Java Performance

Profile

* JVM events categories into 2 groups
  + Instant events
    - Onetime events that have a time stamp and event data

Examples

* + - * Exception events
      * Class events
      * Object allocation events
  + Duration events
    - Provide start and end time of an activity
      * Example
      * Garbage Collection
      * Monitor wait (how long a thread waited on an object)
      * Monitor Contended (how long was waiting in lock to be released)

Profiling Activities

* CPU Profiling
* Memory Profiling
* Thread Profiling
* I/O Profiling

CPU Profiling

* Finds what methods run the most frequently and use most CPU time
* Hot Methods
* CPU Profiling Modes:
  + Sampling (how much time spent on method)
  + Instrumentation
    - modifying application bytecode to insert instrumentation code
    - Bytecode instrumentation can alter performance characteristics

Memory profiling

* Concerned with understanding what objects are using up memory and how memory is freed
* Monitor memory usage of class objects
* Find out what objects are growing and shrinking in size
* Find where memory allocations are taking place
* Garbage Collection
  + Number and types of garbage collection events
  + Length of garbage collection pauses
  + How much memory is free by garbage collection

Thread profiling

* Concerned with understanding what states threads are in and why
* See the concurrency level of thread
* Find out how much time threads spends in the difference states
* Find and analyze cases of high lock contention

Java Profile Software

* JProfile
* YouKit Java Profiler
* Java VisualVM
* NetBeans Profiler
* Java Flight Recorder

JProfile

Features:

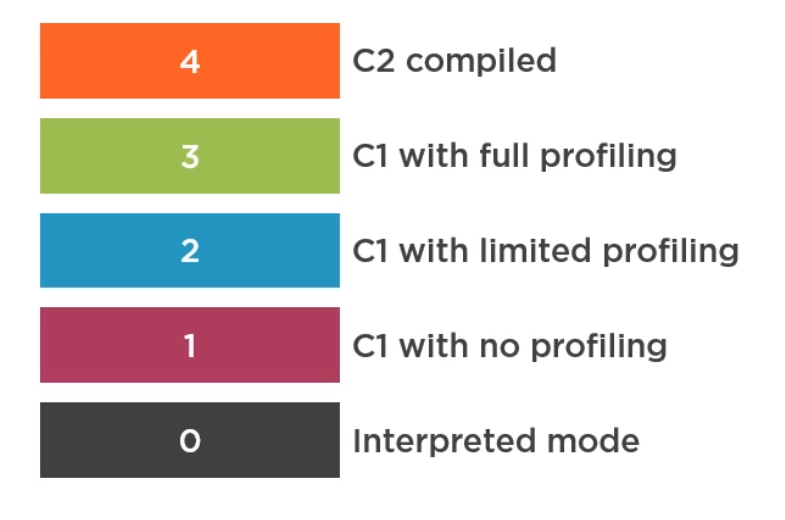
* CPU Profiling (instrumented and sampling mode)
* Memory profiling
* Thread profiling
* Database Query Tuning
* IDE integration
* Cross JVM Tracking

*JIT*

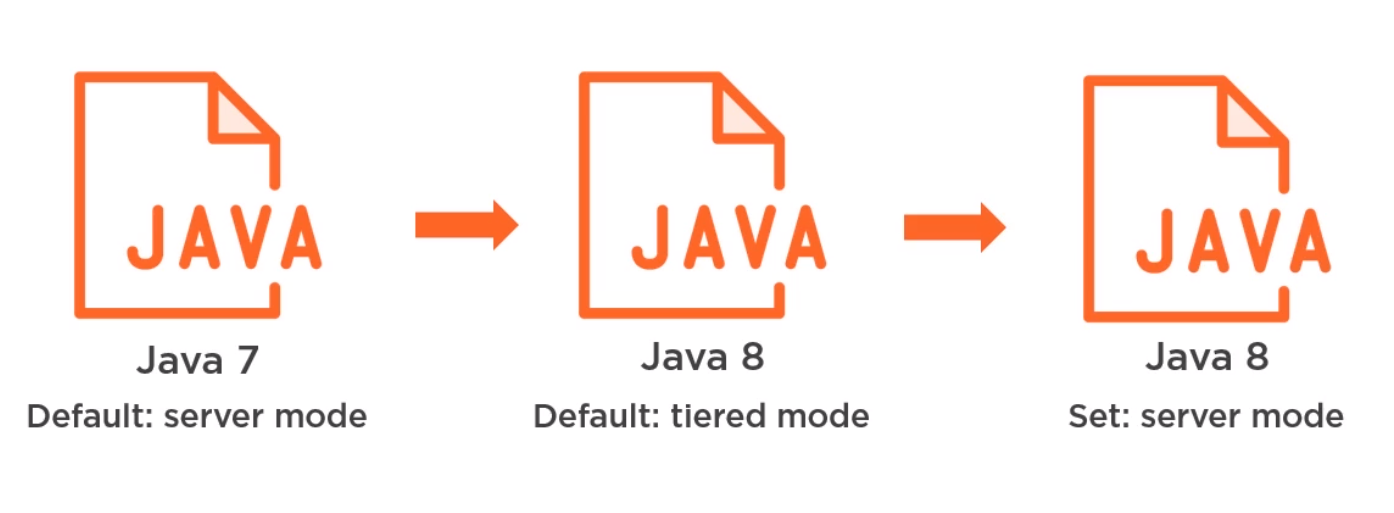
* *There are two types of programming languages* 
  + *compiled languages*
  + *interpreted languages.*
* *Compiled languages*
  + *like C++ and Go take the program code and transform it into files containing the instructions to execute in binary or assembly form.*
  + *This process of transformation is called compilation.*
  + *The compiler produces binaries that target a specific CPU architecture and are optimized to run efficiently on that CPU architecture.*
* *interpreted languages.* 
  + *Instead of precompiling the program code into binaries, interpreted languages have an interpreter that translates each line of program code into binary code as that line is executed.*
  + *The benefit of this approach is portability.*
  + *You can run the same program code on many different CPU architectures without having to do any ahead-of-time compilation that targets a CPU.*
  + *compiled languages have a performance advantage over interpreted languages.*
* *There's a third class of languages.* 
  + *This class covers languages that are compiled to a portable intermediate language,*
  + *usually known as bytecode, and then interpreted.*
  + *This fixes a couple of issues.* 
    - *First, you now have a compiler that can-do things like type checking and code optimizations.*
    - *Second, the bytecode is compiled only once, and because it's close to machine code, the effort to go from bytecode to machine code is much lower.*
    - *third, the bytecode is portable between different CPU architectures. This third style is the approach Java takes. The bytecode is stored in .class files, and the JVM interprets and executes it.*
* *However, the performance of interpreted bytecode still lags the performance of precompiled binary code.*
* *the JIT compiler was introduced and added to Sun's Hotspot JVM.*
* *The function of the JIT compiler is that during runtime,* 
  + *it identifies the parts of the application where the most time is spent executing bytecode, commonly known as hotspots, and then compiles the bytecode dynamically into machine code so that the program can run faster.*
* *The code is compiled just before execution, hence the name just-in-time, and is cached for future runs in the code cache.*

***JIT Compilation Modes***

* *In Hotspot JVM, there are two JIT compilers,* 
  + *compiler 1, commonly known as the C1 compiler, or the client compiler,*
  + *compiler 2, commonly known as the C2 compiler, or server compiler.*
* *C1 Compiler*
  + *The C1 compiler is intended to be used by client applications,*
  + *it's optimized for startup performance. This means that it tries to find hot methods early on, produces some basic, relatively un intrusive optimizations, and then compiles the method's bytecode to machine code.*
* *C2 compiler* 
  + *C2 compiler is intended for use by server applications.*
  + *C2 compiler waits for a longer period before deciding to compile a hot method.*
  + *This gives the compiler more time to learn about the method's execution pattern so that it can infer and apply more aggressive optimizations.*
* *methods compiled with C2 are generally faster than methods compiled with C1.*
* *However, because the C1 starts earlier than C2, an application running with the C1 compiler will be faster earlier on in its lifecycle before the C2 catches up and overtakes it.*
* *Therefore, if an application is short lived or sensitive to startup times, then the C1 compiler is a better choice for it.*
* *But what if you could combine the two?* 
  + *What if you could have an application start running, and then after some runs, the C1 compiler picks up a method and optimizes it get the intermediate performance benefit.*
  + *And then later, after some more runs, the method is reoptimized with the C2 compiler to get the faster machine code.*
* *This is exactly what the JIT compilation does. Tiered compilation is a compilation mode that was introduced in Java 7, and in tiered compilation, hot methods are first compiled with C1, and then as they get hotter, they are recompiled with C2.*
* *Although tiered compilation was introduced in Java 7, it wasn't until Java 8 that it became the default JIT compilation mode in the HotSpot JVM.*
* *The actual mechanism of tiered compilation is a bit complex, so I'll present a simplified version of it here.*
* *In tiered compilation, there are five execution levels.*



* *In level 0, the code is run in pure interpreted mode.*
* *In level 1, hot methods are compiled using the C1 compiler, and no further profiling is done.*
* *In level 2, hot methods are compiled using the C1 compiler, and there's some limited profiling being done.*
* *In level 3, methods are compiled using the C1 compiler with full profiling being done. And*
* *in level 4, methods are compiled using the C2 compiler.*
* *Profiling is needed for the C2 compiler to be able to infer the advanced optimizations that help it produce high-performance code. The normal path for a hot method is that it runs in level 2 interpreted mode, and then, if it meets the threshold for compilation, it's queued up for the C1 compiler and compiled at level 3 with full profiling so that it can later be moved up to the C2 compiler once it meets the requirements for level 4 execution.*
* *There are alternative compilation paths that use the other levels in edge cases, like the method being too trivial for C2 compilation or one of the compilers being overloaded. The primary activity in compilation-based JVM tuning is choosing your compilation mode.*
* *This choice hinges on the type and nature of the application you’re executing.*
* *In general, because tiered compilation combines the best of server and client compilers, it should be your first choice.*
* *But there are situations where using just the client C1 compiler or the server C2 compiler could result in better performance.*
* *If you're developing a server app, then you should be choosing between the server compilation mode or the tiered compilation mode.*
* *For server applications, tiered compilation should perform better because you get the benefits of having C1 compile your hot methods in level 2 or 3 and provide some performance benefits before C2 compilation kicks in, whereas you wouldn't have that in plain server mode.*
* *However, it's still worth testing and benchmarking both compilation modes to see which works better for your application and workload.*
* *I've heard of a company that upgraded from Java 7 to Java 8, and after the upgrade,*



* *they noticed a drop in the performance of one of their applications. They spent a lot of time troubleshooting the performance decrease before realizing that the issue was that the Java 7 JRE they were initially using defaulted to the server compilation mode, while the Java 8 JRE defaults to the tiered compilation mode.*
* *As soon as they switched their Java 8 JRE to using server mode, their performance issue was fixed.*
* *So, I would recommend checking the performance of your app with both compilation modes.*

*JIT Tuning*

* *The method for directing a JVM to strictly use the server or a client compiler for JIT compilation has changed over time.*
* *In older versions of the Java Runtime Environment, you could do so by passing the -server or the -client flag as a Java command line option. But this has been changed since Java 8, and these flags are now largely ignored by the runtime.*
* *In Java 8, the simplest way to run in pure C1, that is client mode, is to set the TieredStopAtLevel flag to 1.*
  + *Java -XX:TieredStopAtLevel=1*
* *This setting keeps tiered compilation turned on, but caps the compilation level at level 1, leaving only 2 compilation levels,*
* *Available Levels*
  + *Level 0: Interpreted Code*
  + *Level 1: C1 Compiled with no profiling*
* *To run in pure C2 compilation mode, you need to turn off tiered compilation. This results in the simple compilation policy being activated instead of the advanced compilation policy.*
* *For the simple compilation policy, hot methods are compiled just once at the highest compilation level, which by default is the C2 compile level.*
* *hot method*
* *Hot method is quite complex and changes frequently between JDK updates. So again, I'm going to present to you a simplified version of it.*
* *The definition is primarily based on the value of two counters,*

|  |  |
| --- | --- |
| *invocation counter (i)* | *back edge counter (b)* |
| *The method invocation counter simply counts the number of times the method has been called* | *the back-edge counter counts the number of times any loops in the method have branched back. This branch back count is effectively the number of times a loop has completed execution*  *the back edge counter counts the number of times any loops in the method have branched back. This branch back count is effectively the number of times a loop has completed execution, either because it reached the end of the loop itself or because it executed a branching statement like continue.* |

*Hot Method Definition -Tired Compilation*

* *In tiered compilation mode, for an interpreted method to get selected for compilation by the C1 compiler in level 3, the following conditions must be met.*
* *i> Tier3InvocationThreshold || ( i > Tier3MinInvocationThreshold && i + b > Tier3CombileThreshold)*
  + *default Tier3InvocationThreshold = 200*
  + *default Tier3MinInvocationThreshold = 100*
  + *default Tier3CombileThreshold = 2000*
* *Default Hot method* 
  + *i> 200 || (i> 100 && i+b >2000)*
* *invocation count either has to be higher than the Tier3InvocationThreshold, or the invocation counts must be higher than the minimum invocation threshold, and the sum of the invocation count and backedge count is higher than the Tier3CompileThreshold.*
* *By default, the Tier2InvocationThreshold is set to 200, and the Tier3CompileThreshold is set to 2, 000.*
* *lower the threshold for level 4 compilation by passing in the appropriate command line flags*

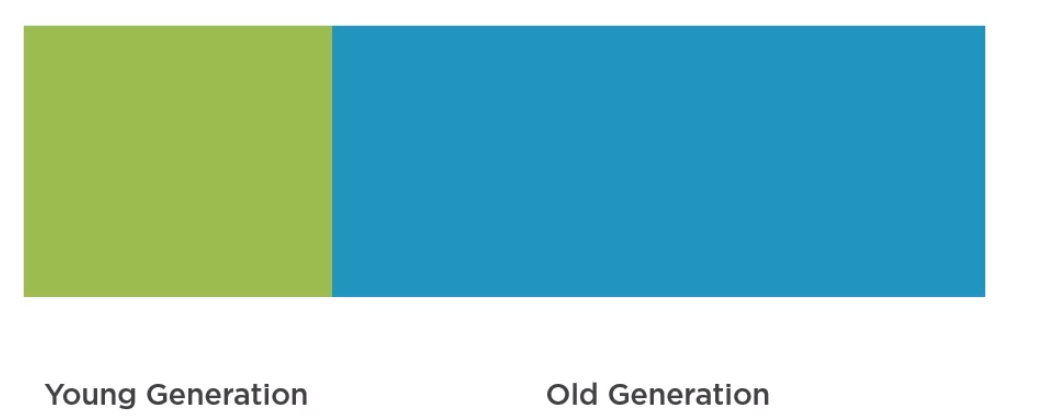
*Java -XX:Tier4InvocationThreshold=4000 –*

*XX:Tier4CompileThreshold=10000*

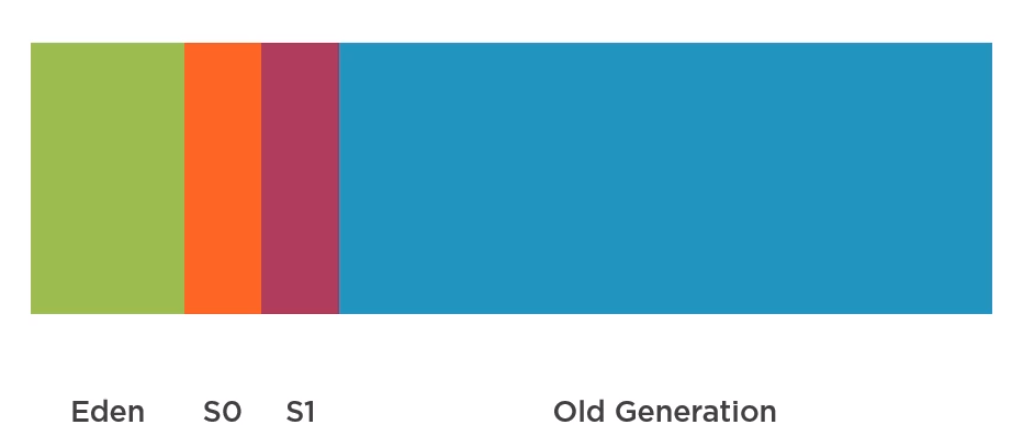
*Garbage collector*

*4 types of collectors*

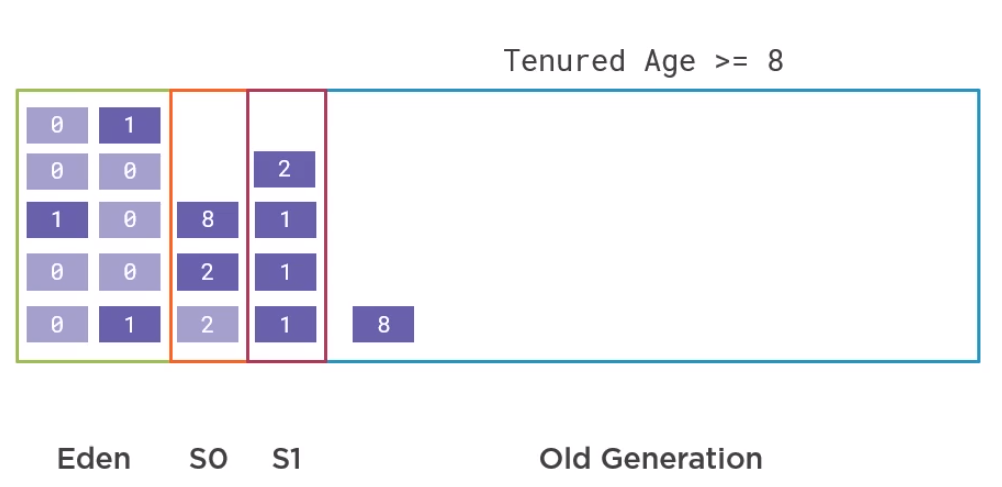
* *Serial Collector* 
  + *To enable -XX:+UseSerialGC*
  + *Use when only 1CPU/vCPU available and no pause time requirements*
  + *Multiple small JVMson a single machine*
  + *Small live data set (up tp 100MB)*
* *Parallel Collector (Throughput Collector)*
  + *Use multiple threads to process the heap*
  + *Fully stops all applications threads which can result in long pause times but higher thoughput*
  + *Use when*
    - *Batch application (application doesn’t need more user interaction, need more performance)*
  + *To Enable* 
    - *For Java 9: -XX: +UseParallelGC*
    - *For java 8: default enable*
* *Concurrent Mark Sleep (CMS Collector)*
  + *First collector introduced the Hotspot JVM*
  + *Can trace reachable objects and cleanup unreachable objects while application threads are running*
  + *GC threads may compete with application threads for CPU*
  + *To enable: -XX:+UseConcMarkSweepGC*
  + *Has been deprecated by G1GC from Java9*
* *Garbage First (G1GC)*
  + *Replace for CMS Collector*
  + *Default from java 9*
  + *Designed for multiprocessor machines with large heaps. Tries to archive the best balance between throughput and latency*
  + *GC Threads mark unreachable objects concurrently while application threads are running*
  + *Use when: you have an interactive application with low pause time requirements*
  + *To enable in java 7/8: -XX:+UseG1GC*
* *Shenandoah Collector (Future)*
  + *Future collector (Coming soon)*
  + *Next step beyond then G1GC*
  + *Can mark unreachable objects and move reachable objects concurrently while application threads are running*
  + *Can produce 10X reduction inpause times with only a 10% throughput decrease.*
* *The standard garbage collection architecture today is generational garbage collection,*
* *all four collectors are generational garbage collectors.*
* *Generational garbage collectors divide the heap into two areas,*



* *young generation area and the old generation, or tenured area.*
* *The young generation itself is split into two logical areas, eden space, also known as the allocation space, and the survivor space.*



* *the survivor space is also split up into two, survivor space S0, and survivor space S1.*
* *Objects are created in the Eden space of the young generation area, and when Eden fills up, a minor garbage collection takes place.*
* *Minor GCs are optimized with the assumption that objects have a high mortality rate and are typically very fast.*
* *During the minor GC, objects are first checked to see if they're still reachable or not and then marked accordingly.*
* *Then reachable objects in eden have their age incremented and are then copied to a designated survivor space.*
* *Objects in the other survivor space also have their age incremented, and if they reach a certain threshold known as the tenured age, they get promoted to the tenured generation.*



* *But if they're under the tenured age, then they also get copied to the currently designated survivor space.*
* *Then both Eden and the other survivor space are cleared, freeing the memory and compacting the space at the same time.*
* *The process is the same on the next minor GC run, but the survivor spaces switch roles, and the empty survivor space becomes the designated survivor space.*
* *Referenced objects are copied to this survivor space or tenured to the old generation, while Eden and the other survivor space get cleared after the GC.*
* *it goes back and forth from there. Eventually, the old generation also gets filled up and must be garbage collected. This is called a major, or full garbage collection, and it usually takes up more time because the search space and number of objects is higher.*
* *Basic collections* 
  + *will stop all application threads,*
  + *mark the unreachable objects,*
  + *free their memory,*
  + *compact the heap,*
  + *resume the application threads.*
* *Advanced collectors*
  + *scan for unreachable objects, even when the application threads are still running,*
  + *only pause all application threads to free the memory and compact the heap.*
  + *These collectors are known as concurrent collectors, mostly concurrent collectors, or low-pause collectors.*
  + *The Concurrent Mark Sweep and G1GC collectors are concurrent collectors, while the parallel collector is a basic collector.*

*GC Logging*

* *To enable Logging up to Java 8*
  + *Java -XX:+printGCDetails -XX:+printGCDateStamps -Xloggc:<file-path> -XX: +UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=10M*
  + *-XX:+printGCDetails* 
    - *Tells the JVM to print statistics on GC events and how much memory is reclaimed by garbage collection*
  + *-XX:+PrintGCDateStamps*
    - *The PrintGCDateStamps flag, which tells the JVM to print the absolute timestamp in the log statement instead of relative time*
  + *-Xloggc:<file-path>*
    - *a path to a file that the log will be written to. If the file path is not given, then the log will be written to standard out*
  + *+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=10M*
    - *tells the garbage collector to rotate log files, keeping the last 10 log files with each file maxing out at 10 MB*
* *GC Logging enable in java 9 and above*
  + *Java -X:log:gc\*:file=<file-path>:filecount=10. Filesize=10M*
  + *The log:gc flag enables GC logging to standard out*
  + *adding the file path redirects the output to the specified file*
  + *To enable log rotation, you can add the filesize and filecount options*
  + *additional details to the GC log, like the liveliness info, the heap info, the tenuring age distribution and more add an asterisk to the GC portion of the flag*

*Note: for Garbage log analysis 🡪 gceasy.com*

*GC Tuning*

* *Tuning the size of the heap*
  + *If the heap is too small*
    - *there’ll be too many GC’s to free memory and application thought put would suffer*
  + *if the heap is too big*
    - *GC pause would take too long, and response time metrics would be affected*
  + *Change the heap size*
    - *Java -XmsN -XmxN*
* *Adaptive sizing*
  + *If number of objects are increase the heap memory size will increase, If no of objects less then heap memory size will decrease this process is called adaptive sizing.*
  + *It should do so only if you’ve finely tuned your application’s GC behavior and size*
  + *Flags#*
    - *-XX:UseAdaptiveSizePolicy*
    - *-Xms6g -Xmx6g*
    - *-XX:newSize=1g -XX:MaxNewSize=1g*

***Search and Data structures***

*Big-O notation*

* *Measure of the performance (time complexity) of an algorithm.*
* *Describe the upper bound of the execution time of an algorithm relative to the size of the input*
* *O (1), which denotes an algorithm which always completes in constant time, regardless of the size of the input*
  + *for example, getting the value of an item from an array;*
* *O(log(n)) which denotes an algorithm whose completion time is a log function of the size of the input*
  + *for example, binary search over a sorted list;*
* *O(n), which denotes an algorithm whose runtime is linear and grows in direct proportion to the size of the input*
  + *for example, a linear search;*
* *O (n log(n)), which denotes an algorithm with a runtime that's directly proportional to the size of the input multiplied by the log of the size of the input*
  + *for example, the merge sort algorithm,*
* *O(n2), which denotes an algorithm whose runtime is quadratic and grows in direct proportion to the size of the square of the input, for example, having an inner loop that iterates over the complete input data set; and lastly,*
* *O(2n), which denotes an algorithm that doubles in runtime with each addition to the input data set*
  + *for example, this recursive Fibonacci implementation.*