- «concurrent mode failure» in GC logs - can cause several seconds “stop the world” pauses even on relatively small heaps (600mb). Happens because there is no space in old generation to promote objects from young generation. But in that case defragmentation doesn’t maked
- “promotion failed” in GC logs means that old generation is too fragmented. Can cause huge pauses (28 seconds for 700mb heap). GC stops the world and makes full GC and defragmentation
- CMSInitiatingOccupancyFraction and UseCMSInitiatingOccupancyOnly
- ConcGCThreads
- CMSPermGenSweepingEnabled, CMSInitiatingPermOccupancyFraction and CMSClassUnloadingEnabled
- G1 has 3 kind of GC
    - Young GC
    - Normal G1 cycle: collect garbage and detect old generation regions which are consist mostly from garbage
    - Mixed G1 cycle: same as normal but also compact few old generation regions market as “mostly garbage”
    - Full GC
- G1 switch to full GC in case of
    - Concurrent mode failure: G1 start marking cycle, but old generation fills up before cycle is complete
    - Promotion failure: old generation fills up during mixed GC
    - Evacuation failure: both survivor space and old generation fills up during young GC
    - Humongous allocation failure: application allocate very large objects
- There usually one normal (concurrent) GC and 8mixed GC. Next concurrent GC can not be started until all segments marked as “garbage” are cleaned up. Segment marked as garbage if it has more than 35% of garbage. After that each next mixed GC process ⅛ part of garbage segments. It can be tuned by -XX:G1MixedGCCountTarget
- Young generation is divided into two survival spaces(0 and 1) and eden
- During first GC objects moved from eden into survival 0, during next GC objects from eden and survival 0 moved into survival 1. At that point eden and survival 0 spaces are absolutely empty. During next GC objects from eden and survival 1 all moved into survival 0. Objects promoted to old generation if there are no space left in target survival space or if objects “survived” during some number of GC(tenuring threshold)
- The share of young generation allocated for survival space can significantly affect GC behavior. By default 10% of young generation is allocated for survival space
- Tenuring threshold is automatically calculated by GC. Something between 7 and 15
- TLAB - thread local allocation buffer. JVM has TLAB for every application thread and by default every object is allocated in TLAB. TLAB is a segment of eden. Once TLAB is full JVM allocates new TLAB for a thread.
- -XX:TLABSize=n default TLAB size, -XX:-ResizeTLAB -disable TLAB resizing. By default TLAB size calculated dynamically based on the eden size

- “Refill waste” is 1% of TLAB size by default(+XX:TLABWastePercent) If object can not be allocated at TLAB and object size bigger than refill waste than it will be allocated at global heap. If it smaller than new TLAB will be allocated. Every time new object allocated at global heap instead of TLAB, refill waste is increased by -XX:TLABWasteIncrement. It also cause TLAB size increase
- If object so big so it can not be allocated in eden it will be allocated at old generation
- Java out of memory exception
    - Out of native memory
    - Out of perm gen memory
        - Because of too many classes
        - Because when new version of app deployed old class loader is not unloaded
    - Out of heap
        - Because live dataset is too big
        - Because of memory leaks
            - mat can compare two heap dumps
            - -XX:HeapDumpOnOutOfMemoryError
    - GC overhead limit is reached
- Overhead of an Java Object is 8byte for 32-bit JVM and 16 byte for 64-bit JVM
- Array header is 16 byte for 32-bit JVM and 24 byte in 64-bit JVM with big heap(more than 32GB)
- Object sizes always 8-byte aligned
- Use lazy initialization to save memory
- Use canonical objects where possible
- Use string interning. Use Eclipse Memory Analyzer to identify strings which should be interned
- Hash table used for string interning can hold only 60k string. After 30k strings hash collisions are likely to happen -XX:StringTableSize=N
- GC time depends much more on live dataset size than on heap size
- Object pooling is unfriendly to GC. Java Beans is build around the notion of object pool
- Soft references essentially is one big LRU pool of objects
- Performance of WeakHashMap is unpredictable, because on every access WeakHashMap have to process weak reference queue
- You should avoid using Finalizers because:
    - Of GC impact (you need at least two GC cycles to free up object with finalizer
    - Possible memory leak: If in finalizer method you will create another hard reference on cleaner object it will never be freed
- Use weak reference or PhantomReference instead of finalizers
- jcmd VM.native_memory baseline and jcmd VM.native_memory summary.diff compare memory usage
- -XX:+UseLargePages enable huge pages support in JVM. Do not use this flag if transparent huge pages are enabled. It works only with traditional huge pages
- Pointer compression is enabled if heap size less than 32 GB
- ForkJoinPool

- In java 8 common ForkJoinPool has been introduced.
  -Djava.util.concurrent.ForkJoinPool.common.parallelism=N
- Amdahl's law
- JRE неявно обеспечивает синхронизацию доступа к volatile переменным. Но это касается только операций чтения и записи. К примеру инкремент volatile переменной уже не является потоко-безопасным
- False sharing
- Biased locking enabled by default, can try -XX:-UseBiasedLocking
- Jconsole
- Java use UTF-16 encoding internally
- JVM Flags
  - AggressiveOpts
  - AutoFill
  - AutoBoxCacheMax
  - OptimizeStringConcat
  - PrintCompilation
  - InitialCodeCacheSize
  - -XX:+PrintFlagsFinal
  - MaxFreqInLineSize
  - MaxInlineSize
  - DoEscapeAnalysis
  - +PrintGC
  - +PrintGCDetails
  - +PrintGCTimeStamps
  - +PrintGCDateStamps
  - -XX:+PrintAdaptiveSizePolicy
  - -XX:PrintNMStatistics
  - -XX:NativeMemoryTracking=detail you can get into about native memory allocation at any time with jcmd process_id VM.native_memory summary
  - -XX:+PrintStringTableStatistics
  - -XX:+PrintTenuringDistribution - логгировать информацию по статистике выживания объектов
  - -XX:+PrintTLAB - monitor TLAB allocation
  - -XX:MaxGCPauseMills=N. Applied to both old and young generations. Too small value cause heap size to be small and cause a lot of GC cycles
- Tools for Java performance analysis:
  - JFR(java flight recorder)
  - Java Mission Control
  - Oracle Solaris analyzer studio,
  - jstack
  - jcmd
  - NetBeans profiler
  - jstat
  - jinfo
  - jvisualvm
  - eclipse memory analyzer,

- C1/C2 JIT compillers,
- 5 level of compilation
- Use GC Histogram to parse GC log and draw charts, jstat
- If more than one JVM run on the same server, than number of GC threads needs to be adopted. Every JVM assumes that it can consume all CPU available at server.
- TODO: read about class loaders and information stored in metaspace.
- jmap -permstat, -clstatsq
- Set heap size, so after full GC only 30% of the heap should be in use. Young generation- 30% of the total heap size
- During Stop the world GC, GC will drive CPU usage to 100%
- CMS garbage collector: stop the world during minor GC, scan old generation in background. No defragmentation in background. If no enough CPU available for background GC, it's switching to serial GC(stop the world GC of old generation)
- G1 is like CMS GC, but split memory into multiple regions(multiple young generations and multiple old generation). Minor GC are still stop the world, old generation GC are in background. Less suffering from old generation fragmentation because can copy one old generation region into another one.
-