Introduction to JVM

Java Performance– Making the JVM perform

- Java is one of the most popular development platform and has been one for many years
- Java platform is much more than just the Java language itself
- The real backbone of the platform is its Virtual Machine that does the major heavy lifting required for managing memory, threads etc
- Java platform has evolved to support new age programming languages like Scala, Groovy etc.
- In all this, the one thing that stays is the JVM

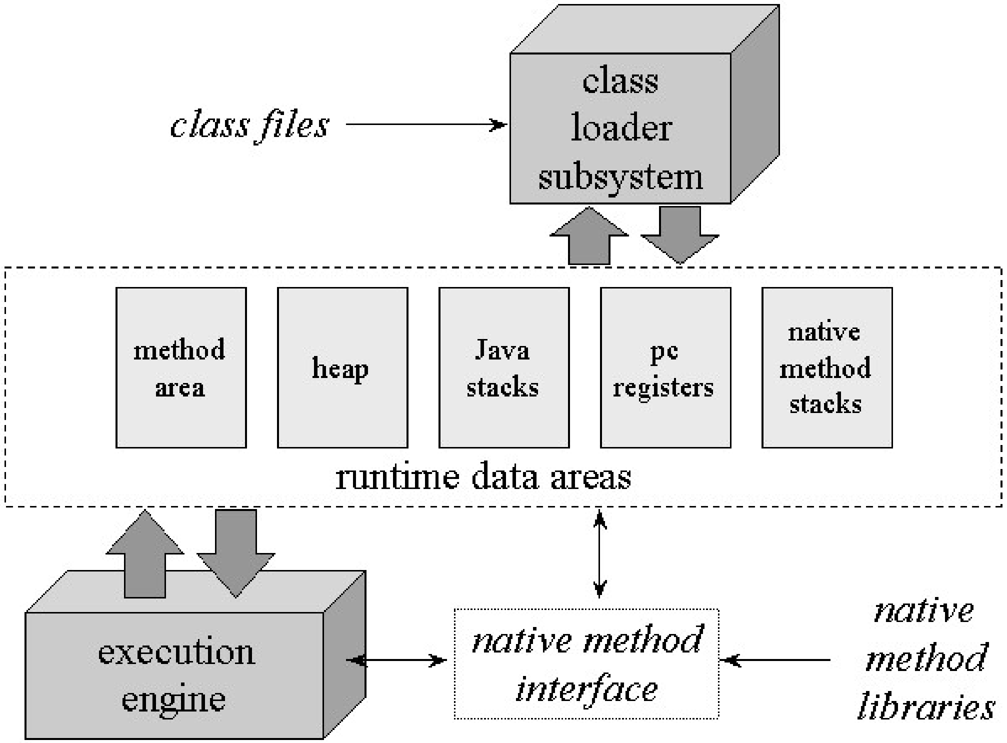
Java – Making the JVM perform

- JVM is something that is very familiar to every Java developer, but to use it with efficiency, its important to know the

JVM hides the complexity under the hoods.
But open the hoods to drive it better.

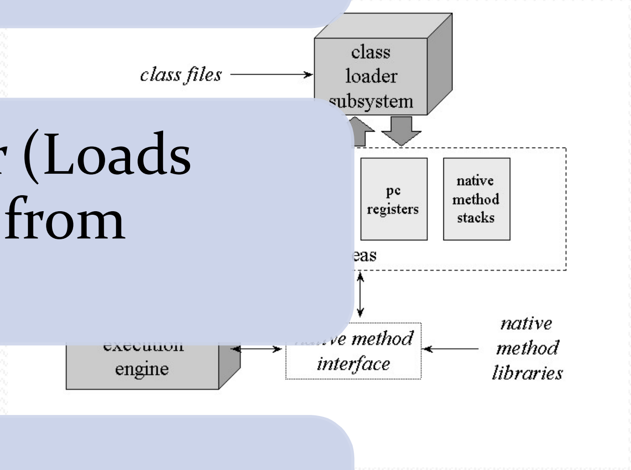
The JVM Architecture

- Before going deeper into the JVM, lets first understand what constitutes the JVM



The Class Loader System

- Befo … to load the r
- JVM … ding system to load
- The … up of … s loaders



Boot

- BootClassLoader (Loads JVM itself)

System

- SystemClassLoader (Loads application classes from classpath)

Custom

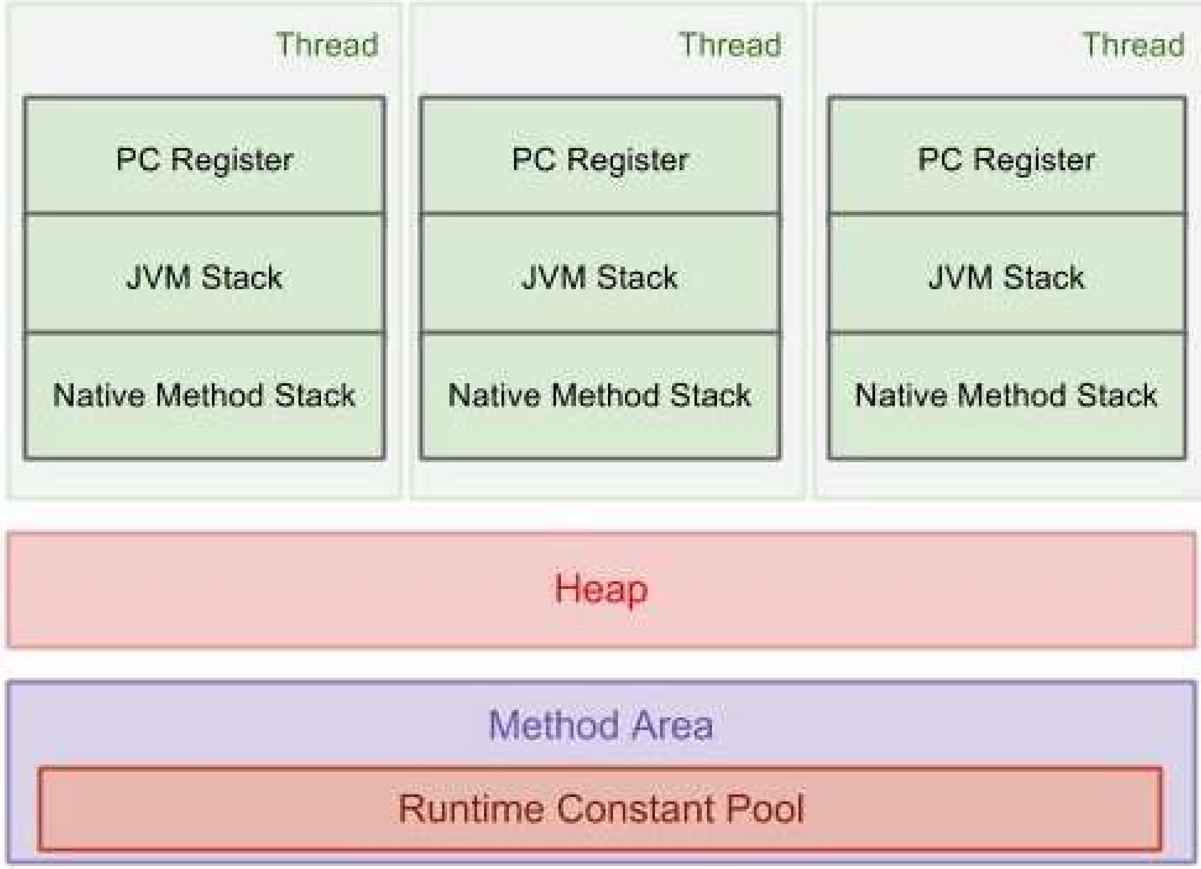
- CustomClassLoader (Usually app servers create this)

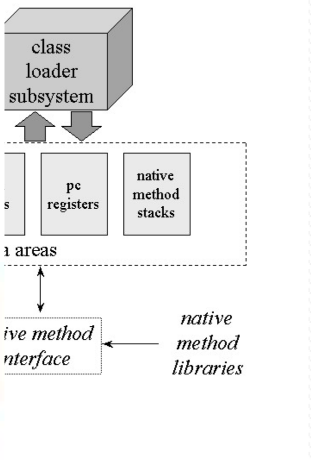
The Runtime Data Area

- There are

JVM

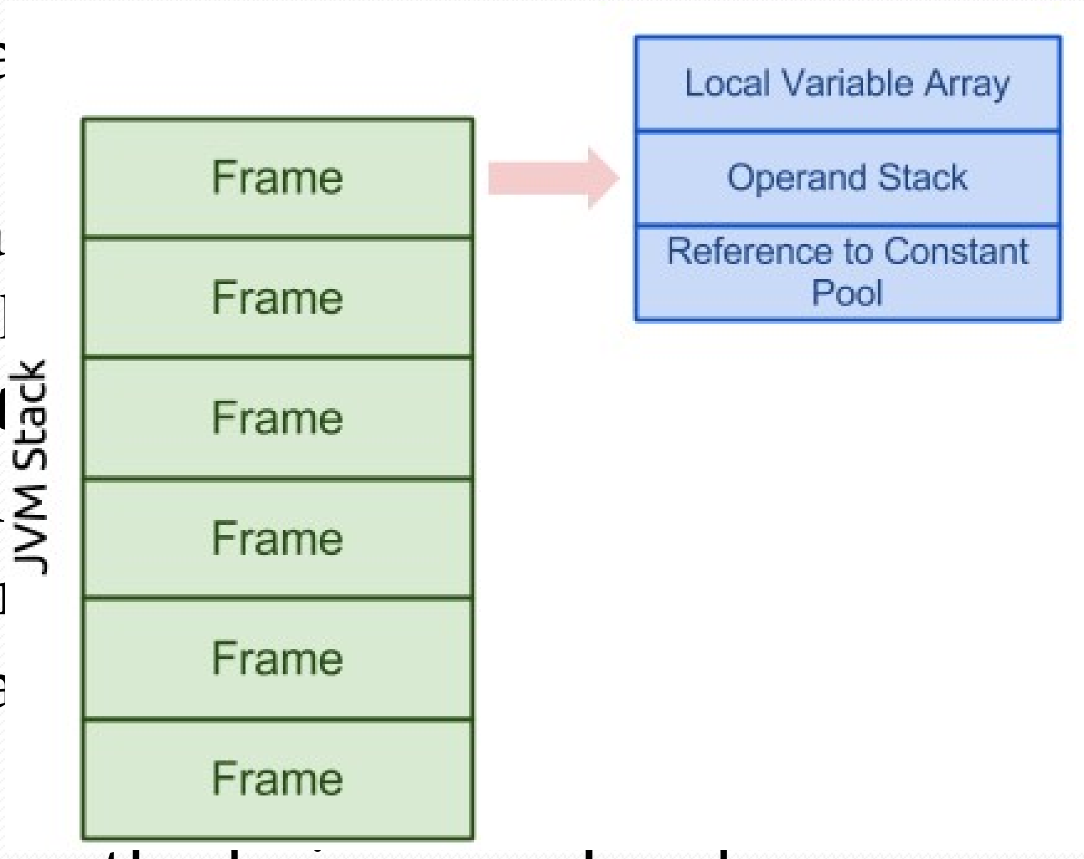
1. Data Ar
2. Data Ar

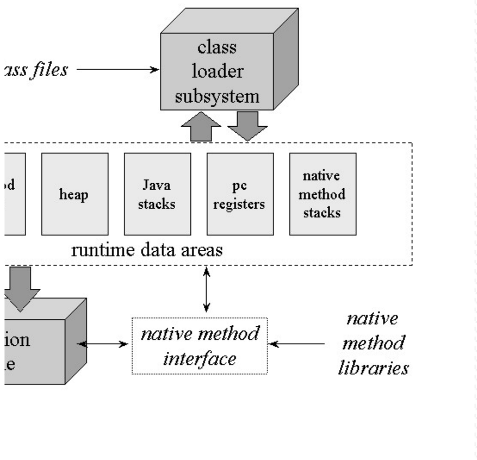




Data Area For Individual Threads

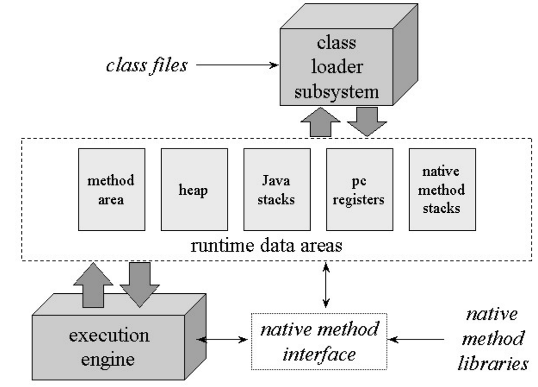
- This are ed and includes
  - Progra Control
  - JVM St contai Operan Consta
  - Native native methods, i.e., non-Java language methods.





Data Area Shared by All Threads

- All threads share Heap and Method area
- Heap: it is the area that we most frequently deal with. It stores arrays and objects, created when JVM starts up. Garbage Collection works in this area.
- Method Area: it stores run-time constant pool, field and method data, me constructors code

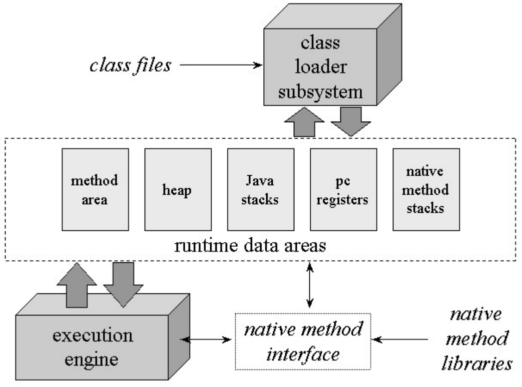


- Runtime Constant Pool: It is a per class, per interface representation of constant Pool. It contains several types of constants ranging from numerals known at compile time to references that must be resolved at run-time

The JVM Execution Engine

- The execution engine is the one that provides the execution environment for the applications
- It provides the necessary services to

The application



Measuring to Tune

Does my application need tuning?

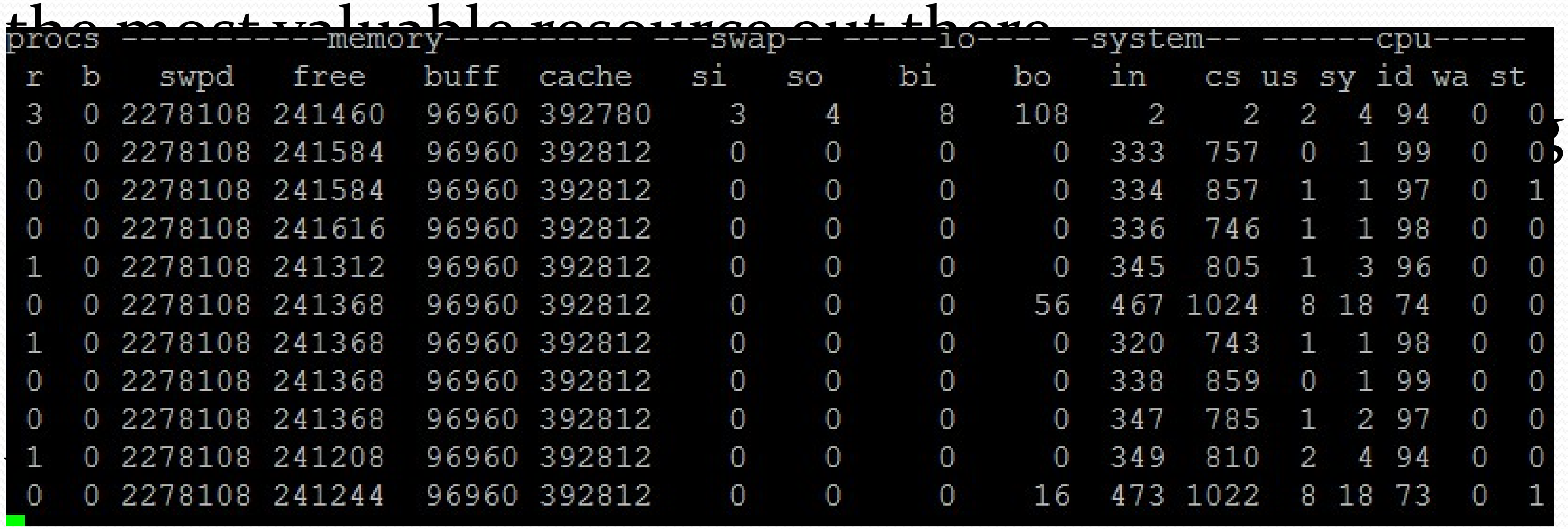
- Before embarking on improving the performance of your application it is important to first find out whether there is a need for tuning
- The answer to this will come from what kind of performance is expected from your application and what is the actual performance that your application is providing.
- There are multiple tools available that can be used to run performance test on your application to set the current benchmark and compare it against the expectations

What is affecting the performance?

- If the performance noted is not up to the mark then it is important to find out what is causing the performance issue
- There could be multiple reasons for the performance being not up to the mark
  1. Inefficient Application Algorithm
  2. Database that is not responsive
  3. Inefficient usage of the available resources
  4. Inefficiently configured JVM
- The first two points would require the developer to dig deep into their code or architecture and find a way to improve things

Is my app using the available resources?

- All programs need CPU time to execute, and CPU is the most valuable resource out there.



```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0 2278108 241460  96960 392780    3    4     8   108    2    2  2  4 94  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  333  757  0  1 99  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  334  857  1  1 97  0  1
 0  0 2278108 241616  96960 392812    0    0     0     0  336  746  1  1 98  0  0
 1  0 2278108 241312  96960 392812    0    0     0     0  345  805  1  3 96  0  0
 0  0 2278108 241368  96960 392812    0    0     0    56  467 1024  8 18 74  0  0
 1  0 2278108 241368  96960 392812    0    0     0     0  320  743  1  1 98  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  338  859  0  1 99  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  347  785  1  2 97  0  0
 1  0 2278108 241208  96960 392812    0    0     0     0  349  810  2  4 94  0  0
 0  0 2278108 241244  96960 392812    0    0     0    16  473 1022  8 18 73  0  1
```

- Running vmstat on a machine will provide insight into the usage of CPU by an application.

Reading the CPU utiliza

us: Time Spent by application

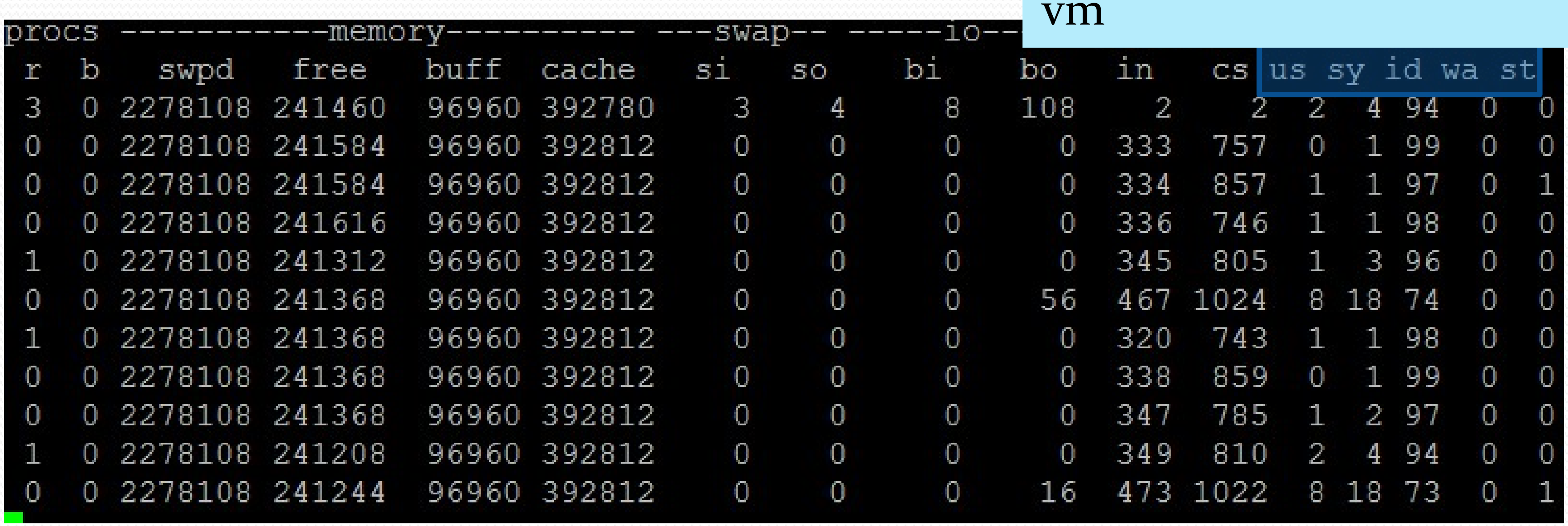
sy: Time Spent by system

id: Time for which CPU was idle

wa: Time for which CPU was waiting on IO

st: Time stolen by hyperv from vm

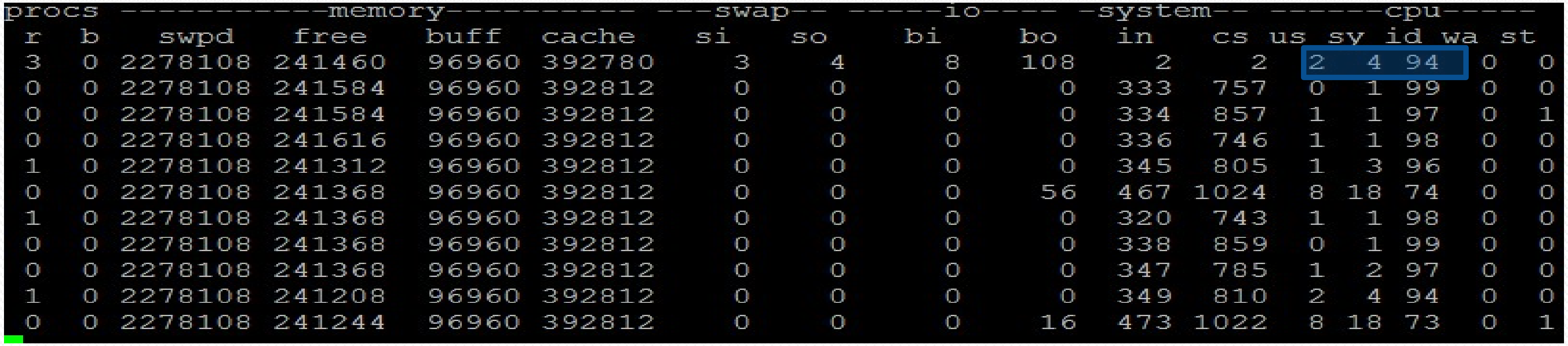
- The reading is provided every secon
  of how long the CPU was busy in th
- The CPU could be used by the appl
  system (OS) itself, for IO or networ



```
procs -----------memory---------- ---swap-- -----io--
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0 2278108 241460  96960 392780    3    4     8   108    2    2  2  4 94  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  333  757  0  1 99  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  334  857  1  1 97  0  1
 0  0 2278108 241616  96960 392812    0    0     0     0  336  746  1  1 98  0  0
 1  0 2278108 241312  96960 392812    0    0     0     0  345  805  1  3 96  0  0
 0  0 2278108 241368  96960 392812    0    0     0    56  467 1024  8 18 74  0  0
 1  0 2278108 241368  96960 392812    0    0     0     0  320  743  1  1 98  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  338  859  0  1 99  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  347  785  1  2 97  0  0
 1  0 2278108 241208  96960 392812    0    0     0     0  349  810  2  4 94  0  0
 0  0 2278108 241244  96960 392812    0    0     0    16  473 1022  8 18 73  0  1
```

So is everything ok?

- The application should aim for maximum utilization of the CPU.
- The aim is to use the CPU to its maximum for as short a time as possible.
- The first reading below shows the app used CPU for 2% of time while system did it for 4 %. The CPU was idle 94 % of the time.



```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0 2278108 241460  96960 392780    3    4     8   108    2    2  2  4 94  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  333  757  0  1 99  0  0
 0  0 2278108 241584  96960 392812    0    0     0     0  334  857  1  1 97  0  1
 0  0 2278108 241616  96960 392812    0    0     0     0  336  746  1  1 98  0  0
 1  0 2278108 241312  96960 392812    0    0     0     0  345  805  1  3 96  0  0
 0  0 2278108 241368  96960 392812    0    0     0    56  467 1024  8 18 74  0  0
 1  0 2278108 241368  96960 392812    0    0     0     0  320  743  1  1 98  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  338  859  0  1 99  0  0
 0  0 2278108 241368  96960 392812    0    0     0     0  347  785  1  2 97  0  0
 1  0 2278108 241208  96960 392812    0    0     0     0  349  810  2  4 94  0  0
 0  0 2278108 241244  96960 392812    0    0     0    16  473 1022  8 18 73  0  1
```

Why is my CPU so idle?

- The CPU could be idle for multiple reasons.
  1. The application might be blocked on synchronization primitive and unable to execute until lock is released
  2. The application is waiting for some reason, like call to a database
  3. The application has nothing to do
- The first two points might be improved through optimizing code for resource access like faster db execution or better thread usage.
- The last point is where the complexity lies. Is the CPU choosing to be idle, when it could be doing something?

Factors affecting Performance

How does JVM play a role?

- The JVM is a highly sophisticated system that not just runs the program but contributes to its performance in multiple ways.
- The two biggest way in which the performance is impacted by JVM are
  - Just in time compilation
  - Garbage collection

Just In Time Compilation (JIT Compiler)

Java code execution

- Java is a cross platform language and achieves this through its generation of intermediate byte code
- The byte codes are loaded on the JVM and executed by the Java interpreter
- Traditional programming languages like C and C++ compile their program to the OS specific binaries. Compiled binaries execute
- Interpreted code run slower than compiled code as the execution environment executes one line at a time and thus can not make any optimizations that might be possible

Just In Time Compilation

- To make the java application perform better, the JVM chooses to compile the code Just In Time
- This means the code that runs more often and needs to be executed faster will be compiled with optimizations applied

HotSpot Compilation

- HotSpot compilation refers to compilation of piece of code that is executed most often
- When a piece of code is executed by JVM it is not compiled immediately
- JVM does not wish to waste cycles on compiling code that might be executed only once in the application life cycle. Interpreting and running such code will work faster

Flavours of Compiler

- The JIT compiler comes in two flavours
  - Client
  - Server
- The choice of which compiler to use depends on the hardware platform and configuration.
- The fundamental difference between the two types of compilers is its aggressiveness in compilation

Client Compiler

- The client compiler begins compiling code faster than the server compiler does
- During beginning of code execution, the client compiler will be faster than the server one.

Server Compiler

- The server compiler begins compiling code later than the what client compiler would do
- Server compiler waits to gain more information about the application usage before it compiles the identified code
- Knowing more about the code enables compiler to apply optimizations

Tiered Compilation

- The tiered compilation was introduced in Java 7
- Tiered compilation brings client start up speed to the server VM
- The client compiler is used to generate compiled versions of methods that collect profiling information about themselves
- The tiered scheme can also achieve better peak performance than a regular server VM because the faster profiling phase allows a longer period of profiling, which may yield better optimization.

Compiler Flags

- Unlike most java flags, the flags to select compiler are different. They do not use –XX
- The standard compiler flags are simple words
  - -client
  - -server
  - -d64 (gets the JVM into the 64-bit mode)

java –server –XX:+TieredCompilation other_args

Optimizations during JIT

- Analysis of the code execution enables JVM to apply multiple types of optimizations.
- Simple optimization like eagerly substituting a specific object type in a polymorphic case

Example-
Consider statement b = obj1.equals(obj2);

Over time, the JVM may notice that the obj1 is always an instance of java.lang.String and may compile the code to call String.equals() directly. Such a code will execute faster than the original code.

Inlining of Code

- Good object oriented code often have Javabeans with several properties accessible as setters and getters.

```
Example-
class A{
        int b;
        void setB(int i){b=i;}
        int getB()={return b;}
}
```

- This means accessing the value b would result in method calls which have its over heads. Fortunately, the JVM inlines such code that boosts performance immensely.

Inlining of code

So code like

```
A  a = new A();
a.setB(a.getB() * 2)
```

will be inlined to

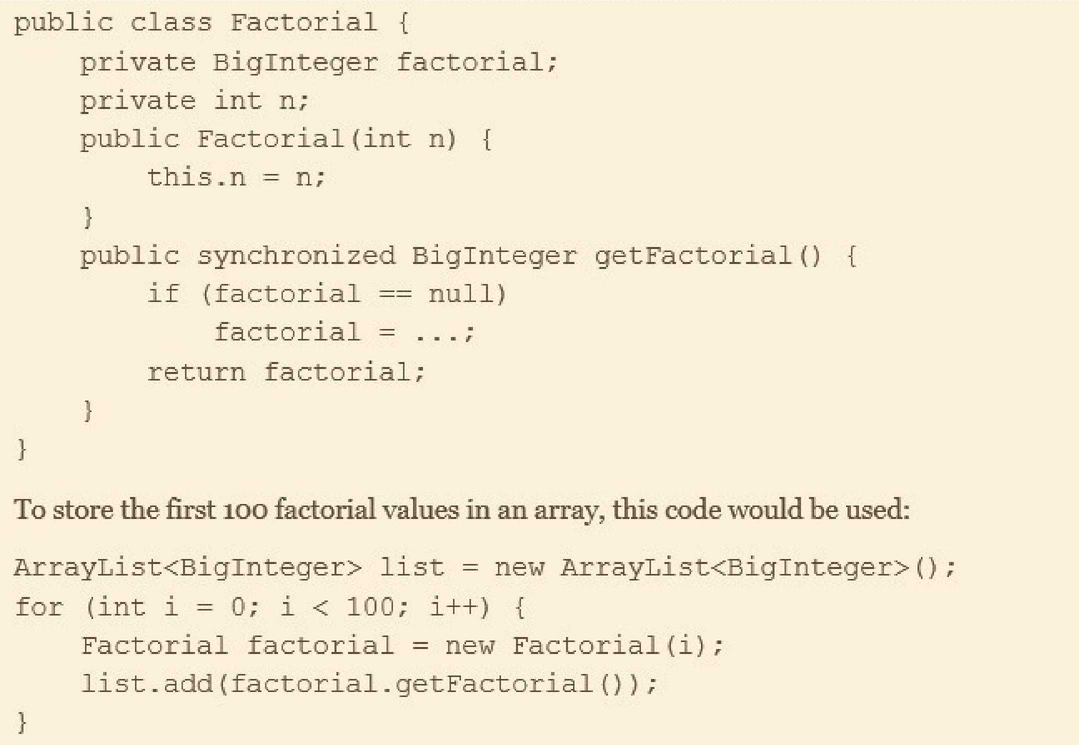
```
A a = new A();
a.b = a.b*2;
```

Inlining is enabled by default.

Escape Analysis

- Server compiler performs very aggressive optimizations if the escape analysis is on –XX:DoEscapeAnalysis. This is on by default
- If the JVM finds out getFactorial() is called only within the loop it will bypass the need to get synchronization lock.

  It can also choose to keep "n" in register.



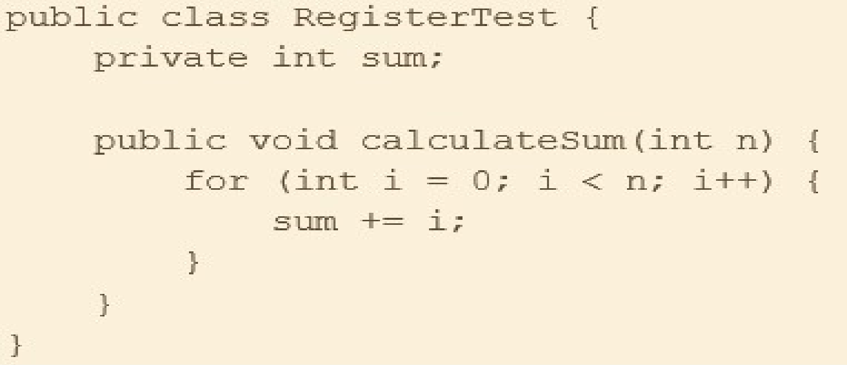
```
public class Factorial {
    private BigInteger factorial;
    private int n;
    public Factorial(int n) {
        this.n = n;
    }
    public synchronized BigInteger getFactorial() {
        if (factorial == null)
            factorial = ...;
        return factorial;
    }
}
```

To store the first 100 factorial values in an array, this code would be used:

```
ArrayList<BigInteger> list = new ArrayList<BigInteger>();
for (int i = 0; i < 100; i++) {
    Factorial factorial = new Factorial(i);
    list.add(factorial.getFactorial());
}
```

Registers and Main Memory

- One of the main optimization that compiler can make is to decide on when to use values from main memory and when to store values in register
- Consider the code
- The JVM can optimize the performance of this code by loading the sum into the register instead of reading it from main memory

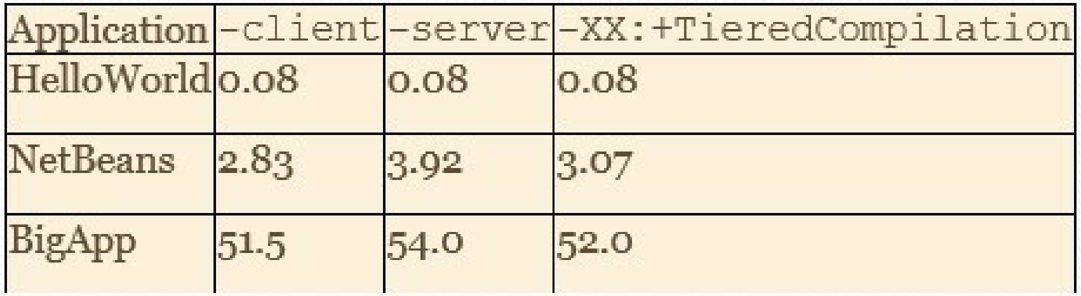


```
public class RegisterTest {
    private int sum;

    public void calculateSum(int n) {
        for (int i = 0; i < n; i++) {
            sum += i;
        }
    }
}
```

Optimizing Startup

- The client compiler is most often used when fast startup is the primary goal



Application	-client	-server	-XX:+TieredCompilation
HelloWorld	0.08	0.08	0.08
NetBeans	2.83	3.92	3.07
BigApp	51.5	54.0	52.0

- As can be seen, an application like NetBeans that loads 10K classes and performs various initializations at startup gains immensely from client compiler.
- TieredCompilation, proves to be faster than the server compiler

Optimizing Batch Operations

- Batch operations run a fixed amount of operations and the choice of compiler depends on which compiler gives best optimization in a given amount of time

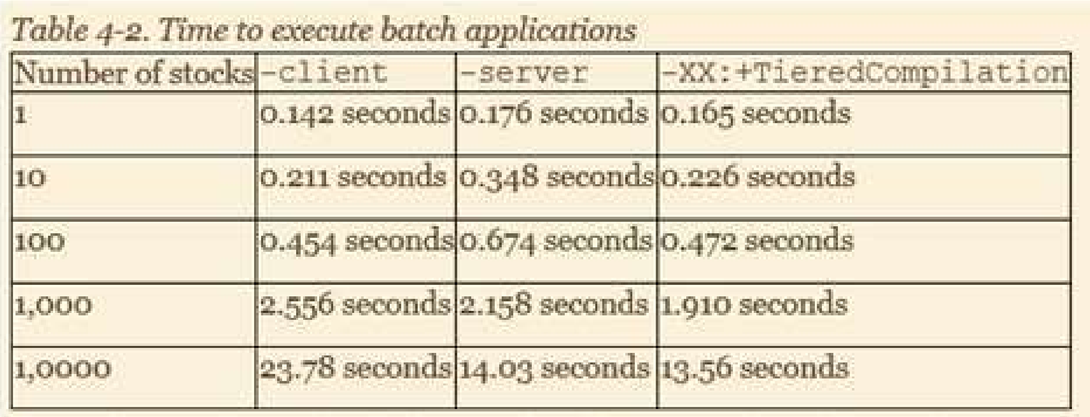


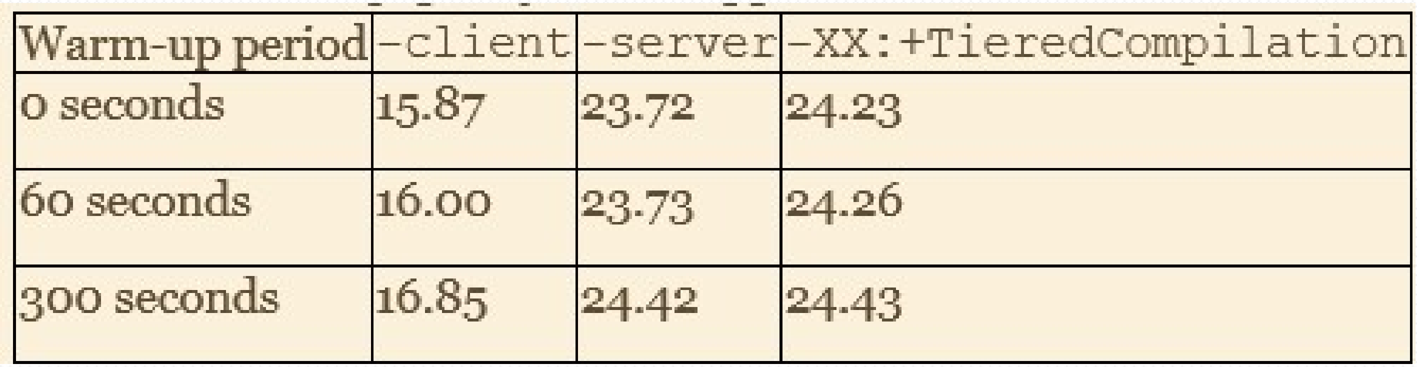
Table 4-2. Time to execute batch applications

Number of stocks	-client	-server	-XX:+TieredCompilation
1	0.142 seconds	0.176 seconds	0.165 seconds
10	0.211 seconds	0.348 seconds	0.226 seconds
100	0.454 seconds	0.674 seconds	0.472 seconds
1,000	2.556 seconds	2.158 seconds	1.910 seconds
1,0000	23.78 seconds	14.03 seconds	13.56 seconds

- For 1-100, faster startup of client compiler completes the job faster
- Beyond that the advantage tilts towards server, and particularly the one with TieredCompilation on.

Optimizing Long Running Operations

- Performance of long running applications is typically measured after the “Warm-up” period.



Warm-up period	-client	-server	-XX:+TieredCompilation
0 seconds	15.87	23.72	24.23
60 seconds	16.00	23.73	24.26
300 seconds	16.85	24.42	24.43

- The server compiler provide better throughput even in a very low warm up scenario, since it can apply enough optimizations to derive faster performance.

32 bit JVM vs 64 bit JVM

- If you have a 32 bit OS, then you have to use 32 bit JVM
- If you have a 64 bit OS, then you can either choose 32 bit JVM or a 64 bit one
- If the size of your heap is less than 3 GB, then a 32 bit JVM will perform better than a 64 bit one.
  - This is because memory references within JVM will be 32 bits and manipulating them is easier than manipulating 64 bit references
  - The 32 bit references also consume less memory
- The downside to 32 bit JVM is that the total process size must be less than 4 GB

Code Cache

- When JVM compiles code, it loads the assembly instructions into a code cache
- The code cache has a fixed size and once it is full, the JVM will not be able to compile any further code
- The possibility of code cache filling up is much higher in case of client or tiered compilation
- The server compiler will usually pick very few classes to compile and so is unlikely to run out of code cache
- When using the tiered compilation, its often important to monitor the code cache threshold and adjust it to tune the application

Code Cache

- The maximum size of code cache is set by –XX:ReservedCodeCacheSize=N Flag
- The initial size is set by –XX:InitialCodeCacheSize=N flag
- Reserved Code Cache size setting is enough as resizing of code cache happens in the background and does not impact performance

When does the code get compiled?

- The major factor that trigger the code compilation is the number of time a method or a loop has been executed
- Once the code is executed a certain number of time, the compilation threshold is reached
- Compilation is based on two counters in JVM
  - Number of times a method has been called
  - Number of times a loop has completed execution, either because it reached the end or encountered a *continue*
- When the JVM executes a method it checks if the compilation threshold has been reached. If it has then it adds it to the compilation queue.

On-Stack Replacement

- In case there is a loop that lasts long or does not end but controls the entire execution of the program, then the loop is compiled based on the threshold defined for the how many times a loop has executed
- Once the threshold is reached, the loop (not the entire method) is compiled
- The JVM will load the compiled code and the next execution of the loop will happen with the compiled code and will be much faster. This kind of compilation is called On-Stack-Replacement compilation.

Inspecting compilation process

- Standard compilation is triggered by -XX:CompileThreshold=N flag.
- To inspect compilation process use the tool *jstat*. Enable printing of compilation with the flag -XX:+PrintCompilation
- Tiered compilation levels are as follows
  - 0. Interpreted Code
  - 1. Simple C1 compiled code
  - 2. Limited C1 compiled code
  - 3. Full C1 compiled code
  - 4. C2 compiled code

Compilation Threads

- When a method becomes eligible for compilation, it is queued for compilation
- The queue is processed by one or more background threads ensuring that the compilation is achieved asynchronously.

View Example

De-optimzations

- When running the example, we can see couple of places where the compiler
- The common types of de-optimizations are
  1. Making code “Not Entrant”
  2. Making code “zombie”

Not Entrant Code

- There are two reasons for a code to get not-entrant
- One is the way classes and interfaces work
- One is due to the behaviour of tiered compilation
- Consider you program with 1 interface and 2 implementations.
- If JVM determines use of only one implementation, then it will compile method for that implementation
- If in some scenario, the other implementation is used, then the original compiled method is made not entrant
- In second case for a tiered compilation, when server compiler is ready, JVM marks some of code compiled by client as not entrant.

Zombie Code

- Zombie code occurs when objects associated with a not entrant code are no longer referenced and are up for GC
- Once all the objects are replaced the compiler notices that the compiled methods are now no longer required and is marked as zombie
- For performance this is a good thing as code cache is limited and need to be freed up to perform more compilations

Garbage Collection

What is garbage collection?

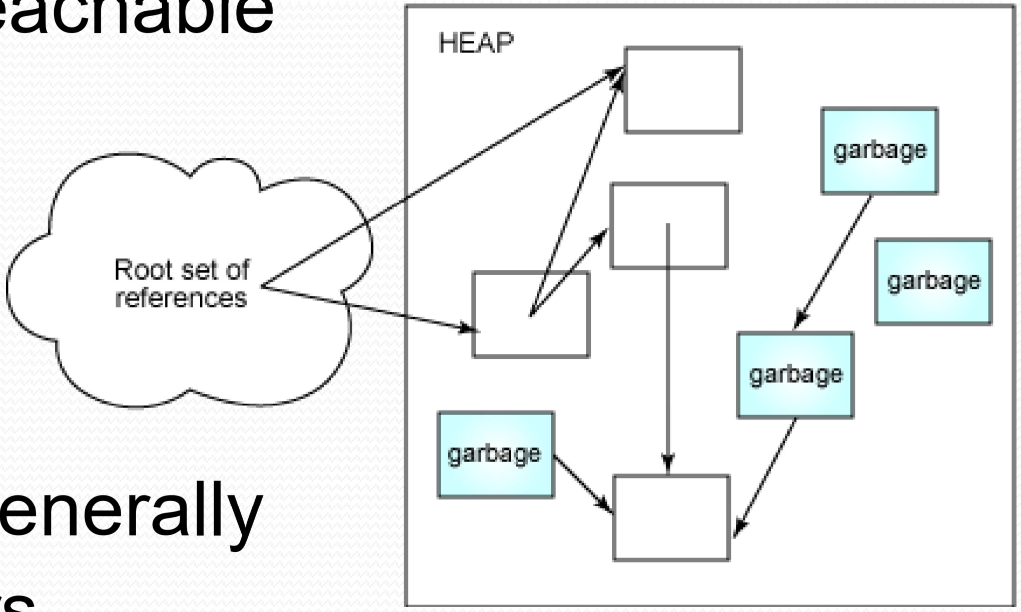
- Java platform free up the developers from the headaches of memory management
- Memory management is one of the main functions of JVM and it accomplishes this using the garbage collectors
- Garbage collectors are responsible to keep track of all the objects that are unused and remove them so as to free up the memory
- Without Garbage Collection, your program would of course not work too long

What are GC challenges?

- Freeing up memory and not leaving behind any unused object is a challenge in its own and requires some complex algorithms
- Two of the common ways to find out if a particular object is a garbage or not is through
  - Reference Counting
  - Object graph tracing
- Java does not use Reference Counting as it is prone to issues, especially in the case of circular reference

Tracing Collectors

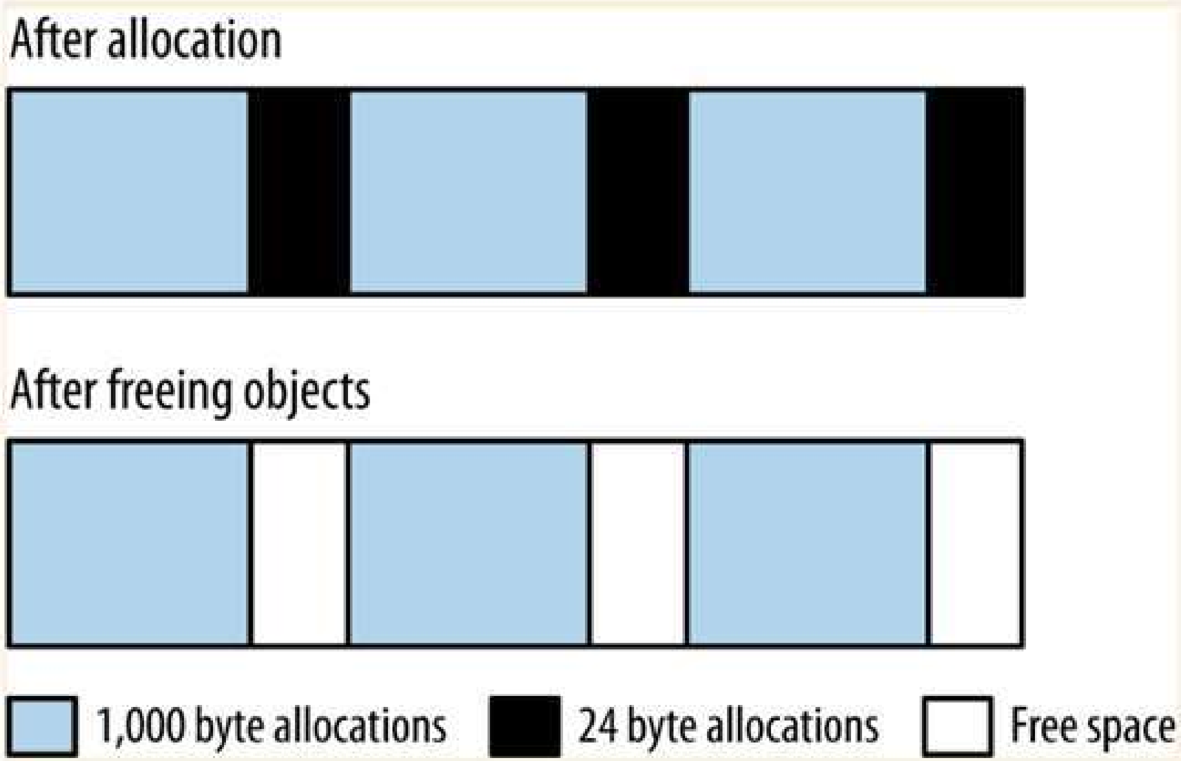
- Collectors take a snapshot of root objects. Objects that referenced from stack (Local Variables) or Perm Gen Space (Static Variables)
- Starts tracing objects reachable from root objects and marks them as reachable.
- Rest is Garbage
- Tracing collectors are generally Stop the world collectors



Types of Tracing Collectors

- There are multiple types of tracing collectors
  - Mark-Sweep
  - Copying Collector

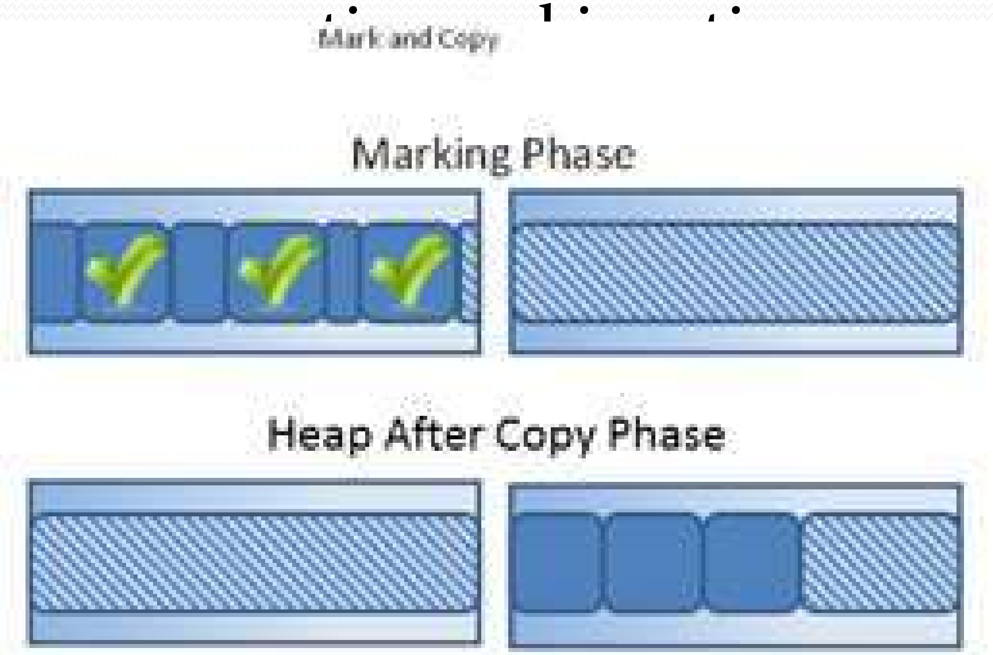
Mark and Sweep Collectors



- This is the
- Marking
  - Every obj tor clears all the ma le
- Sweep
  - The colle nd collects t
- Challenges
  - Collector has to go through all allocated objects in the sweep phase
  - Leaves the memory fragmented

Copying Collectors

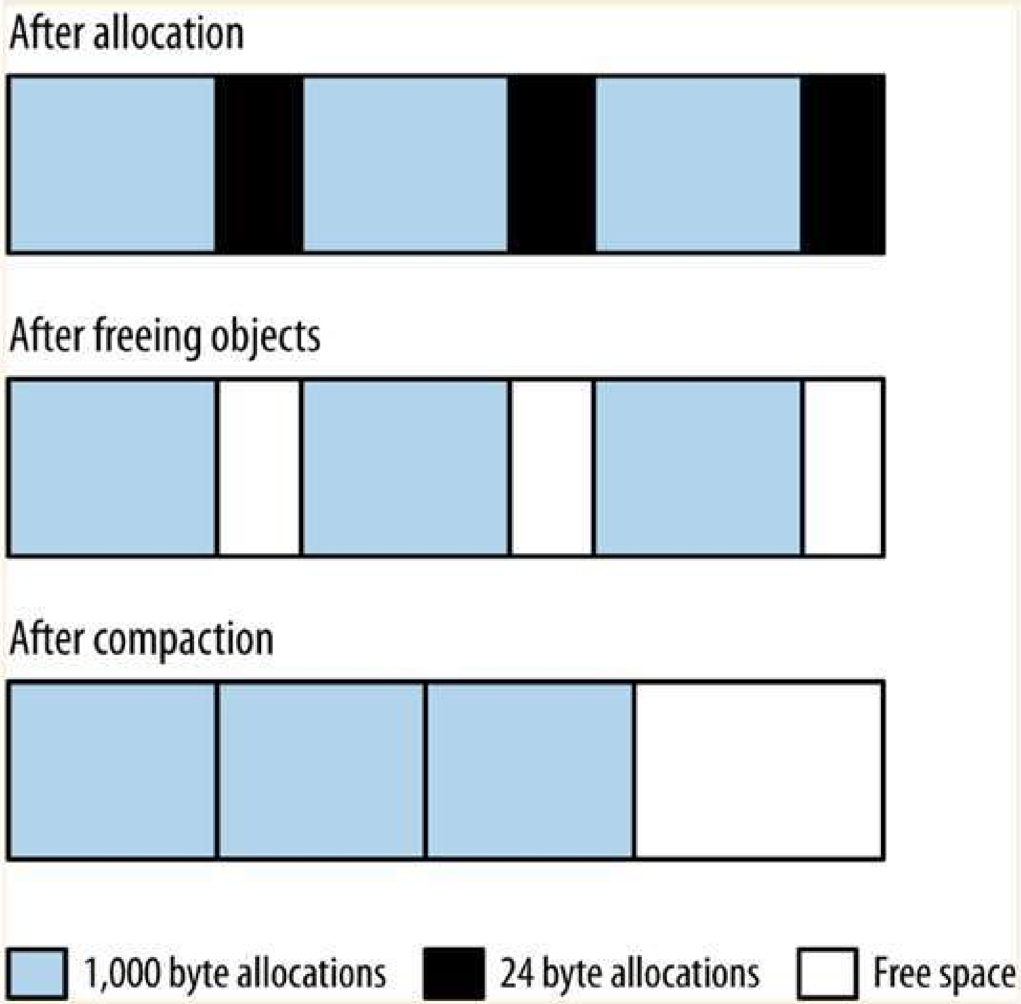
- Overcomes issue with Mark Sweep
- Creates two
- Moves survi e space.
- Roles of spa
- Advantages
  - Does not l its marker.
  - Solves the



- Disadvantag
  - Overhead of copying objects
  - Adjusting references to point to new location

Mark and Compact

- Overcomes c t needed) & Mark-Sweep
- Marking - Sa ive objects and marks as
- Compaction hat all live objects are co
- Clear demarc ap and free area.
- Long lived ol ttom of the heap so that e in copying colle



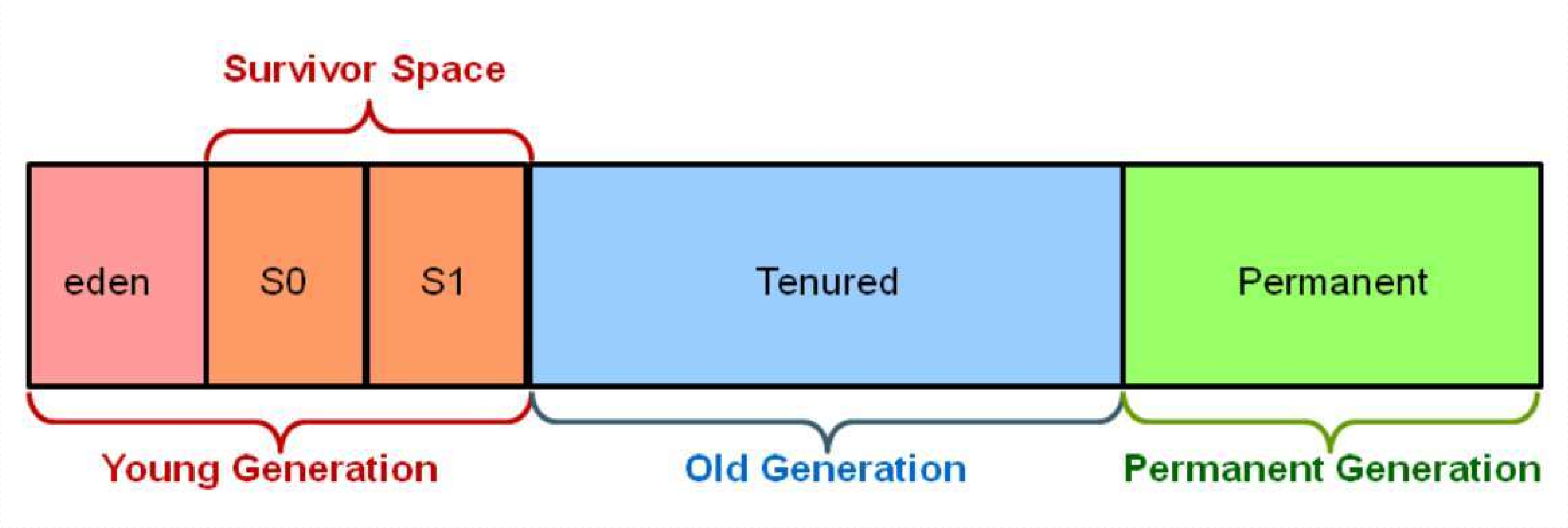
Performance impact of Garbage Collection

How GC impacts Performance?

- GC is one of the biggest creator of performance issues in JVM
- The various operations that the GC performs in the background impact performance
    - Stop the world collection of garbage
    - Mark and sweeping in a background thread, taking away CPU cycles from application threads
    - Compaction of memory
    - Copying of survivors and update of reference

Heap Organization

- All garbage collectors take a generational approach to



How do generations help?

- A typical (well written) object oriented code will create a lot of objects that stay for a very small duration and then are discarded

```
sum = BigDecimal(10);
for(Price p:prices.value()){
    BigDecimal diff = p.closingPrice().subtract(averagePrice);
    diff=diff.multiply(diff);
    sum=sum.add(diff);
}
```

- As can be seen in the code above, the BigDecimal diff, which is immutable is assigned multiple times in a loop. Every time it creates a new reference and the earlier reference is discarded

How do generations help?

- Garbage collection is designed to take advantage of such behaviour and by allocating the newly created objects in the young generation space which occupies a smaller percentage of the overall memory
- Once the space fills up, the garbage collector stops the application to perform garbage collection
- The fact that it has to cover a smaller memory footprint means the applications are paused for a smaller amount of time
- Also, since all objects are allocated in Eden, the GC moves all referenced objects to survivor space while leaving Eden Empty. This achieves automatic compaction.

How does garbage collection happen in spaces

- All GC algorithms have stop the world pauses for the young generation collection.
- As objects are moved to the old generation, that space will also eventually fill up and will need garbage collection
- This where GC algorithms have the biggest difference. The simpler algorithms stop all the application threads, find unused objects, free up the space and eventually compact. This process is called full GC and generally causes the long pause.
- On the other hand, it is possible though computationally complex to find unused objects while application threads are still running. CMS and G1 both take this approach.

What GC should I use?

- Choice of garbage collector will depend on the application
- In a JEE application, each individual request will be impacted in the following way.
  1. Individual requests will be impacted by long pause times. If the goal is to ensure minimal impact of pause time then choose concurrent collector
  2. If the average response time is all that matters and a few slow responses is not an issue, then you can go for throughput collectors
  3. The benefits of avoiding long pauses with concurrent collector comes at the expense of CPU usage

What GC should I use?

- in batch operation, the choice of GC is guided by following
  1. If enough CPU is available then the use of concurrent collector with small pauses will enable job to be done faster
  2. If CPU is limited then the extra CPU consumption by the GC will make the jobs slower

GC Algorithms

There are choices

- JVM comes with multiple implementations of garbage collectors. You can choose the one that suits you
- The various algorithms available are
  1. The Serial Collector
  2. The throughput Collector
  3. The CMS collector
  4. The G1 Collector

Serial Collector

- The serial collector is the simplest collector and is used by default in a client machine
- The serial collector uses a single thread to process the heap and stops all application for processing the heap
- During a full GC it will fully compact the old generation.
- The serial collector is enabled using -**XX:+UseSerialGC** flag

Throughput Collector

- This is the default collector for a server class machine
- The throughput collector uses multiple threads to process the young generation, which makes minor GCs much faster than the serial collector
- The collector can use multiple threads when collecting old generation. This can be enabled by -**XX:+UseParallelOldGC** flag.
- This collector stops all application threads when performing the minor and full GCs.
- This collector is enabled using -**XX:+UseParallelGC** -**XX:+UseParallelOldGc**

CMS Collector – Working with young generation

- This collector is designed to eliminate long pauses
- CMS stops all application threads during a minor GC, which it performs with multiple threads
  - It uses a different algorithm than the throughput collector. (-XX:+UseParNewGC)

CMS Collector – Working with old generation

- Instead of stopping the application for full GC, the CMS collector frequently analyses the old generation using one or more background threads and discards the unused objects
- This ensures there is no pause in application threads but comes at the expense of CPU availability to the application.

CMS Collector - Issues

- The fact that there is no compaction done by the background thread means that the memory can become fragmented over time
- The background thread may also not get a chance to analyze the memory if there is not enough CPU
- If the collector notices that the memory has become too fragmented or CPU too full, then it can revert to a Serial collector behaviour and does a clean and compact operation, before reverting to the concurrent background processing again.
- CMS is enabled by specifying flags - **XX:+UseConcMarkSweepGC -XX:+UseParNewGC**

G1 Collector – Young Generation

- The G1 (Garbage First) collector is designed to process large heaps (size greater than 4 GB) with minimum pauses.
- This collector divides heap into multiple regions. The young generation is also one of the regions.
- The young generation collection requires a full pause and is done in multiple threads.

G1 Collector – Old Generation

- The old generation collection is where things differ
- Since the old generation is divided into multiple regions, the collector performs garbage cleaning by moving survivors from one region to another
- This approach ensures that each region is compacted automatically and there is less chance of memory fragmentation.
- G1 is enabled by the flag -**XX:+UseG1GC**

Choosing a GC algorithm

How to make the choice?

- The choice of a GC algorithm depends on the application and what are its performance goals.
- The serial collector is useful for an application that takes up low memory, generally less than 100 MB
    - These scenarios are better of handled without multiple threads of the concurrent collectors.
- Most program though, would need to choose between the throughput collector and concurrent collectors

GC Algorithm for batch jobs

- For batch jobs, the pauses introduced by the collectors, particularly the full GC is a big concern.
- If extra CPU is available then the concurrent collector will give the application a performance boost.

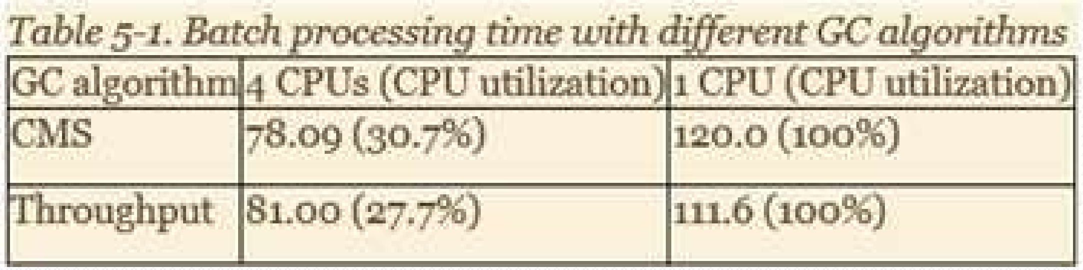


Table 5-1. Batch processing time with different GC algorithms

GC algorithm	4 CPUs (CPU utilization)	1 CPU (CPU utilization)
CMS	78.09 (30.7%)	120.0 (100%)
Throughput	81.00 (27.7%)	111.6 (100%)

- The table shows that the background threads introduced by the CMS takes up extra CPU time when there are 4 CPUs, but the job finishes 3 seconds faster
- While in case of a single CPU, the background thread tends to interfere with the application thread and this increased contention for CPU makes the processing slower

GC Algorithm for greater throughput

- The basic tradeoffs here are same as in the case of batch jobs but the effect of pause is quite different.

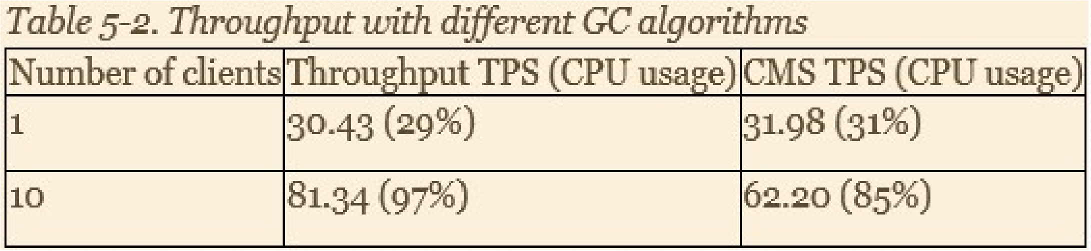


Table 5-2. Throughput with different GC algorithms

Number of clients	Throughput TPS (CPU usage)	CMS TPS (CPU usage)
1	30.43 (29%)	31.98 (31%)
10	81.34 (97%)	62.20 (85%)

- As can be seen when there is only one client, which leaves more CPU free, then the CMS collector performs better.
- But when there are more clients and consequently lesser CPU available, then the fact that background threads contend for the CPU time means the throughput is lesser

Between CMS and G1

- As a general rule of thumb, the CMS performs better when the memory footprint is smaller. Less than 4GB
- G1 performs better in case of heap size greater than 4GB.

Basic GC Tuning

Sizing the heap

- The first basic tuning is the size of the heap
- The right heap size is a matter or balance.
  - If the size is too small, then the program will spend too much time performing GC and not enough time performing application logic
  - On the other hand, if the size of the heap is too large then the pauses will be less frequent, but each pause will take that much more time.

Why not make it very large?

- Apart from the fact that GC will take a really long time, there is issue of virtualization
- In case of large size memory, the OS may choose to allocate some of that in the virtual memory
- This is ok for memories of multiple application since they are not used at the same time, but for JVM this is a problem as there can be multiple threads accessing the objects in memory.
- The single most important rule in heap sizing is not to specify it more than the available physical memory.
  - If there are multiple JVMs on the machine then this applies to the sum of all their heap sizes.

Controlling Heap Size

- The size of the heap is controlled by two parameters
  - Minimum specified by (-XmsN)
  - Maximum specified by (-XmxN)
- The default depends on the operating system, the amount of system RAM and the JVM in use.

Why the two parameters?

- Having an initial and maximum size allows JVM to tune its behaviour depending on the workload
- If the JVM sees that it is doing too much GC, then it will increase the heap until JVM does the "correct" amount of GC or until maximum size is hit.
- If the configured heap size is not providing adequate performance then the size can be adjusted using the parameters.

How much should we set the value to?

- A good rule of thumb is to size the heap to be 30% more than the memory utilization after a full GC
- This can be achieved by setting up the application to reach its steady state
- Then connecting using a profiler tool and performing a full GC.
- Observe how much memory is consumed after the full GC. Determine your values based on that

Sizing the generations

- Once the heap size has been determined, it will have to be determined how much of it has to be provided for the young generation and how much for the old.
- If young generation is too big then it will fill up slowly and the GC will run less frequently
- On the other hand, because the older generation is smaller, it will fill up faster and the full GC will run that often
- Striking a balance is key.

How to configure?

- Different GC algorithms try to strike this balance in different ways. All of them use the same set of flags to achieve this
  - -XX:NewRatio=N. Default is 2

    Sets the ration of new generation to old generation
  - -XX:NewSize=N

    Sets the initial size of the young generation
  - -XX:MaxNewSize=N

    Sets the maximum size of the young generation
  - -XmnN

    Shorthand for setting both newsize and maxsize to the same amount

How are parameters used?

- The young generation is first sized using the NewRatio.
- The formula used is

Initial Young Gen Size = Initial Heap Size / (1+NewRatio)

- If NewSize is set then it takes precedence over the one calculated using this formula
- As the heap expands, the young generation size will expand as well, until it reaches the max new size.
- Tuning using minimum and maximum young generation size is difficult. If an application needs dynamically sized heap then focus on setting the NewRatio.

Sizing Perm Gen Space / Metaspace

- When JVM loads classes, it must keep track of certain metadata about the classes
- This data is held in a separate heap space called the permgen (or permanent generation) in Java 7 and Meta Space in Java 8.
- In Java 7 perm gen contains some miscellaneous objects that are unrelated to the class data. These are moved into the regular heap in Java 8
- Either of these spaces don't hold the actial instance of the class object, neither the reflection Method object. These are stored in regular Heap.

How to configure?

- There isn't a good way to calculate how much perm gen space a particular program will need
- It will be proportional to the number of classes it uses.
- Larger the application, larger should be the size of the perm gen space.
- The sizes are specified via the flags
  - -XX:PermSize=N
  - -XX:MaxPermSize=N
  - -XX:MetaSpaceSize=N
  - -XX:MaxMetaSpaceSize=N
- The spaces are sized dynamically based on the initial size and increase as needed to a maximum size.

Controlling Parallelism

- All GC algorithms except the Serial collector uses multiple threads.
- The number of these threads is controlled by -**XX:ParallelGCThreads=N** flag.
- The value of this flag effects the number of threads created for the following operations
  - Collection of young generation when using -XX:+UseParallelGC
  - Collection old generation when using -XX:+UseParallelOldGC
  - Collection of young generation when using -XX:+UseParNewGC
  - Collection of the young generation when using -XX:+UseG1GC

How many threads should I run?

- Because the GC operations stop the application threads from executing the JVM attempts to use as many CPU resources as it can in order to minimize the pause time
- By default the JVM will run one thread for each CPU on the machine, up to eight.
- Once it has reached 8 threads and there are more than 8 CPUs, then it allocates one new thread for every 5/8 of a CPU.

Parallel GC Threads = 8 + ((N-8)*5/8)

What if there are multiple JVMs on a machine

- If there are multiple JVMs running on a machine, then it is a good idea to limit the number of GC threads among all the JVMs
- For example :
  - On a 16 CPU machine running 4 JVMs, each JVM will by default have 18 GC threads. This will result in the machine having 52 CPU hungry threads running in contention.
  - It might be better to provide each JVM with 4 GC threads.

Performance Enhancements in Java 8

Does Java 8 give better performance

- Java 8 has focused a lot on improving the Java language and adding modern features to it
- Its focus has not necessarily been on improving performance
- Still some new Java 8 features provide better performance

Concurrency utiities

- The Collections Framework has undergone a major revision in Java 8 to add aggregate operations based on the newly added streams facility and lambda expressions.
- As a result, the ConcurrentHashMap class introduces over 30 new methods in this release.
- These include various forEach methods (forEach, forEachKey,forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods (reduce, reduceToDouble, reduceToLong etc.)
- These utilities enable developer to write much more efficient algorithms to work with collections.
- Efficient algorithms enable compilers to optimize better

Tiered Compilation

- Tiered compilation introduced in Java 7
- It has been perfected and made default in Java 8

Meta Space

- Permament generation, where the VM stored internal native data structures, such as class information, has been replaced with metaspace.
- Metaspace eliminates java.lang.OutOfMemoryError: PermGen space errors.
- The java command introduces the option -**XX:MetaspaceSize**, which sets the maximum amount of native memory that can be allocated for class metadata
- In addition, the experimental option -Xshare has been introduced, which reduces footprint and startup time by sharing loaded JDK classes between all Java processes on the same host

Lambda Expressions

- Lambda expressions introduced in Java 8 perform better than the inner classes
- An Example

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```