Java 8+ version JVM reduces the chance to get **OutOfMemory** error.

We also have flags to tune this part of memory:

**MetaspaceSize** and **MaxMetaspaceSize** – we can set the **Metaspace** upper bounds.

**MinMetaspaceFreeRatio** – is the minimum percentage of class metadata capacity free after garbage collection

**MaxMetaspaceFreeRatio** – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

**“pass by value concept”:** Actually, both tests from the beginning of light talk will pass, but there is significant difference: example 1: we make changes, and they are applied; example 2: we make changes and they are not applied. This happens due to difference in how JVM work with primitives and objects;