

# Java Performance Tuning

## Specification

Provides API , standards, recommended practices, codes and technical publications, reports and studies.

JCP - Java Community Process

JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)

The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)

The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)

Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa      JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

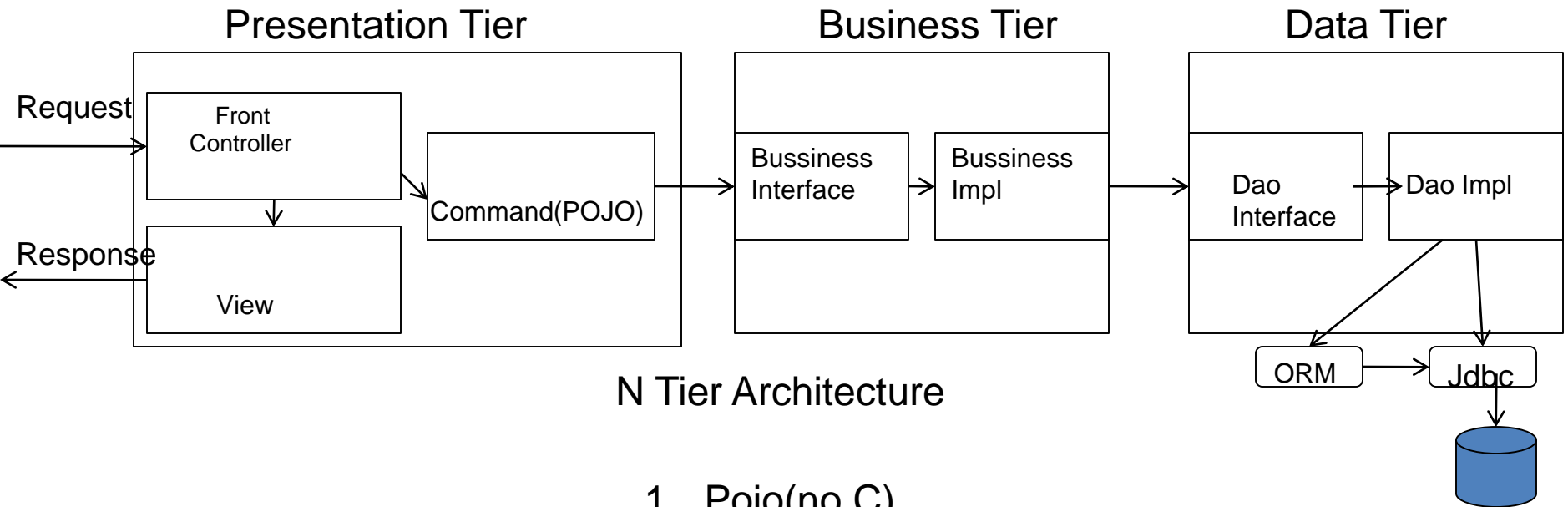
JSR 311 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)



1. Servlet/jsp
2. MVC
- Struts
- JSF
- Flex
- Gwt
- Spring MVC
- ...

> Any MVC + Spring

1. Pojo(no C)
2. Ejb 2.x(HW C)
  - Session Bean
  - Mdb
3. Pojo + LW Containe
  - Spring
  - Microcontainer
  - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
  - Hibernate
  - Kodo
  - Toplink
  - MyBatis
5. JPA

+ Spring Templates

# Design Patterns

# GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
SCOPE	Class	Factory Method	Class Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



# **Performance - Tuning**

Define the requirements



*Specific* – target a specific area for improvement.

*Measurable* – quantify or at least suggest an indicator of progress.

*Attainable* – *A goal is* realistic and attainable. .

*Realistic* – state what results can realistically be achieved, given available resources.

*Time-related* – specify when the result(s) can be achieved.

## Specific

The criterion stresses the need for a specific goal rather than a more general one. A specific goal will usually answer the five "W" questions:

What: What do I want to accomplish?

Why: Specific reasons, purpose or benefits of accomplishing the goal.

Who: Who is involved?

Where: Identify a location.

Which: Identify requirements and constraints.

## **Measurable**

The second criterion stresses the need for concrete criteria for measuring progress toward the attainment of the goal.

A measurable goal will usually answer questions such as:

How much?

How many?

How will I know when it is accomplished?

Indicators should be quantifiable

## **Attainable**

The third criterion stresses the importance of goals that are realistic and attainable. While an attainable goal may stretch a team in order to achieve it, the goal is not extreme.

An attainable goal will usually answer the question:

How: How can the goal be accomplished?

## **Relevant**

The fourth criterion stresses the importance of choosing goals that matter

A relevant goal can answer yes to these questions:

Does this seem worthwhile?

Is this the right time?

Does this match our other efforts/needs?

Are you the right person?

Is it applicable in current socio- economic- technical environment?

## **Time-bound**

The fifth criterion stresses the importance of grounding goals within a time frame, giving them a target date.

A time-bound goal is intended to establish a sense of urgency.

A time-bound goal will usually answer the question:  
When?

What can I do six months from now?

What can I do six weeks from now?

What can I do today?

**Measure don't guess**

Need a tool to measure and optimize your code?

Some of the good tools are :

➤ Jstat

➤ JMC

➤ Jvisualvm

➤ Jconsole

➤ Jprofiler

➤ Dynatrace

➤ AppDynamics

➤ Your Kit Java Profiler



**One thing at a time**

Don't think that having lots of developers optimize code will lead to better performance.

Wrong! Changing lot of things in parallel can have strange results.

Developer A could have done a good optimization but at the same time developer B deploys code that makes things worst

- so nobody will see the optimization of developer A.

**Automate**

Most of the times, developers have no time for performance testing and analysis in most cases. Especially because every change can lead to a decrease of performance.

Therefore It is recommend to automate the process of analyzing performance.

What you need is a Continuous Delivery Pipeline which includes profiling and performance analysis of acceptance and load tests and a performance report for each build/release – including comparison of performance metrics for two builds.

.

**Only optimize if needed**

When is tuning needed? The answer is: When the requirements are not met. What this tip means is that we should stop tuning if the requirements are met.

In many cases there is a rule that the more you optimize your application, the harder it is and the more you have to change your code.

In many cases “better performance” therefore also is in conflict with “better maintainability”. So stop tuning when the requirements are met – even if you have the ambition to optimize the application to the limit.

**Learn to parallelize**

With more CPUs on the boards that have more and more Cores we have to develop applications that are able to utilize these architecture.

Knowledge of Threads, Concurrency, Locking etc. will get most performance and throughput out of a system.

Sometimes it can also be interesting to look at specialized programming languages like Scala for better built-in concurrency support.



**Learn to scale**

Vertical scalability (or **scale up**) is maybe the easiest way to scale a system – you just put more hardware into one box. But this approach is limited, as normally you cannot put unlimited CPUs and RAM into one box. In many cases (like mainframes) this is also a very expensive approach.

Horizontal scalability (or **scale out**) means to run the application on more than one box. This approach is more scalable and cheaper but also more complex from an application point of view: We have to deal with clustering, replication of data and new approaches to store and scale data.

✓ **Virtualization or Cloud** infrastructure can be even more complicated because the scale out process is automated and bidirectional – which means that hardware is not only added on demand but also removed if not needed.

✓ **Map/Reduce frameworks** like Hadoop are kind of a mixture of parallelism and scalability for an application, because you split your application logic into pieces that can be run in parallel on many nodes. This can also help to improve performance and scalability of some problem domains.

✓ **Microservices :**

Microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

**Cache it**

Memory is cheap these days and retrieving data from disks or via network (distributed systems) is still expensive. So caching is one of the things you have to keep in mind when you want to get performance.

Ex : Hibernate L1 & L2 Caches, WebServer Cache

# **Right Abstraction**

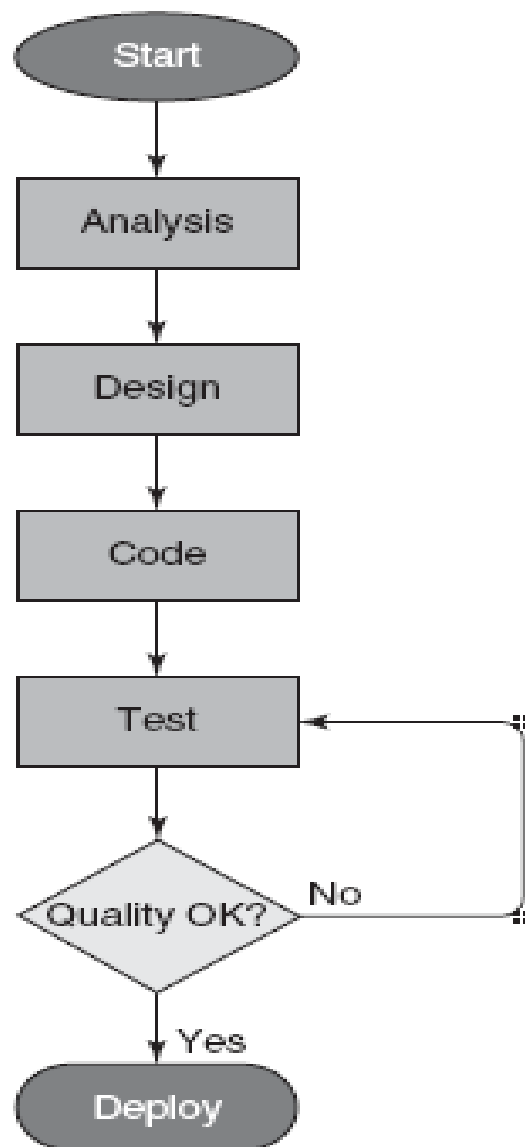
This is also a simple rule: Don't be too abstract in your application design.

E.g. don't create a layer above your database that abstracts away all special features of Oracle/DB2/MSSQL/.....this will make your application slow

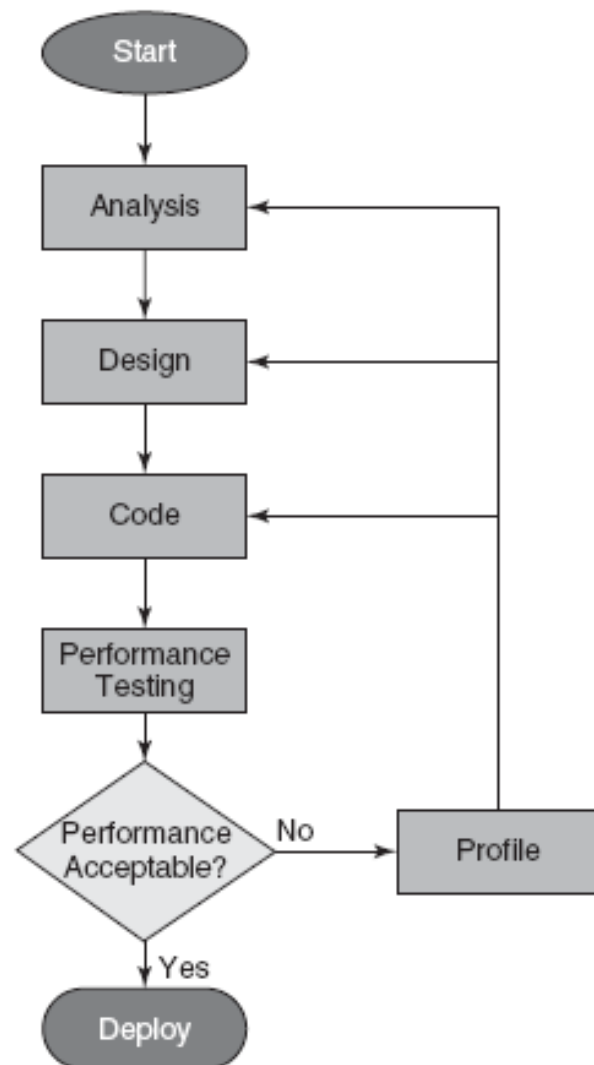
Object-oriented Business Model, Legacy systems, Batch processing & Report generation Systems

# **Traditional Software Development Process**





# **Performance Process Software Development**

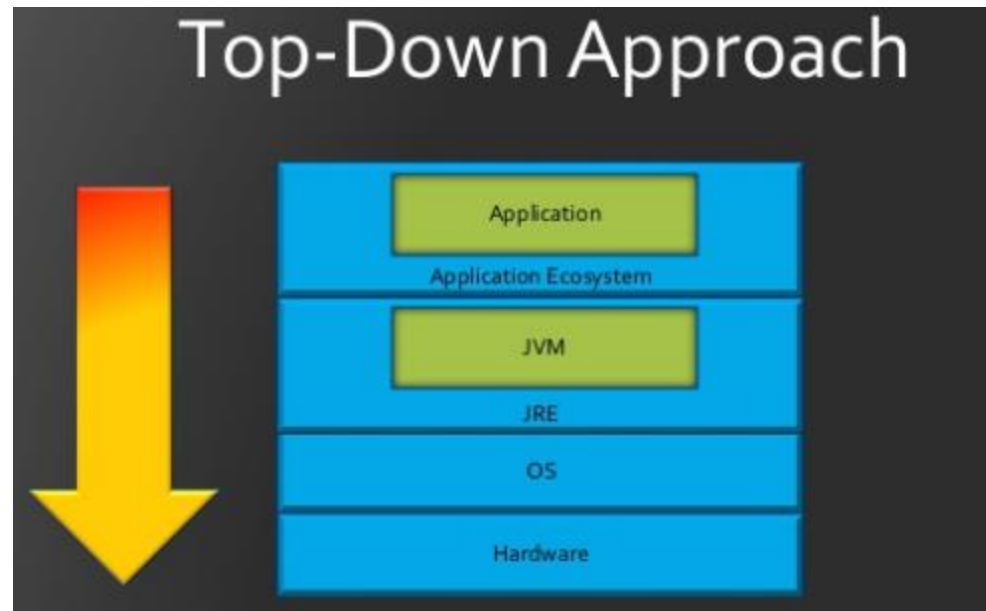


The following list is an example of the types of questions these requirements should answer:

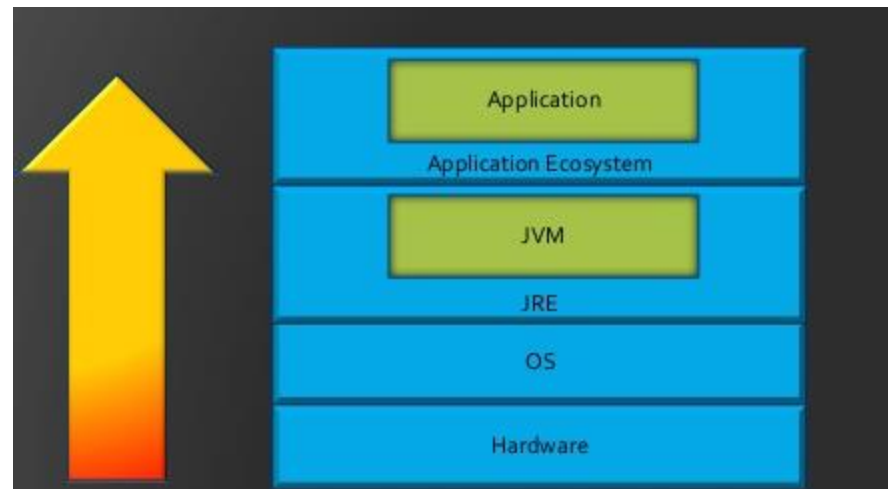
- ✓ What is the expected throughput of the application?
- ✓ What is the expected latency?
- ✓ How many concurrent users or concurrent tasks shall the application support?
- ✓ What is the accepted throughput and latency at the maximum number of concurrent users or concurrent tasks?
- ✓ What is the maximum worst case latency?
- ✓ What is the frequency of garbage collection induced latencies that will be tolerated?

## **Two Approaches, Top Down and Bottom Up**

Top down, as the term implies, focuses at the top level of the application and drills down the software stack looking for problem areas and optimization opportunities.



Bottom up begins at the lowest level of the software stack, at the CPU level looking at statistics such as CPU cache misses, inefficient use of CPU instructions, and then working up the software stack at what constructs or idioms are used by the application.



# **Principles of Performance Tuning**



## The Principles of Java Application Performance Tuning :

To fully tune a Java application, we need at least a basic level of understanding of:

- ✓ Hardware;
- ✓ OS processes
- ✓ The JVM
- ✓ Garbage collection
- ✓ JIT compilation
- ✓ Locks
- ✓ Concurrency
- ✓ Class loading
- ✓ Object creation

One procedure for Java performance tuning is to repeatedly:

- ✓ Specify target performance
- ✓ Specify the JVM configuration(s)
- ✓ Check that OS CPU
- ✓ memory, and IO are acceptable, or tune
- ✓ Check that response times are acceptable, or tune
- ✓ Check that throughput is acceptable, or tune

After any change from tuning, start again from the beginning of this sequence.

✓ Throughput and response times often impact each other, tuning to optimize one frequently adversely affects the other, so we need to balance to get the overall best performance.

✓ Currently the CMS is the best collector for low pause times

**What is application throughput?**

- ✓ Throughput is the way of quantifying the volume of requests/responses in relation to time.
- ✓ Transactions per second or TPS is the most common ratio used.
- ✓ A performance test plan usually contains certain throughput goals.
- ✓ Often we see the workload to a web application measured by Throughput

**What is Latency (Responsiveness)?**

- ✓ Latency is the amount of time a message takes to traverse a system.
- ✓ In a computer network, it is an expression of how much time it takes for a packet of data to get from one designated point to another.
- ✓ It is sometimes measured as the time required for a packet to be returned to its sender.

Performance requirements should be defined SMART - like “95% of the Login requests should respond in less than 2 seconds measured on the web server”.



# **The JVM Architecture**

**JRE** is the implementation of **Java Virtual Machine** (JVM)

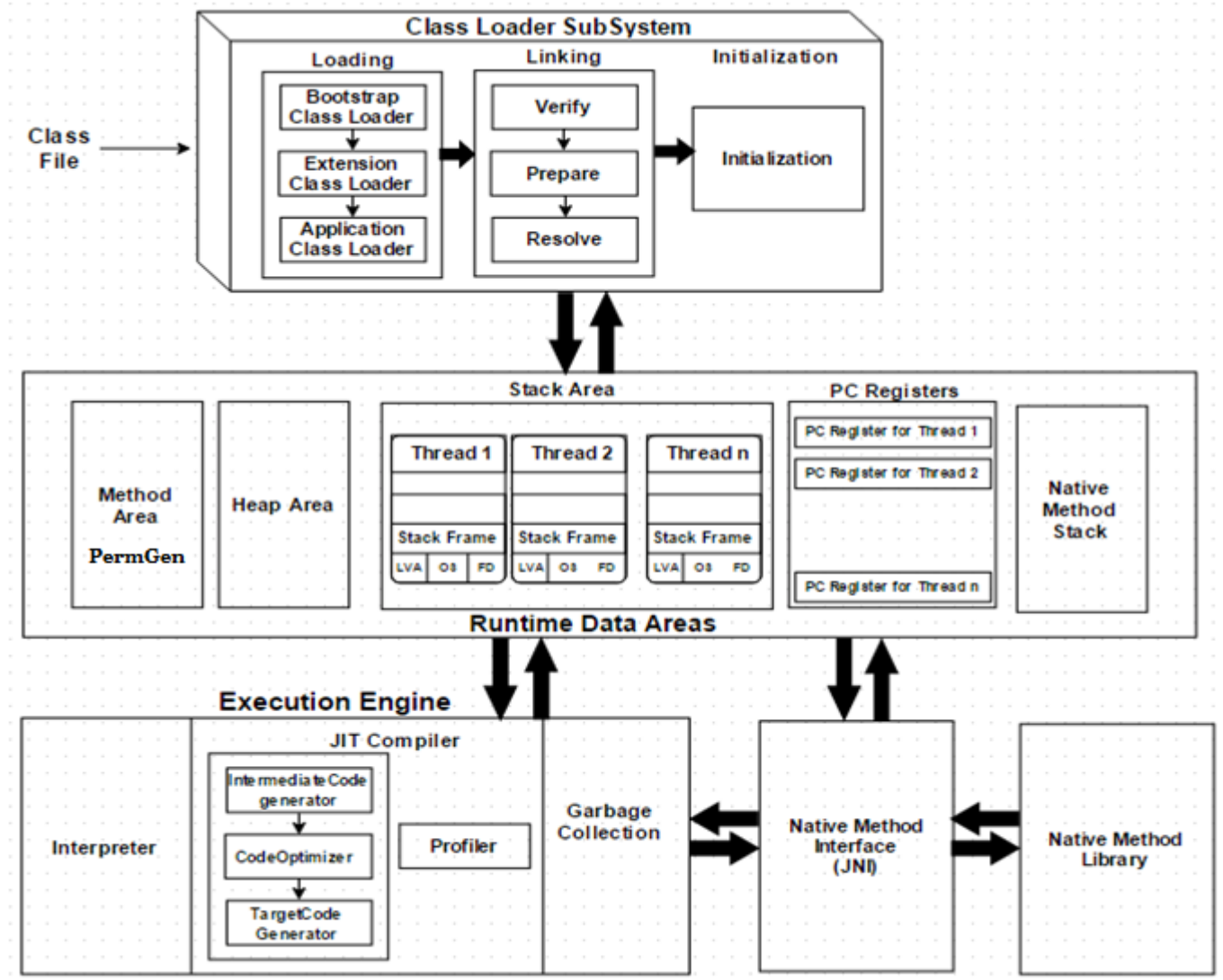
- ✓ which analyzes the bytecode
- ✓ interprets the code
- ✓ and executes it

## **What is the JVM?**

A Virtual Machine is a software implementation of a physical machine. Java was developed with the concept of WORA (Write Once Run Anywhere), which runs on a VM.

The compiler compiles the Java file into a Java .class file, then that .class file is input into the JVM, which Loads and executes the class file.

# JVM Architecture Diagram



## **How Does the JVM Work?**

JVM is divided into three main subsystems:

- ✓ Class Loader Subsystem
- ✓ Runtime Data Area
- ✓ Execution Engine

## **Class Loader Subsystem**

Java's dynamic class loading functionality is handled by the class loader subsystem.

It loads, links, and initializes the class file when it refers to a class for the first time at runtime, not compile time.

## **Class Loading**

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

**Boot Strap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.

**Extension ClassLoader** – Responsible for loading classes which are inside ext folder (jre\lib\ext).

**System ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable etc.

The above Class Loaders will follow Delegation Hierarchy Algorithm.

Note : Use OSGI/ Java Dynamic Modules

## Linking

**Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

**Prepare** – For all static variables memory will be allocated and assigned with default values.

**Resolve** – All symbolic memory references are replaced with the original references from Method Area.



## **Initialization**

This is the final phase of Class Loading, here all static variables will be assigned with the original values, and the static block will be executed.

## Runtime Data Area

The Runtime Data Area is divided into 5 major components:

**Method Area (Metaspace)**— All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread safe.

**Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory

**PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

**Native Method stacks** – Native Method Stack holds native method information.

## **Execution Engine**

The bytecode which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

**Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

**JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.

**Intermediate Code generator** – Produces intermediate code

**Code Optimizer** – Responsible for optimizing the intermediate code generated above

**Target Code Generator** – Responsible for Generating Machine Code or Native Code

**Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

**Garbage Collector:** Collects and removes unreferenced objects. Garbage Collection can be triggered by calling "System.gc()", but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI):** JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries:** It is a collection of the Native Libraries which is required for the Execution Engine.

# GC Tuning

**Instead Of :**

Experimenting lot of GC-related JVM parameters??

Changing random parts of your application code??



**Following a simple process** will guarantee that we are actually moving towards the right target while being transparent about the progress:

- State your performance goals
- Run tests
- Measure the results
- Compare the results with the goals
- If goals are not met, make a change and go back to running tests

Performance goals in regards of Garbage Collection fall into three categories:

- Latency
- Throughput
- Capacity

Also, Reliable & Easy to use/tune

## **Latency (Responsiveness)**

Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:

How quickly a desktop UI responds to an event

How fast a website returns a page

How fast a database query is returned

For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

## **Latency** (Responsiveness)

Latency goals for the GC have to be derived from generic latency requirements.

Generic latency requirements are typically expressed in a form similar to the following:

- All user transactions must respond in less than 10 seconds
- 90% of the invoice payments must be carried out in under 3 seconds
- Recommended products must be rendered to a purchase screen in less than 100 ms

## **Throughput**

Throughput focuses on maximizing the amount of work by an application in a specific period of time.

Examples of how throughput might be measured include:

The number of transactions completed in a given time.  
The number of jobs that a batch program can complete in an hour.

The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.

## Throughput

Generic requirements for throughput can be similar to the following:

- The solution must be able to process 1,000,000 invoices/day
- The solution must support 1,000 authenticated users each invoking one of the functions A, B or C every five to ten seconds
- Weekly statistics for all customers have to be composed in not more than six hours each Sunday night between 12 PM and 6 AM

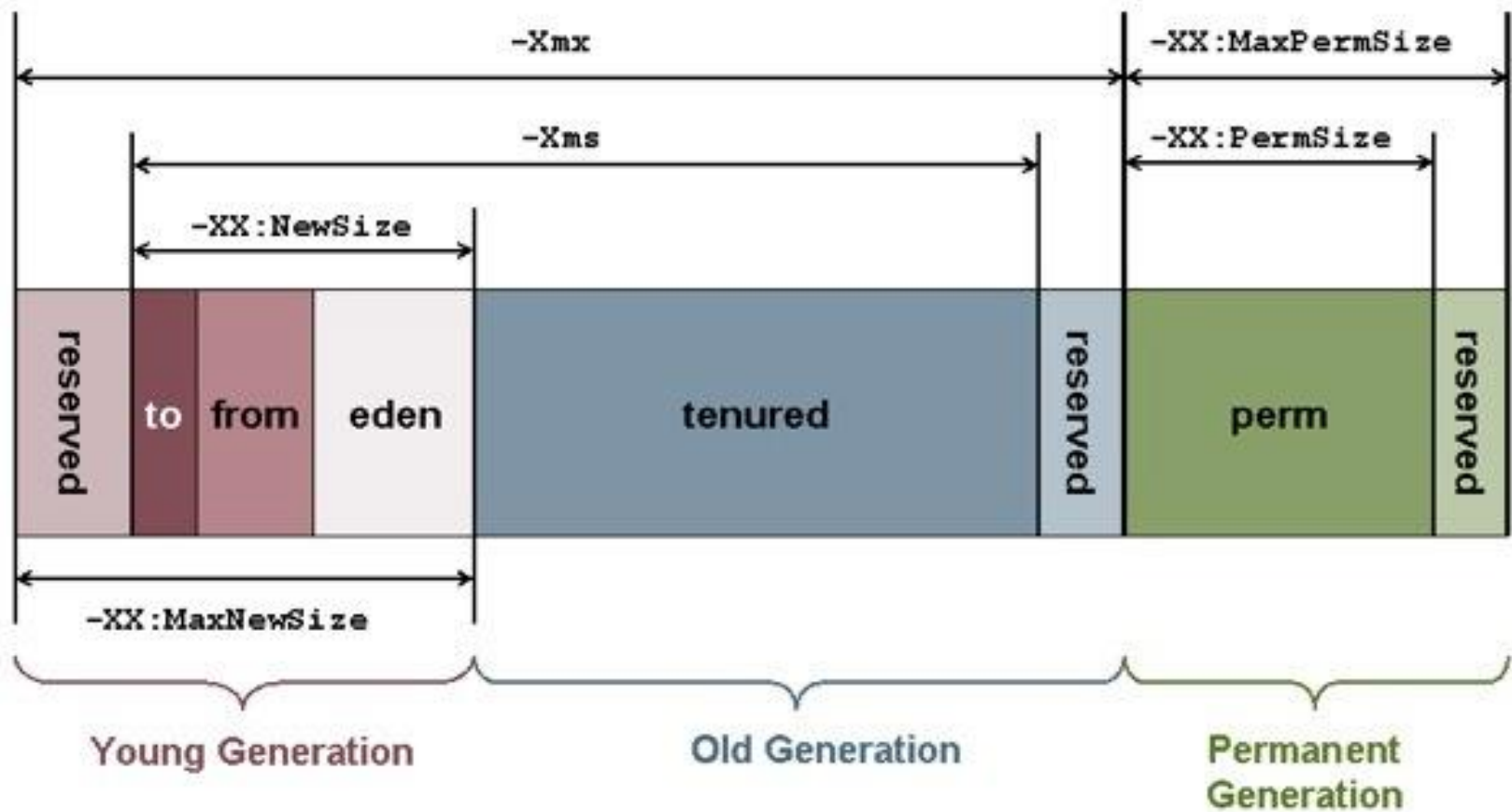
## Capacity

Capacity requirements put additional constraints on the environment where the throughput and latency goals can be met.

These requirements might be expressed either in terms of computing resources or in cold hard cash.

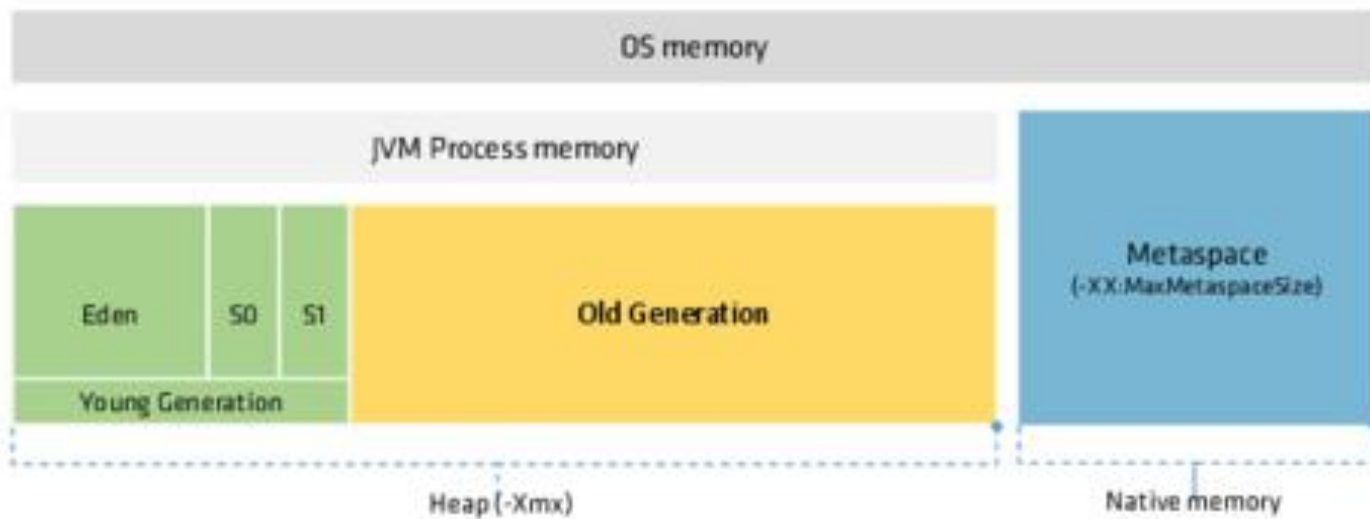
Examples are :

- The system must be deployed on Android devices with less than 512 MB of memory
- The system must be deployed on Amazon EC2. The maximum required instance size must not exceed the configuration model c3.xlarge (8 G, 4 cores)
- The monthly invoice from Amazon EC2 for running the system must not exceed \$12,000





# JVM memory JDK8



## Java 64bit environment

The primary advantage of running Java in a 64-bit environment is the larger address space. This allows for a much larger Java heap size and an increased maximum number of Java Threads, which is needed for certain kinds of large or long-running applications.

## **Large memory pages**

The default memory page size in most operating systems is 4 kilobytes (kb).

Large memory pages are pages of memory which are significantly larger than 4 kb, usually 2 Mb. In some instances it's configurable, from 2MB to 256MB.

To enable large page memory , use :

-XX:+UseLargePages

## **Generational Hypothesis**

- ✓ Most objects die young
- ✓ Old objects rarely reference young objects

## Generational Garbage Collector

- ✓ Split the heap into regions
- ✓ Create new objects in Young
- ✓ Move mature objects to Old

\* Different strategies for different regions



## Characteristics of GC's

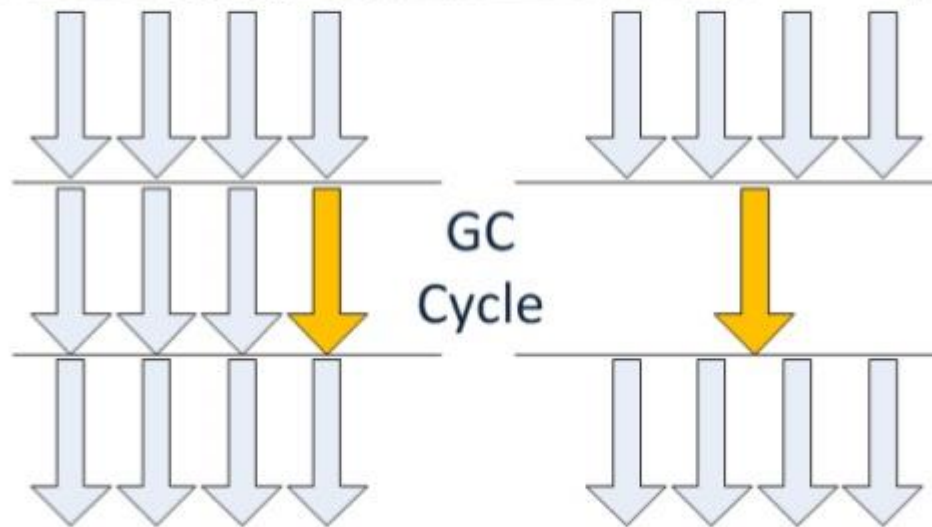
- ✓Concurrent

- ✓Parallel

- ✓Compacting

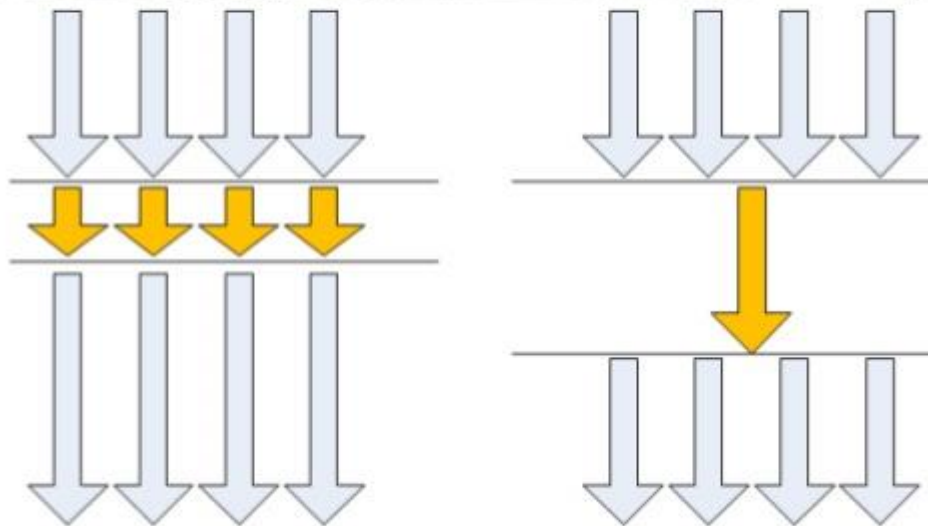
## Concurrent GC

---



## Parallel GC

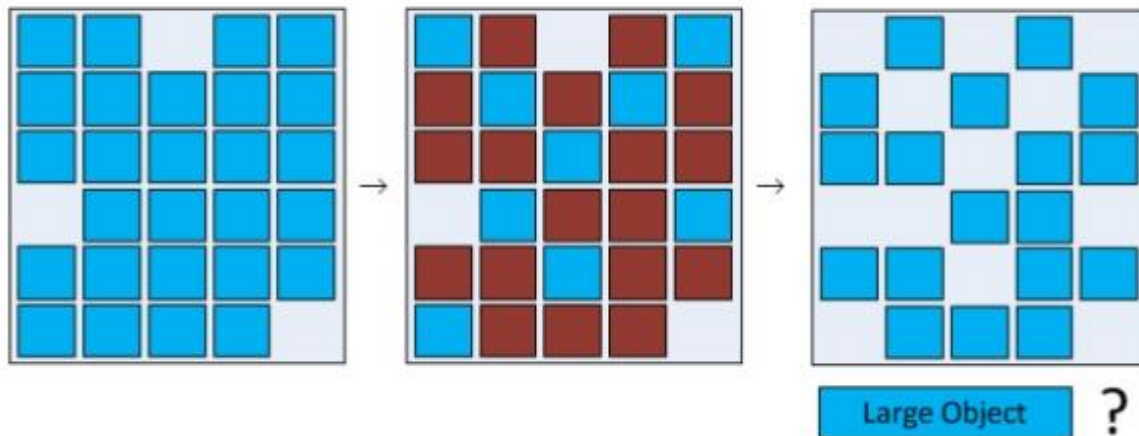
---





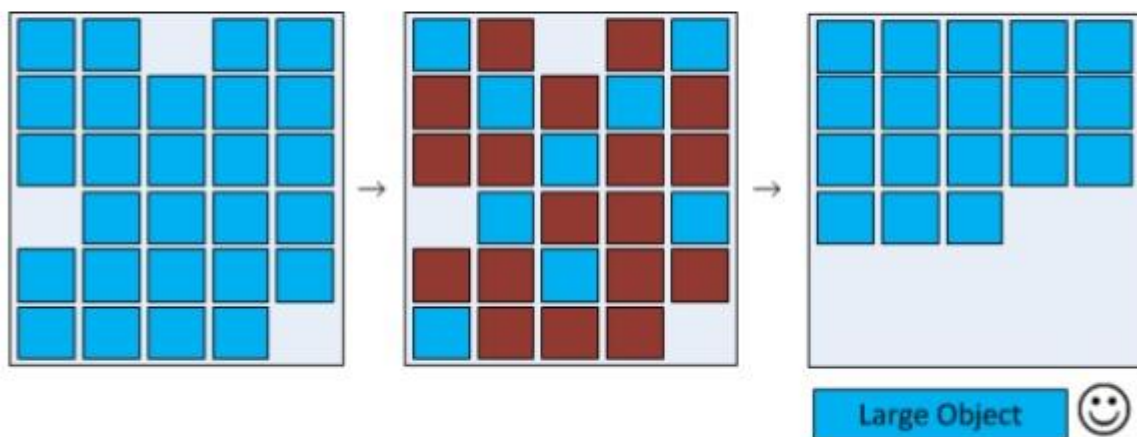
# Heap Fragmentation

---



# Heap Compacted

---



4 GC algorithms available in the Java Hotspot VM:

The **Serial GC** - recommended for client-style applications that do not have low pause time requirements.

The **Parallel GC** - use when the throughput matters.

The **Mostly-Concurrent GC** (also known as Concurrent Mark-Sweep GC(CMS)) - use when the latency matters.

The **Garbage First GC** (G1) - new GC algorithm, for CMS replacement.

## How to reach the goal:

To achieve the performance, Garbage Collection is to combine an understanding of the runtime requirements of the application, the physical characteristics of the application and the understanding of G1 to tune a set of options and achieve an optimal running state that satisfies the business requirements.

It's important to keep in mind that tuning is a constantly evolving process in which we establish a set of baselines and optimal settings through repetitive testing and evaluation.

There is no definitive guide or a magic set of options, we are responsible for evaluating performance, making incremental changes and re-evaluating until we reach our goals.

Check the default GC algorithm  
(using **java** command)

1. To check the default GC algorithm, issue the below command

```
> java -XX:+PrintCommandLineFlags -XX:+PrintGCDetails -version
```

2. Run the command with G1 collector

```
> java -XX:+PrintCommandLineFlags -XX:+PrintGCDetails  
-XX:+UseG1GC -version
```

Example :

```
set "JAVA_OPTS=-Xms1G -Xmx1G -XX:PermSize=256M -  
XX:MaxPermSize=256M -XX:+UseG1GC "
```

```
-XX:+UseParallelGC  
-XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC
```

## GC for the Old Generation

The old generation basically performs a GC when the data is full. The execution procedure varies by the GC type.

According to JDK 7, there are 5 GC types.

- Serial GC

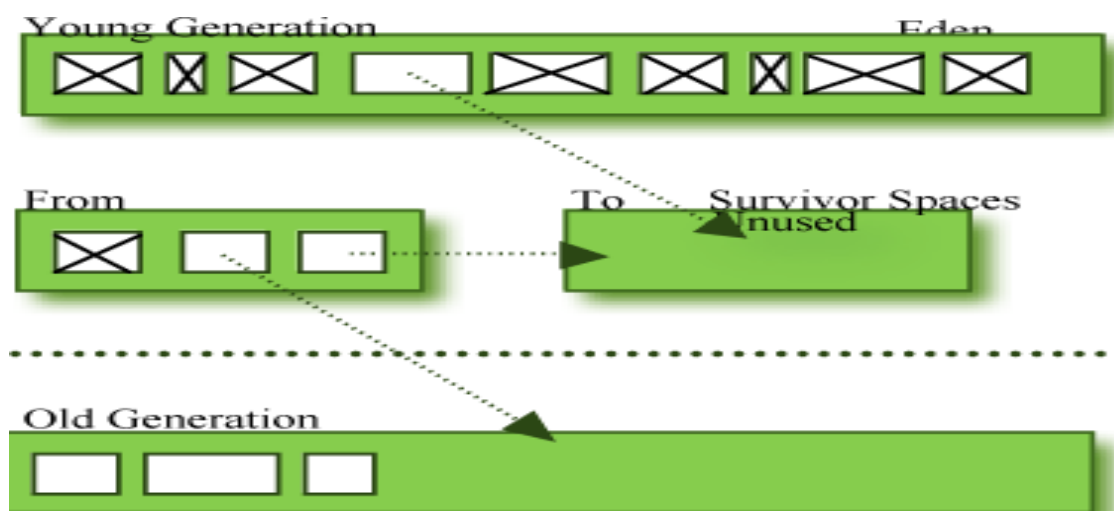
- Parallel GC

- Parallel Old GC (Parallel Compacting GC)

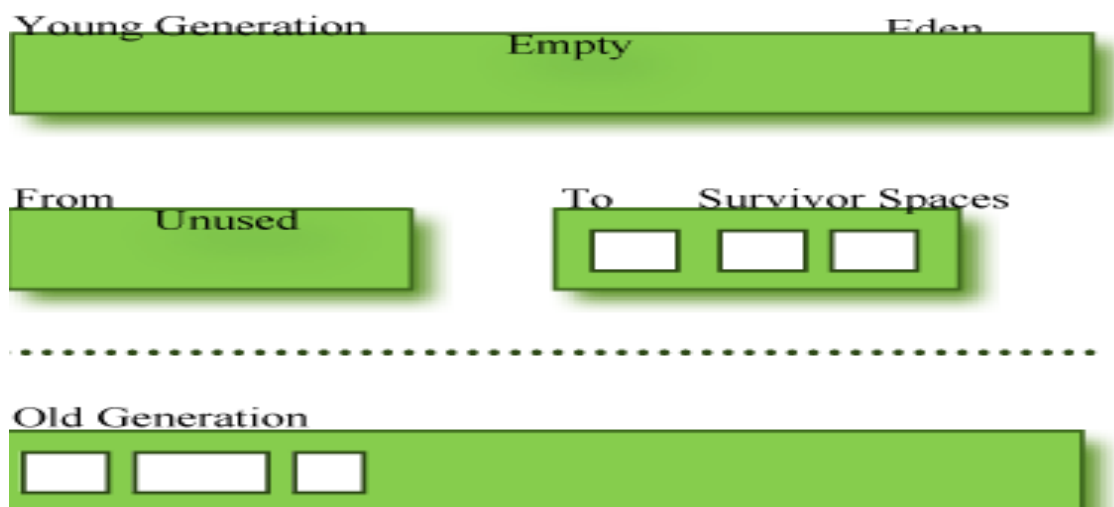
- Concurrent Mark & Sweep GC (or "CMS")

- Garbage First (G1) GC





After a GC



## **Major GC vs Full GC**

Major GC is cleaning the Tenured space.

Full GC is cleaning the entire Heap – both Young and Tenured spaces.

-> many Major GCs are triggered by Minor GCs, so separating the two is impossible in many cases.

-> On the other hand – many modern garbage collections perform cleaning the Tenured space partially.

Note : We need to focus on whether the GC at hand stopped all the application threads or was it able to progress concurrently with the application threads.

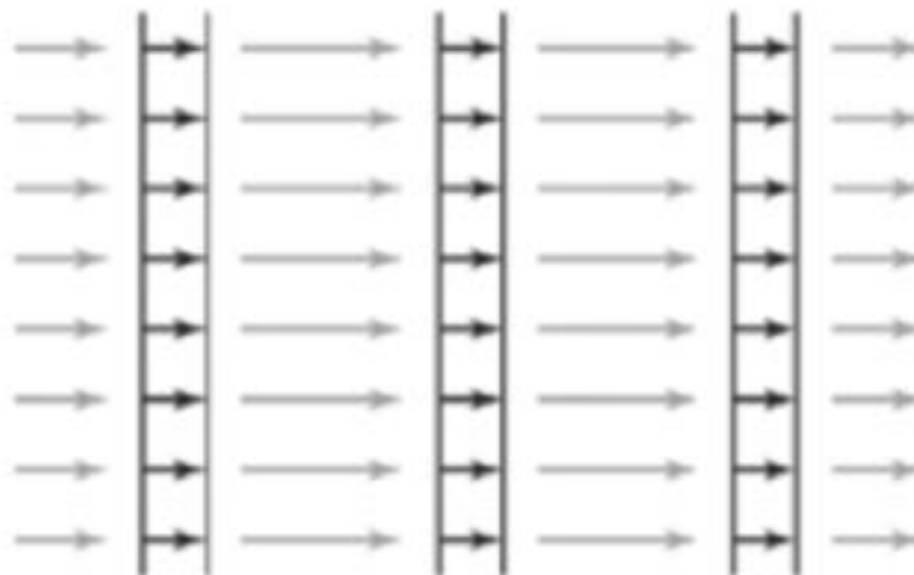
# **The Parallel Collector**

## The Parallel Collector

- ✓ It is also referred as "throughput collector"
- ✓ Multiple threads are used to speed up garbage collection.
- ✓ It is enabled with the command-line option `-XX:+UseParallelGC`.

By default, with this option, both minor and major collections are executed in parallel to further reduce garbage collection overhead.

Parallel GC



The number of garbage collector threads:

- ✓ For  $N \leq 8$  parallel GC will use just as many, i.e.,  $N$  GC threads.
- ✓ For  $N > 8$  available processors, the number of GC threads will be computed as  $3 + 5N/8$ .

`-XX:ParallelGCThreads`

The number of garbage collector threads can be controlled with the command-line option `-XX:ParallelGCThreads=<N>`.

## **Parallel Collector Ergonomics**

- ✓ The parallel collector is selected by default on server-class machines.
- ✓ In addition, the parallel collector uses a method of automatic tuning that allows us to specify specific behaviors instead of generation sizes and other low-level tuning details.
- ✓ We can specify maximum garbage collection pause time, throughput, and footprint (heap size).

## **Maximum Garbage Collection Pause Time:**

-XX:MaxGCPauseMillis=<N> is a hint that pause times of <N> milliseconds or less are desired;

## **Throughput:**

The throughput goal is measured in terms of the time spent doing garbage collection versus the time spent outside of garbage collection (referred to as application time). The goal is specified by the command-line option -XX:GCTimeRatio=<N>, which sets the ratio of garbage collection time to application time to  $1 / (1 + \text{<N>})$ .

For example, -XX:GCTimeRatio=19 sets a goal of  $1/(1+19)$  or 5% of the total time in garbage collection. The default value is 99, resulting in a goal of 1% of the time in garbage collection.



## **GC Ergonomics -> UseParallelGC with -XX:+UseAdaptiveSizePolicy**

UseParallelGC with -XX:+UseAdaptiveSizePolicy is the ergonomic (tries to grow or shrink the heap to meet the specified maximum pause time and/or throughput goal ) throughput collector.

The default goal is no pause time target and throughput with 98% of the time doing application work and 2% of the time doing GC work.

-XX:AdaptiveSizePolicyOutputInterval=1 which will print the ergonomics details every GC

By default a generation grows in increments of 20% and shrinks in increments of 5%.

The percentage for growing is controlled by the command-line flag

-XX:YoungGenerationSizeIncrement=<Y>

-XX:TenuredGenerationSizeIncrement=<T> for the tenured generation.

The percentage by which a generation shrinks is adjusted by the command-line flag

-XX:AdaptiveSizeDecrementScaleFactor=<D>.

If the growth increment is X percent, then the decrement for shrinking is  $X/D$  percent.

-XX:+UseAdaptiveSizePolicy consider three goals:

- ✓ a desired maximum GC pause goal
- ✓ a desired application throughput goal
- ✓ minimum footprint

The implementation checks (in this order):

If the GC pause time is greater than the pause time goal then reduce the generations sizes to better attain the goal.

If the pause time goal is being met then consider the application's throughput goal. If the application's throughput goal is not being met, then increase the sizes of the generations to better attain the goal.

If both the pause time goal and the throughput goal are being met, then the size of the generations are decreased to reduce footprint.

## **Explicitly setting the Generation sizes :**

To disable automatic sizing strategy, use the below option  
`-XX:-UseAdaptiveSizePolicy`

Now, we can explicitly set the generation sizes:

`-XX:NewSize=400m -XX:MaxNewSize=400m -XX:SurvivorRatio=6`

Maximum Garbage Collection Pause Time: The maximum pause time goal is specified with the command-line option `-XX:MaxGCPauseMillis=<N>`.

This is interpreted as a hint that pause times of <N> milliseconds or less are desired; by default, there is no maximum pause time goal.

If a pause time goal is specified, the heap size and other parameters related to garbage collection are adjusted in an attempt to keep garbage collection pauses shorter than the specified value.

These adjustments may cause the garbage collector to reduce the overall throughput of the application, and the desired pause time goal cannot always be met.

# GC Log

2016-03-21T13:46:58.371+0000: 500.203: [Full GC [5] YoungGen: 1047584K->0K(1048064K)] [ParOldGen: 2097151K->865378K(2097152K)] 3144735K->865378K(3145216K) [PSPermGen: 29674K->29634K(262144K)], 1.8485590 secs] [Times: user=3.17 sys=0.21, real=1.84 secs] 8

1 2016-03-21T13:46:58.371 – *Timestamp at which GC event ran*

2 500.203 – *Number of seconds since application started*

3 Full GC – *Type of GC*

4 PSYoungGen: 1047584K->0K(1048064K) – *Young Gen size dropped from 1047584K (i.e.1gb) to 0k. Total allocated Young Gen size is 1048064K*

5 ParOldGen: 2097151K->865378K(2097152K) – *Old Gen size dropped from 2097151K (i.e.1.99gb) to 865378K(i.e.845mb). Total allocated Old Gen size is 2097152k (i.e.2gb)*

6 3144735K->865378K(3145216K) – *overall heap size dropped from 3144735K (i.e.2.99gb) to 865378K (i.e.845mb)*

7 PSPermGen: 29674K->29634K(262144K) – *Perm Gen Size dropped from 29674K to 29634K. Overall Perm Gen Size is 262144K (i.e.256mb)*

8 [Times:user=3.17 sys=0.21, real=1.84 secs]

[GC 246656K->243120K(376320K), 0.0929090 secs]

[Full GC 243120K->241951K(629760K), 1.5589690 secs]

In the first line, 246656K->243120K(376320K) means that the GC reduced the occupied heap memory from 246656K to 243120K. The heap capacity at the time of GC was 376320K, and the GC took 0.0929090 seconds.

```
[GC  
  [PSYoungGen: 142816K->10752K(142848K)] 246648K-  
>243136K(375296K),  
  0,0935090 secs  
]  
[Times: user=0.55 sys=0,10, real=0.09 secs]
```

We have a young generation GC which reduced the occupied heap memory from 246648K to 243136K and took 0.0935090 seconds.

In addition to that, we obtain information about the young generation itself: the collector used as well as its capacity and occupancy. In our example, the “PSYoungGen” collector was able to reduce the occupied young generation heap memory from 142816K to 10752K.



## Java GC Times

This is the same concept that is applied in GC logging as well. Let's look at couple examples to understand this concept better.

### Example 1:

[Times: user=11.53 sys=1.38, real=1.03 secs]

In this example: 'user' + 'sys' is much greater than 'real' time. That's because this log time is collected from the JVM, where multiple GC threads are configured on a multi-core/multi-processors server. As multiple threads execute GC in parallel, the workload is shared amongst these threads, thus actual clock time ('real') is much less than total CPU time ('user' + 'sys').

### Example 2:

[Times: user=0.09 sys=0.00, real=0.09 secs]

Above is an example of GC Times collected from a Serial Garbage Collector. As Serial Garbage collector always uses a single thread only, real time is equal to the sum of user and system times.

## **VM related Problem**(sys time can be higher than user time??)

If your application is running in a virtualized environment, may be because of nature of the emulation sys time can be higher than user time.

Make sure virtualized environment is NOT OVERLOADED with too many environments and also ensure that there are adequate resources available in the Virtual Machine for your application to run

An OutOfMemoryError occurring as a result of a too small old generation space:

```
2010-11-25T18:51:03.895-0600: [Full GC  
[PSYoungGen: 279700K->267300K(358400K)]  
[ParOldGen: 685165K->685165K(685170K)]  
964865K->964865K(1043570K)  
[PSPermGen: 32390K->32390K(65536K)],  
0.2499342 secs]  
[Times: user=0.08 sys=0.00, real=0.05 secs]  
Exception in thread "main"  
java.lang.OutOfMemoryError: Java heap space
```

An OutOfMemoryError occurring as a result of a too small permanent generation space:

```
2010-11-25T18:26:37.755-0600: [Full GC  
[PSYoungGen: 0K->0K(141632K)]  
[ParOldGen: 132538K->132538K(350208K)]  
32538K->32538K(491840K)  
[PSPermGen: 65536K->65536K(65536K)],  
0.2430136 secs]  
[Times: user=0.37 sys=0.00, real=0.24 secs]  
java.lang.OutOfMemoryError: PermGen space
```

# GC(Allocation Failure)

```
2015-05-26T14:45:37.987-  
02001:151.1262:[GC(Allocation Failure) 151.126:  
[DefNew:629119K->69888K(629120K), 0.0584157  
secs]1619346K->1273247K(2027264K),0.0585007  
secs][Times: user=0.06 sys=0.00, real=0.06 secs]
```

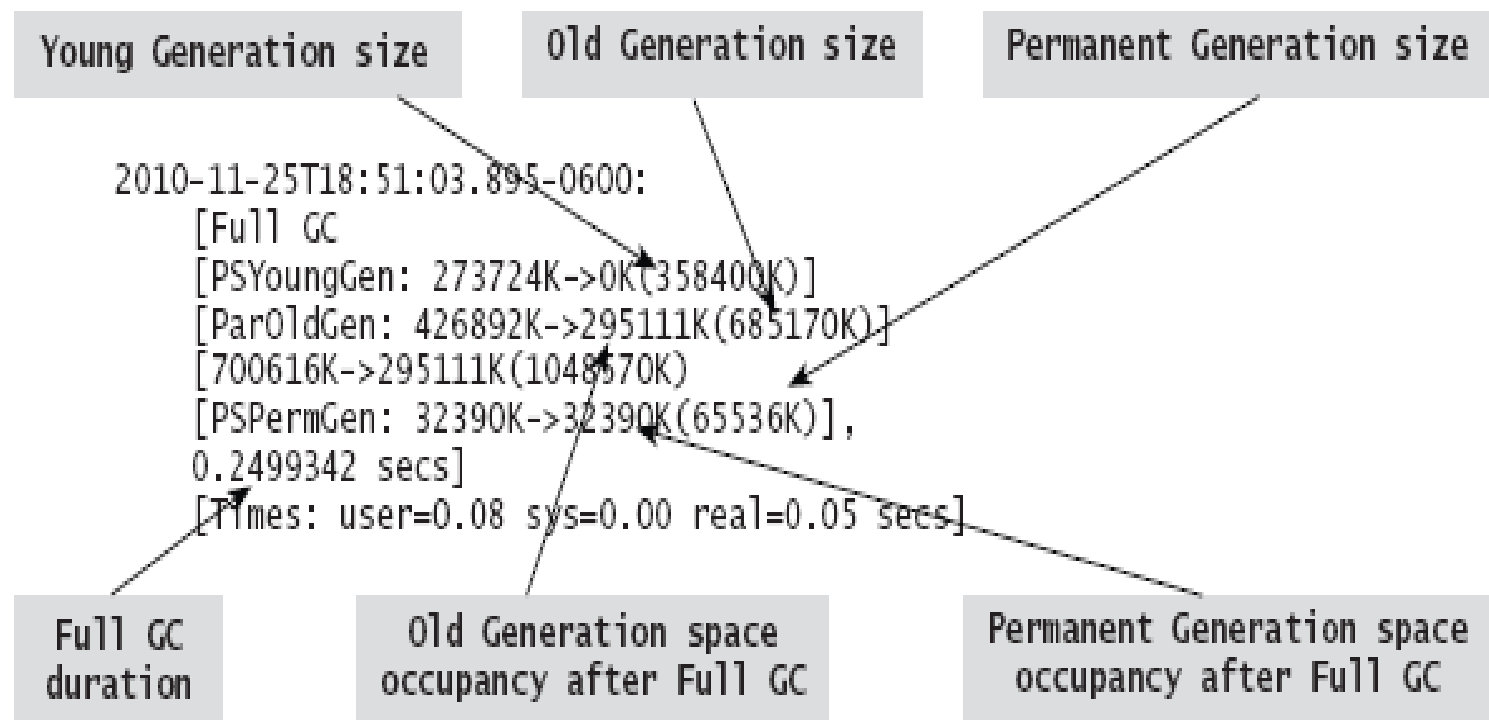
Allocation Failure – Cause of the collection. In this case, the GC is triggered due to a data structure not fitting into any region in Young Generation.

To force full garbage collections, monitor the application with VisualVM or JConsole and click the Perform GC button in the VisualVM or JConsole window.

A command line alternative to force a full garbage collection is to use the Hot-Spot JDK distribution jmap command

```
$ jmap -histo:live 348
```

The jmap command induces a full garbage collection and also produces a heap profile that contains object allocation information.



Garbage collection log after full GC event

# **Concurrent Mark Sweep (CMS) Collector**



The *Concurrent Mark Sweep* (CMS) collector is designed to be a lower latency collector (less pause time) than the parallel collectors.

The key part of this design is trying to do part of the garbage collection at the same time as the application is running.

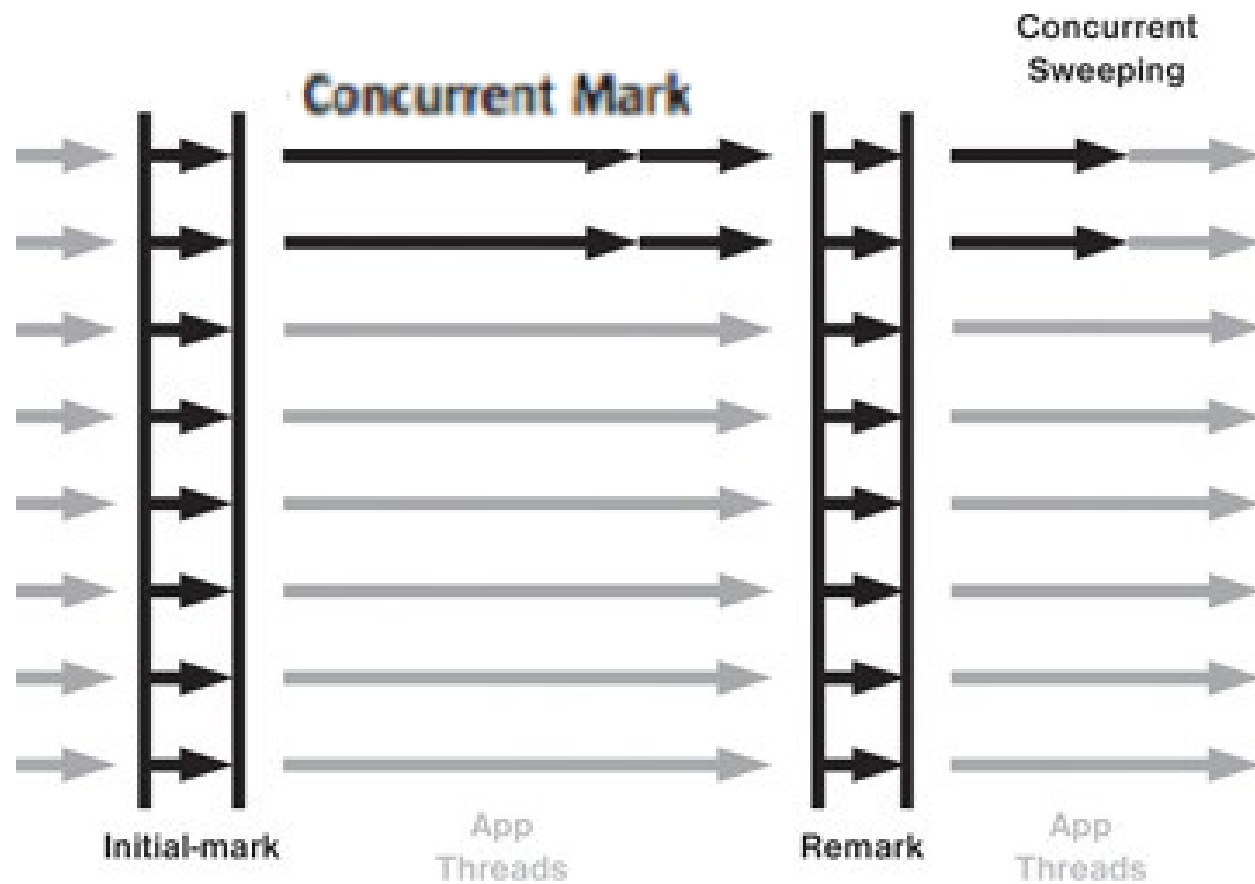
This means that when the collector needs to pause the application's execution it doesn't need to pause for as long.

Typically applications that have a relatively large set of long-lived data (a large tenured generation) and run on machines with two or more processors tend to benefit from the use of this collector.

The HotSpot command-line options to support multithreaded initial-mark and remark phases are :

- XX:+CMSParallelInitialMarkEnabled and
- XX:+CMSParallelRemarkEnabled.

These are automatically enabled by default when CMS GC is enabled by the -XX:+UseConcMarkSweepGC command-line option.

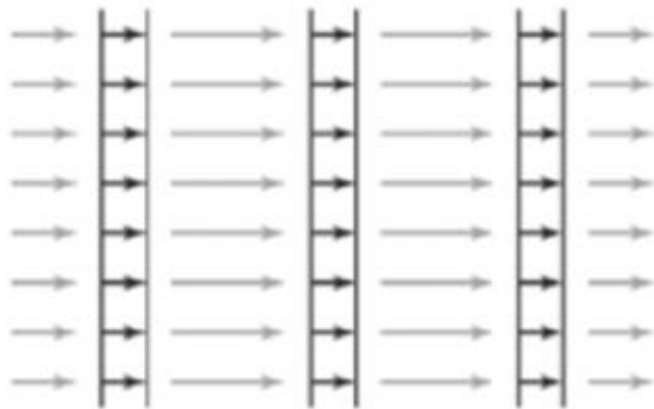


## Parallel GC vs Concurrent Mark Sweep collector:

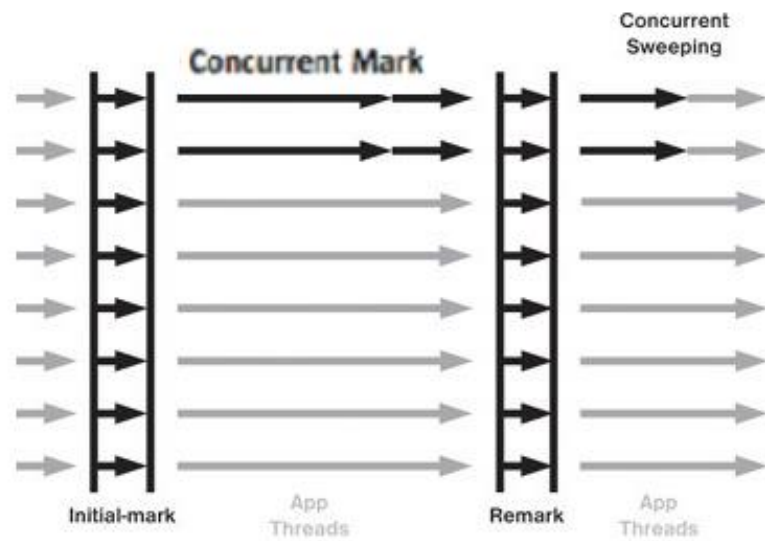
The parallel garbage collectors are designed to minimise the amount of time that the application spends undertaking garbage collection, which is termed *throughput*. This isn't an appropriate tradeoff for all applications - some require individual pauses to be short as well, which is known as a *latency* requirement.

The *Concurrent Mark Sweep* (CMS) collector is designed to be a lower latency collector (less pause time) than the parallel collectors. The key part of this design is trying to do part of the garbage collection at the same time as the application is running. This means that when the collector needs to pause the application's execution it doesn't need to pause for as long.

## Parallel GC



## Concurrent Mark-Sweep Collector



## **CMS GC (-XX:+UseConcMarkSweepGC)**

( “-XX:+UseParNewGC” flag activates the parallel execution of young generation )

*Initial mark* : The surviving objects among the objects the closest to the classloader are searched. So, the pausing time is very short. **Stop-the-world.**

*concurrent mark* : The objects referenced by the surviving objects that have just been confirmed are tracked and checked. The difference of this step is that it proceeds while other threads are processed at the same time.

*remark* step : The objects that were newly added or stopped being referenced in the concurrent mark step are checked. **Stop-the-world.**

***concurrent sweep* step: The garbage collection procedure takes place.**

The garbage collection is carried out while other threads are still being processed.

Note : It uses the parallel stop-the-world mark-copy algorithm in the Young Generation and the mostly concurrent mark-sweep algorithm in the Old Generation

## Phase 2 - **Initial mark**

During initial mark CMS should collect all root references to start marking of old space.

This includes:

*References from thread stacks,  
References from young space.*

References from stacks are usually collected very quickly (less than 1ms), but time to collect references from young space depends on size of objects in young space.

Normally initial mark starts right after young space collection, so Eden space is empty and only live objects are in one of survivor space.

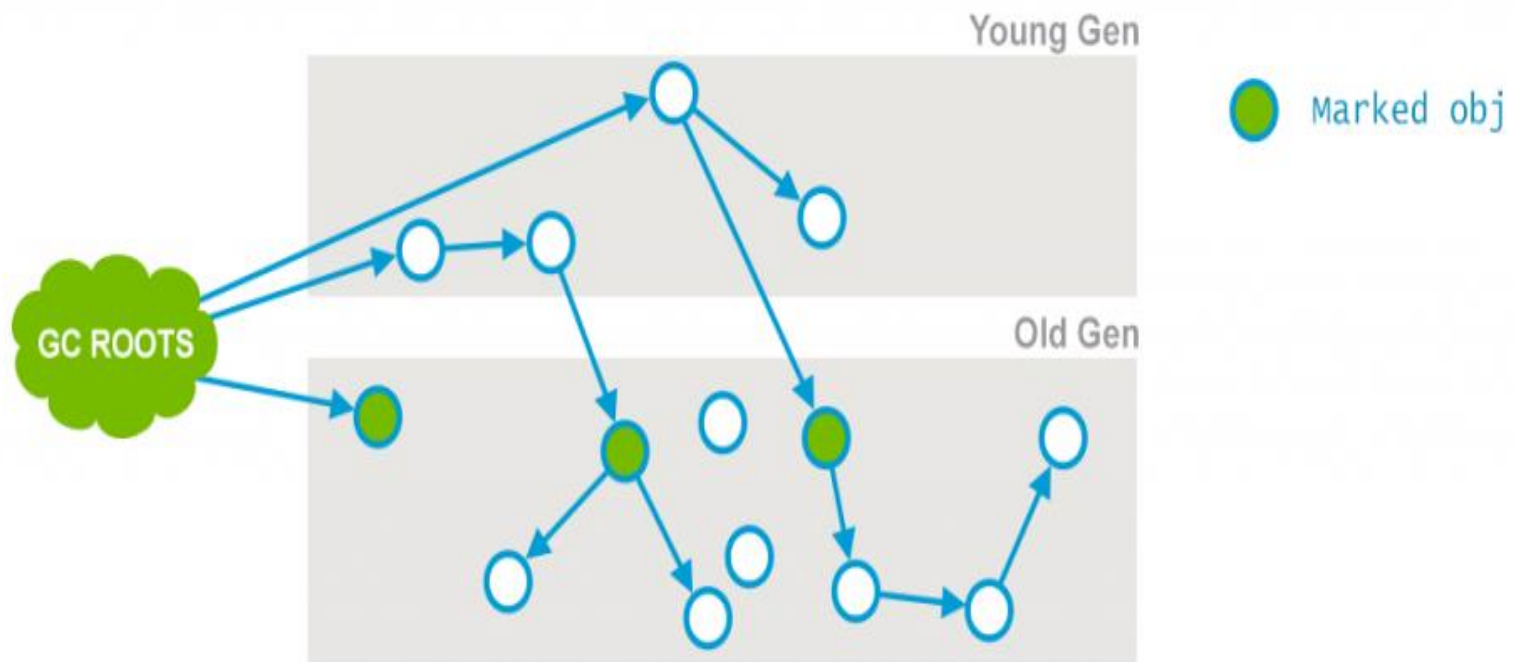
Survivor space is usually small and initial mark after young space collection often takes less than millisecond.



## Initial mark (continued)

But if initial mark is started when Eden is full it may take quite long (usually longer than young space collection itself). Once CMS collection is triggered, JVM may wait some time for young collection to happen before it will start initial marking.

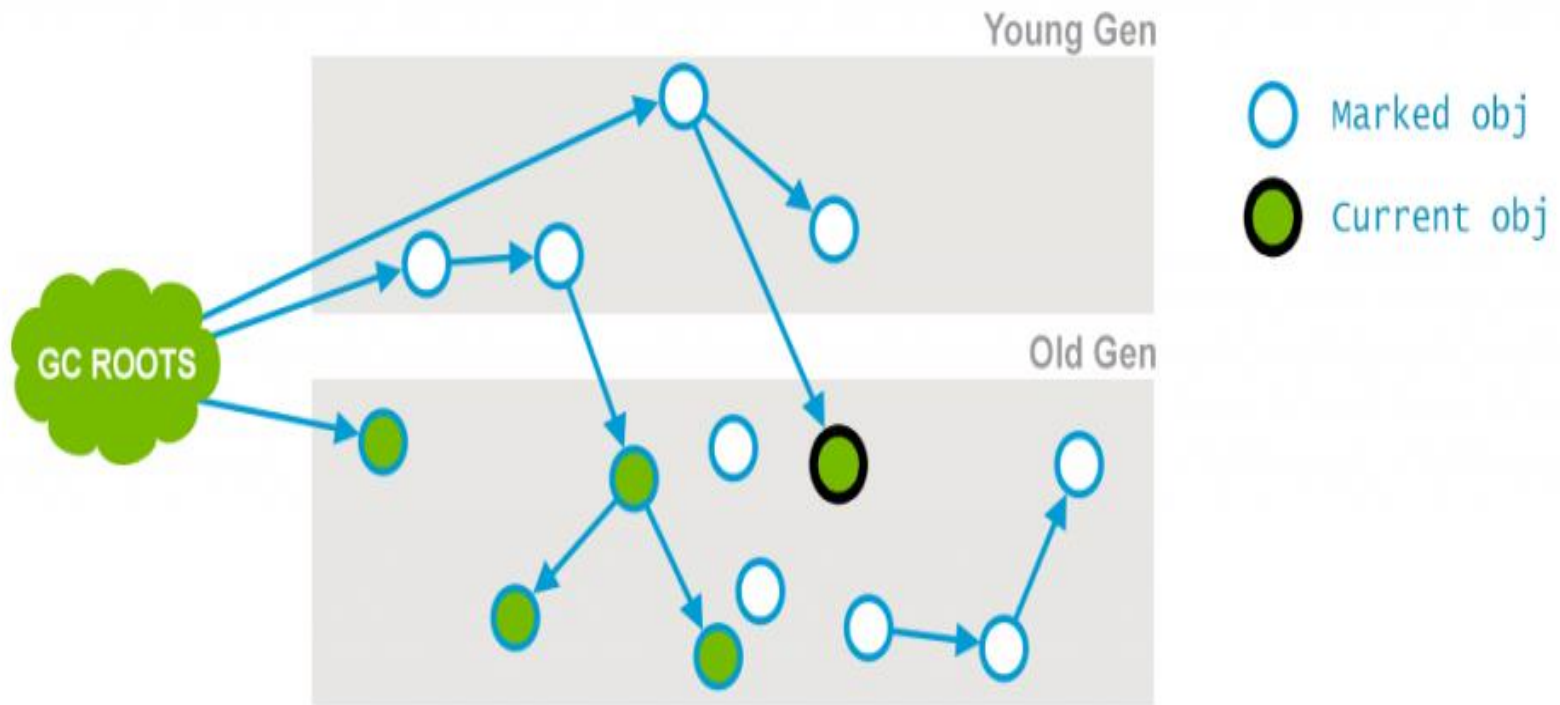
JVM configuration option **-XX:CMSWaitDuration=** can be used to set how long CMS will wait for young space collection before start of initial marking. If you want to avoid long initial marking pauses, you should configure this time to be longer than typical period of young collections in your application.



## **Phase 2: Concurrent Mark**

During this phase the Garbage Collector traverses the Old Generation and marks all live objects, starting from the roots found in the previous phase of “Initial Mark”.

The “Concurrent Mark” phase, as its name suggests, runs concurrently with the application and does not stop the application threads. Note that not all the live objects in the Old Generation may be marked, since the application is mutating references during the marking.



## Remark

Most of marking is done in parallel with application, but it may not be accurate because application may modify object graph during marking. When concurrent marking is finished; garbage collector should stop application and repeat marking to be sure that all reachable objects marked as alive. But collector doesn't have to traverse through whole object graph;

it should traverse only reference modified since start of marking (actually since start pre clean phase). **Card table** is used to identify modified portions of memory in old space, but thread stacks and young space should be scanned once again.

Usually most time of remark phase is spent of scanning young space. This time will be much shorter if we collect garbage in young space before starting of remark. We can instruct JVM to always force young space collection before CMS remark. Use JVM parameter **-XX:+CMSScavengeBeforeRemark** to enable this option.

## **-XX:CMSWaitDuration**

Normally objects are allocated in old space only during young space collection

But in certain cases object may be allocated directly in old space and CMS cycle could start while Eden has lots of objects. In this case initial mark can take more time.

Usually this is happening due to allocation of very large objects (few megabyte arrays). To avoid these long pauses we should configure reasonable -XX:CMSWaitDuration.

*-XX:CMSWaitDuration : Time in milliseconds that CMS thread waits for young GC (default is 2000ms)*

The pausing time for GC is very short. The CMS GC is also called the low latency GC, and is **used when the response time from all applications is crucial**.

While this GC type has the advantage of short stop-the-world time, it also has the following disadvantages.

It uses more memory and CPU than other GC types.

CMS is not a compacting collector, which over time can result in *old* generation fragmentation.

CMS collector does most of its tracing and sweeping work with the application threads still running, so only brief pauses are seen by the application threads.

However, if the CMS collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped.

The inability to complete a collection concurrently is referred to as concurrent mode failure and indicates the need to adjust the CMS collector parameters.



## **-XX:+UseConcMarkSweepGC**

This flag is needed to activate the CMS Collector in the first place. By default, HotSpot uses the Throughput Collector instead.

Note:

When the CMS collector is used, “-XX:+UseParNewGC” flag activates the parallel execution of young generation GCs using multiple threads.

## **-XX:+CMSConcurrentMTEnabled**

When this flag is set, the concurrent CMS phases are run with multiple threads (and thus, multiple GC threads work in parallel with all the application threads).

This flag is already activated by default.

If serial execution is preferred, which may make sense depending on the hardware used, multithreaded execution can be deactivated via -XX:-CMSConcurrentMTEnabled.

## **-XX:ParallelGCThreads**

-XX:ParallelGCThreads=<value> we can specify the number of GC threads to use for parallel GC

default value which is computed based on the number of available (virtual) processors.

The determining factor is the value N returned by the Java method `Runtime.availableProcessors()`.

For  $N \leq 8$  parallel GC will use just as many, i.e., N GC threads.  
For  $N > 8$  available processors, the number of GC threads will be computed as  $3 + 5N/8$ .

## **-XX:ConcGCThreads**

The flag -XX:ConcGCThreads=<value>

defines the number of threads with which the concurrent CMS phases are run.

The formula used is  $\text{ConcGCThreads} = (\text{ParallelGCThreads} + 3)/4$ .

## **-XX:CMSInitiatingOccupancyFraction**

This parameter sets the threshold percentage occupancy of the old generation at which the CMS GC is triggered.

The Throughput Collector starts a GC cycle only when the heap is full, i.e., when there is not enough space available to store a newly allocated or promoted object.

With the CMS Collector, it is not advisable to wait this long because if the application keeps on running (and allocating objects) during concurrent GC.

Thus, in order to finish a GC cycle before the application runs out of memory, the CMS Collector needs to start a GC cycle much earlier than the Throughput Collector.

set via `-XX:CMSInitiatingOccupancyFraction=<value>`

Note : the default value of CMSInitiatingOccupancyFraction is 68

## **-XX:+UseCMSInitiatingOccupancyOnly**

We can use the flag `-XX:+UseCMSInitiatingOccupancyOnly` to instruct the JVM not to base its decision when to start a CMS cycle on run time statistics.

Instead, when this flag is enabled, the JVM uses the value of `CMSInitiatingOccupancyFraction` for every CMS cycle, not just for the first one

## **-XX:+CMSClassUnloadingEnabled**

In contrast to the Throughput Collector, the CMS Collector does not perform GC in the permanent generation by default.

If permanent generation GC is desired, it can be enabled via -XX:+CMSClassUnloadingEnabled

`-XX:+CMSIncrementalMode` (**deprecated** in JDK 8)

This flag activates the incremental mode of the CMS Collector.

Incremental mode pauses the concurrent CMS phases regularly, so as to yield completely to the application threads. As a consequence, the collector will take longer to complete a whole CMS cycle.



## **-XX:+ExplicitGCInvokesConcurrent**

The flag `-XX:+ExplicitGCInvokesConcurrent` instructs the JVM to run a CMS GC instead of a full GC whenever system GC is requested.

## **-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses**

ensures that the permanent generation is included into the CMS GC in case of a system GC request.

Note : `System.gc()` triggers a full GC by default, recommended to disable explicit Gc (`-XX:+DisableExplicitGC`)

## **Challenges**

When using the CMS Collector in real-world applications, we face two major challenges that may create a need for tuning:

Heap fragmentation

High object allocation rate

## Heap fragmentation

It is possible because, unlike the Throughput Collector, the CMS Collector does not contain any mechanism for defragmentation.

As a consequence, an application may find itself in a situation where an object cannot be allocated even though the total heap space is far from exhausted – simply because there is no consecutive memory area available to fully accommodate the object.

When this happens, the concurrent algorithms do not help anymore and thus, as a last resort, the JVM triggers a full GC.

## **High object allocation rate**

If the rate at which objects get instantiated is higher than the rate at which the collector removes dead objects from the heap, the concurrent algorithm fails once again.

At some point, the old generation will not have enough space available to accommodate an object that is to be promoted from the young generation.

This situation is referred to as “concurrent mode failure”, and the JVM reacts just like in the heap fragmentation scenario: It triggers a full GC.

## **Solution:**

One possible countermeasure is to increase young generation size, in order to prevent premature promotions of short-lived objects into the old generation.

Another approach is to take heap dumps of the running system, to analyze the application for excessive object allocation, identify these objects, and eventually reduce the amount of objects allocated.

## **Excessive GC Time and OutOfMemoryError**

The CMS collector throws an OutOfMemoryError if too much time is being spent in garbage collection: if more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an OutOfMemoryError is thrown.

This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small.

If necessary, this feature can be disabled by adding the option `-XX:-UseGCOverheadLimit` to the command line

# **CMS GC Logs**

**39.910: [GC 39.910: [ParNew: 261760K-  
>0K(261952K), 0.2314667 secs] 262017K-  
>26386K(1048384K), 0.2318679 secs]**

Young generation (ParNew) collection. Young generation capacity is 261952K and after the collection its occupancy drops down from 261760K to 0. This collection took 0.2318679 secs.



**40.146: [GC [1 CMS-initial-mark: 26386K(786432K)]  
26404K(1048384K), 0.0074495 secs]**

Beginning of tenured generation collection with CMS collector. This is initial Marking phase of CMS where all the objects directly reachable from roots are marked and this is done with all the application threads stopped.

Capacity of tenured generation space is 786432K and CMS was triggered at the occupancy of 26386K.

## **40.154: [CMS-concurrent-mark-start]**

Start of concurrent marking phase.

In Concurrent Marking phase, application thread stopped in the first phase are started again and all the objects transitively reachable from the objects marked in first phase are marked here.

## **40.683: [CMS-concurrent-mark: 0.521/0.529 secs]**

Concurrent marking took total 0.521 seconds cpu time and 0.529 seconds wall time that includes the yield to other threads also.

## **40.683: [CMS-concurrent-preclean-start]**

Start of precleaning.

Precleaning is also a concurrent phase. Here in this phase we look at the objects in CMS heap which got updated by promotions from young generation or new allocations or got updated by mutators while we were doing the concurrent marking in the previous concurrent marking phase

## **40.701: [CMS-concurrent-preclean: 0.017/0.018 secs]**

Concurrent precleaning took 0.017 secs total cpu time and 0.018 wall time.

**40.704: [GC40.704: [Rescan (parallel) ,  
0.1790103 secs]40.883: [weak refs processing,  
0.0100966 secs] [1 CMS-remark:  
26386K(786432K)] 52644K(1048384K),  
0.1897792 secs]**

Stop-the-world phase. This phase rescans any residual updated objects in CMS heap, retraces from the roots and also processes Reference objects.

Here the rescanning work took 0.1790103 secs and weak reference objects processing took 0.0100966 secs. This phase took total 0.1897792 secs to complete

**40.894: [CMS-concurrent-sweep-start]**

Start of sweeping of dead/non-marked objects. Sweeping is concurrent phase performed with all other threads running.

**41.020: [CMS-concurrent-sweep: 0.126/0.126 secs]**

Sweeping took 0.126 secs.

**41.020: [CMS-concurrent-reset-start]**

Start of reset.

**41.147: [CMS-concurrent-reset: 0.127/0.127 secs]**

In this phase, the CMS data structures are reinitialized so that a new cycle may begin at a later time. In this case, it took 0.127 secs.

This was how(previous logs) a normal CMS cycle runs.

Now let us look at some other CMS log entries:

**197.976: [GC 197.976: [ParNew: 260872K-  
>260872K(261952K), 0.0000688 secs]197.976:  
[CMS197.981: [CMS-concurrent-sweep:  
0.516/0.531 secs]  
(concurrent mode failure): 402978K-  
>248977K(786432K), 2.3728734 secs] 663850K-  
>248977K(1048384K), 2.3733725 secs]**

## **Problem:**

This shows that a ParNew collection was requested, but it was not attempted because it was estimated that there was not enough space in the CMS generation to promote the worst case surviving young generation objects.

We name this failure as “full promotion guarantee failure”.

Due to this, Concurrent Mode of CMS is interrupted and a Full GC is invoked at 197.981. This mark-sweep-compact stop-the-world Full GC took 2.3733725 secs and the CMS generation space occupancy dropped from 402978K to 248977K.

## **Solution:**

The concurrent mode failure can either be avoided by increasing the tenured generation size or initiating the CMS collection at a lesser heap occupancy by setting `CMSInitiatingOccupancyFraction` to a lower value and setting `UseCMSInitiatingOccupancyOnly` to true.

The value for `CMSInitiatingOccupancyFraction` should be chosen appropriately because setting it to a very low value will result in too frequent CMS collections.



# **The G1 Garbage Collector**

## **The G1 Garbage Collector**

The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories.

It meets garbage collection (GC) pause time goals (defined through `-XX:MaxGCPauseMillis`) with a high probability, while achieving high throughput.

G1 works to accomplish those goals in a few different ways.

First, being true to its name, G1 collects regions with the least amount of live data (Garbage First!) and compacts/evacuates live data into new regions.

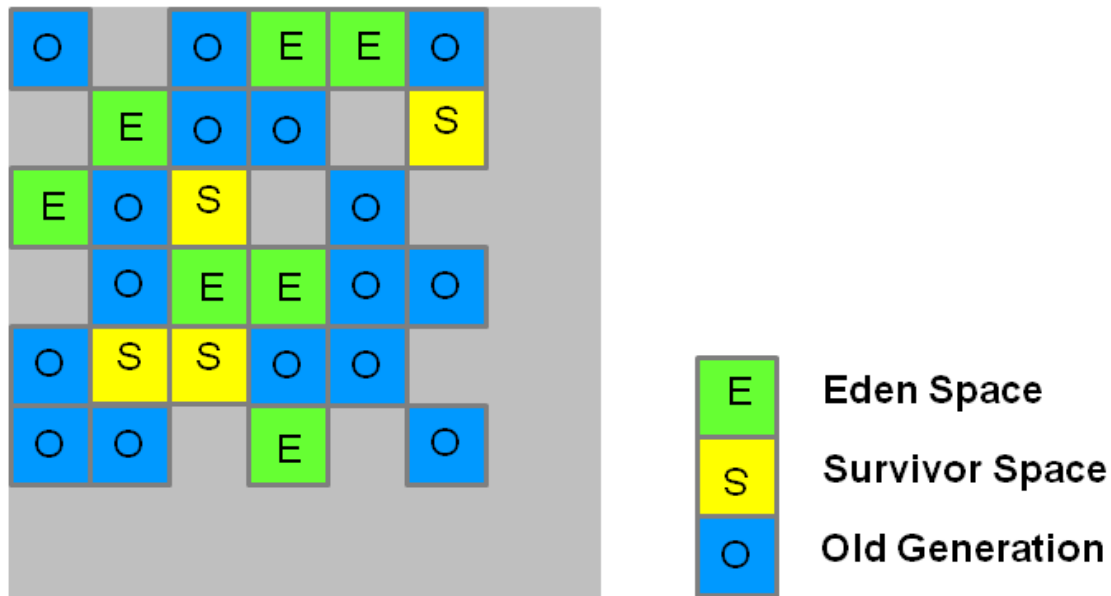
Secondly, it uses a series of incremental, parallel and multi-phased cycles to achieve its soft pause time target.

## “Regions”

A region represents a block of allocated space that can hold objects of any generation without the need to maintain contiguity with other regions of the same generation.

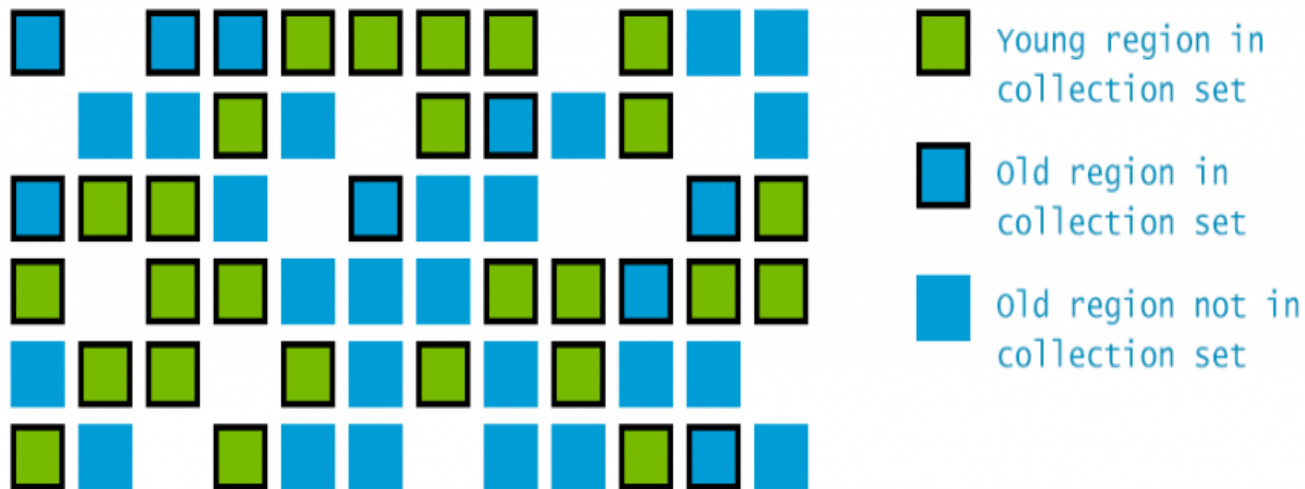
The heap is one memory area split into many fixed sized regions.

## G1 Heap Allocation



## Collection set

To avoid collecting the entire heap at once, only a subset of the regions, called the collection set will be considered at a time.



## **Collection Set (continued)**

During the concurrent phase it estimates the amount of live data that each region contains. This is used in building the collection set: the regions that contain the most garbage are collected first.

Hence the name: *garbage-first* collection.

## **Heap Memory allocation:**

Allocation is done on a per-thread basis.

Thread Local Area (TLA), is a dedicated partition that a thread allocates freely within, without having to claim a full heap lock. Once the area is full, the thread is assigned a new area until the heap runs out of areas to dedicate.

Note : TLAs are part of the heap. Thread Stacks are not on the heap.



Objects can then be allocated within those thread-local buffers without the need for additional synchronization.

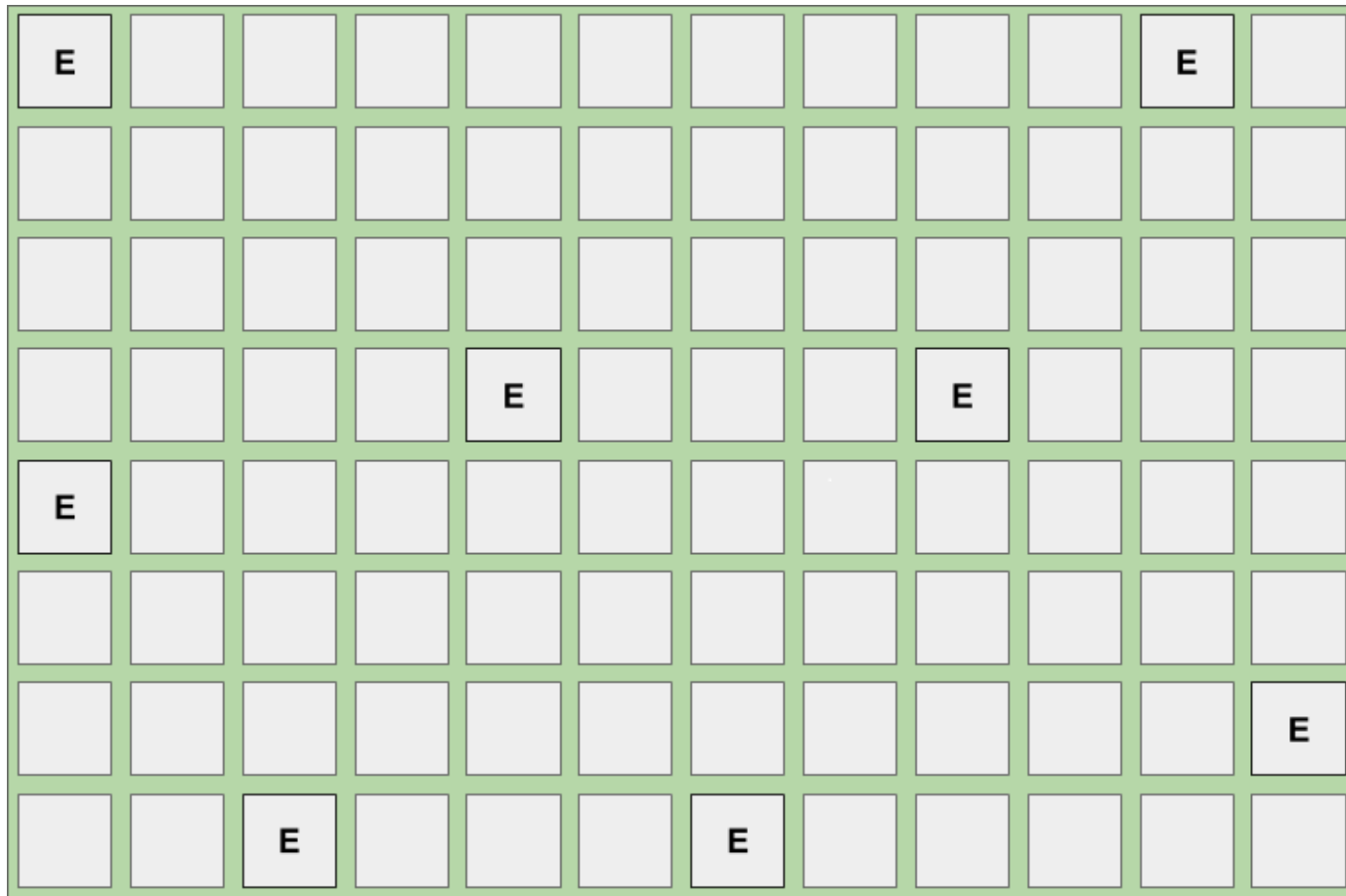
When the region has been exhausted of space, a new region is selected, allocated and filled.

This continues until the cumulative Eden region space has been filled, triggering an evacuation pause (also known as a young collection / young gc / young pause or mixed collection / mixed gc / mixed pause).

The cumulative amount of Eden space represents the number of regions we believe can be collected within the defined soft pause time target.

The percentage of total heap allocated for Eden regions can range from 5% to 60% and gets dynamically adjusted after each young collection based on the performance of the previous young collection.

objects being allocated into non-contiguous Eden regions:



GC pause (young); #1

[Eden: 612.0M(612.0M)->0.0B(532.0M) Survivors: 0.0B->80.0M Heap:  
612.0M(12.0G)->611.7M(12.0G)]

GC pause (young); #2

[Eden: 532.0M(532.0M)->0.0B(532.0M) Survivors: 80.0M->80.0M Heap:  
1143.7M(12.0G)->1143.8M(12.0G)]

pause #1, evacuation was triggered because Eden reached **612.0M** out of a total of **612.0M** (153 regions).

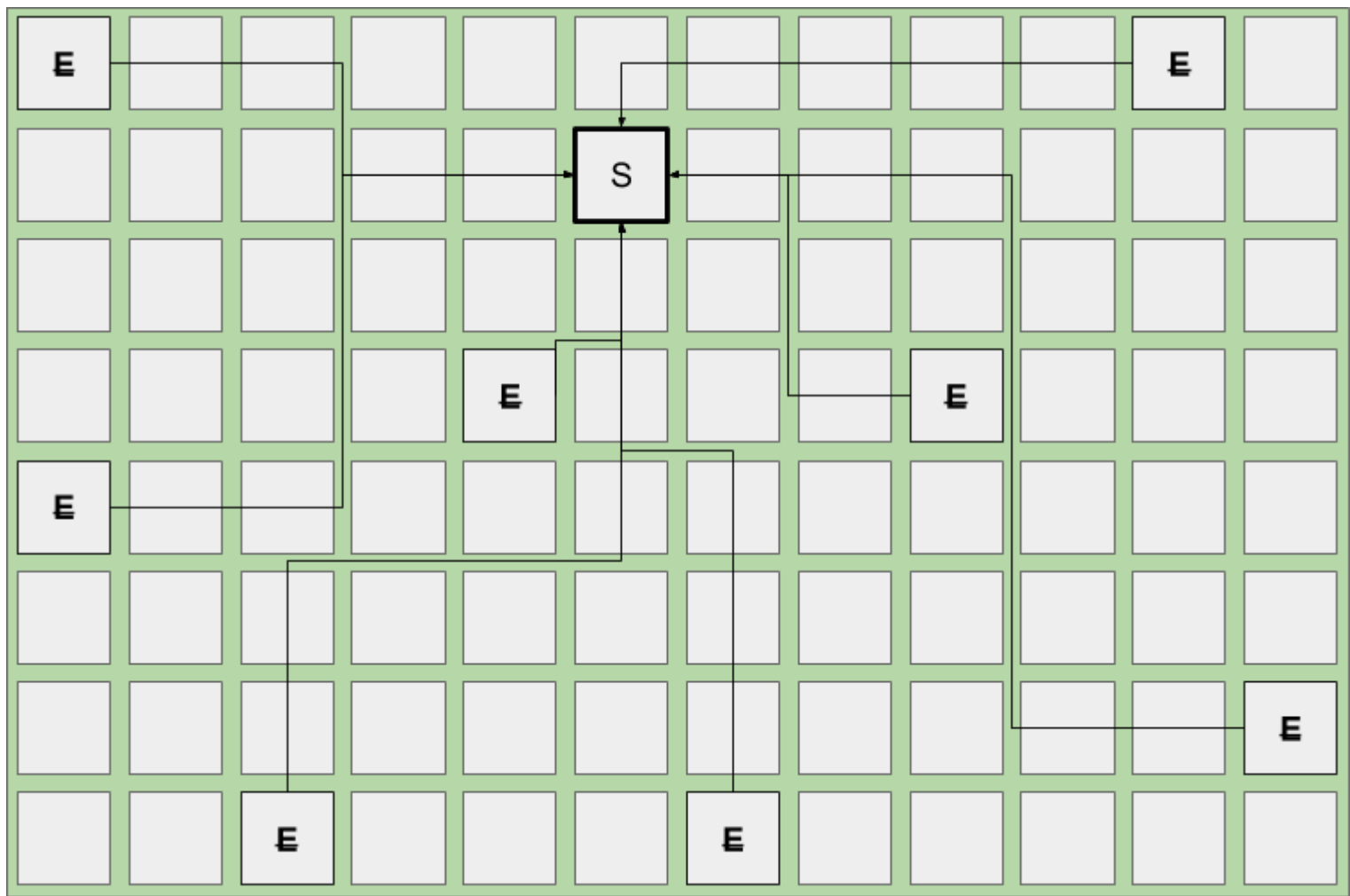
The current Eden space was fully evacuated, **0.0B** and, given the time taken, it also decided to reduce the total Eden allocation to **532.0M** or 133 regions.

In pause #2, we can see the evacuation is triggered when we reach the new limit of **532.0M**. Because we achieved an optimal pause time, Eden was kept at **532.0M**.

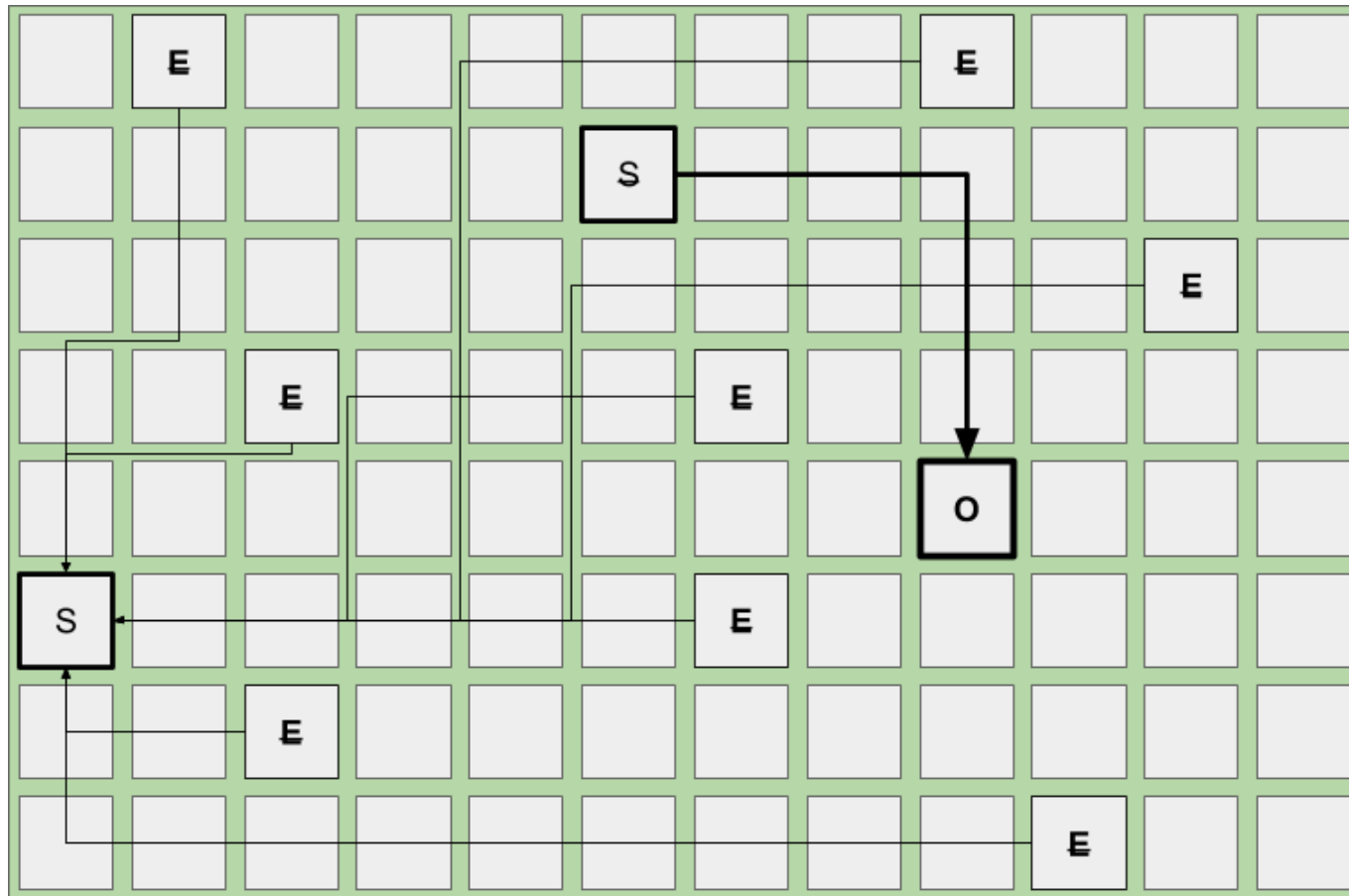
*Young collection takes place, dead objects are collected and any remaining live objects are evacuated and compacted into the Survivor space.*

G1 has an explicit hard-margin, defined by the G1ReservePercent (default 10%), that results in a percentage of the heap always being available for the Survivor space during evacuation.

This principle ensures that after every successful evacuation, all previously allocated Eden regions are returned to the free list and any evacuated live objects end up in Survivor space.



Survivor space are evacuated and promoted to a new region in the Old space while live objects from Eden are evacuated into a new Survivor space region.



G1 will continue with this pattern until one of three things happens:

- ✓ It reaches a configurable soft-margin known as the InitiatingHeapOccupancyPercent (IHOP).
- ✓ It reaches its configurable hard-margin (G1ReservePercent)
- ✓ It encounters a humongous allocation



InitiatingHeapOccupancyPercent (number of objects in the **old regions**> **45%**)

A request is made to start a concurrent marking cycle:

8801.974: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation, reason: occupancy higher than threshold, occupancy: 12582912000 bytes, allocation request: 0 bytes, threshold: 12562779330 bytes (45.00 %), source: end of GC]

8804.670: [G1Ergonomics (Concurrent Cycles) initiate concurrent cycle, reason: concurrent cycle initiation requested]

8805.612: [GC concurrent-mark-start]

8820.483: [GC concurrent-mark-end, 14.8711620 secs]

Once the concurrent marking cycle completes, a young collection is immediately triggered, followed by a second type of evacuation, known as **a mixed collection**.

8821.975: [G1Ergonomics (**Mixed GCs**) start mixed GCs, reason: candidate old regions available, candidate old regions: 553 regions, reclaimable: 6072062616 bytes (21.75 %), threshold: 5.00 %]

A mixed collection is starting because the number of candidate Old regions (553) have a combined 21.75% reclaimable space. This value is higher than our 5% minimum threshold (5% default in JDK8u40+ / 10% default in JDK7) defined by the G1HeapWastePercent and as such, mixed collections will begin.

## Overflow and Exhausted Log Messages

When you see to-space overflow/exhausted messages in our logs, the G1 GC does not have enough memory for either survivor or promoted objects, or for both. The Java heap cannot expand since it is already at its max.

Example messages:

924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs]

OR

924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs]

Solution : try the following adjustments:

Increase the value of the `-XX:G1ReservePercent` option (and the total heap accordingly) to increase the amount of reserve memory for "to-space".

8822.178: [GC pause (mixed) 8822.178: [G1Ergonomics (CSet Construction)  
start choosing CSet, \_pending\_cards: 74448, predicted base time: 170.03 ms,  
remaining time: 829.97 ms, target pause time: 1000.00 ms]

A mixed collection will look to collect all three generations within the same pause time target.

It manages this through the incremental collection of the Old regions based on the value of `G1MixedGCCountTarget` (defaults to 8).

Meaning, it will divide the number of candidate Old regions by the **`G1MixedGCCountTarget`** and try to collect at least that many regions during each cycle.

After each cycle finishes, the liveness of the Old region is re-evaluated.

If the reclaimable space is still greater than the G1HeapWastePercent, mixed collections will continue.

8822.704: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions available, candidate old regions: 444 regions, reclaimable: 4482864320 bytes (16.06 %), threshold: 10.00 %]

Mixed collections will continue until all eight are completed or until the reclaimable percentage no longer meets the G1HeapWastePercent.

From there, we will see the mixed collection cycle finish and the following events will return to standard young collections.

8830.249: [G1Ergonomics (Mixed GCs) **do not continue mixed GCs**, reason: **reclaimable percentage not over threshold**, candidate old regions: 58 regions, **reclaimable: 2789505896 bytes (9.98 %)**, threshold: 10.00 %]

The above process is for standard use-cases.

Suppose the size of an object is greater than 50% of a single region. In this case, objects are considered to be humongous and are handled by performing specialized humongous allocations.

Region Size: 4096 KB

Object A: 12800 KB

Result: Humongous Allocation across 4 regions

S	E		O		O	O		E			O
					S					S	
O	O		E	H				O		E	
		E		O		E	S			S	
	O			S				O		O	
S			O			E			S		
O		E								E	
	O			O			E		E		



Humongous allocation represents a single object, and as such, must be allocated into contiguous space. This can lead to significant fragmentation.

Humongous objects are allocated to a special humongous region directly within the Old generation. This is because the cost to evacuate and copy such an object across the young generations can be too high.

Even though the object in question is only 12.5 MB, it must consume four full regions accounting for 16 MB of total usage.

Humongous allocations always trigger a concurrent marking cycle, whether the IHOP criteria is met or not.

A steady allocation of them can lead to significant heap fragmentation and a noticeable performance impact. Prior to JDK8u40, humungous objects could only be collected through a Full GC so the potential for this to impact JDK7 and early JDK8 users is very high.

4948.653: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation, reason: requested by GC cause, GC cause: G1 Humongous Allocation]

**Full GC (Need to fine tune the application & GC)**

## **Full GC (Need to fine tune the application & GC)**

6229.578: [GC pause (young) (to-space exhausted),  
0.0406140 secs]

6229.691: [Full GC 10G->5813M(12G), 15.7221680 secs]

57929.136: [**GC concurrent-mark-start**]

57955.723: [**Full GC 10G->5109M(12G)**, 15.1175910 secs]

57977.841: [**GC concurrent-mark-abort**]

### ***Young GC:***

The collection set of the *Young GC* includes only young/survivor regions.

### ***Mixed GC:***

The collection set of the *Mixed GC* includes both young/survivor regions, but also old regions.

## **Humongous Objects and Humongous Allocations**

For G1 GC, any object that is more than half a region size is considered a "**Humongous object**". Such an object is allocated directly in the old generation into "**Humongous regions**". *These Humongous regions are a contiguous set of regions.*

## **G1 Garbage Collector - Big Heaps and Low Pauses?**

The **goal of the G1 collector** is to achieve a predictable soft-target pause time, defined through -XX:MaxGCPauseMillis, while also maintaining consistent application throughput.

G1 collects regions with the least amount of live data (Garbage First!) and compacts/evacuates live data into new regions. Secondly, it uses a series of incremental, parallel and multi-phased cycles to achieve its soft pause time target.

The key is that while G1 is a generational collector, the allocation and consumption of space is both non-contiguous and free to evolve as it gains a better understanding of the most efficient young to old ratio.

## **G1 GC Defaults**

-XX:G1HeapRegionSize=n

Sets the size of a G1 region. The value will be a power of two and can range from 1MB to 32MB. The goal is to have around 2048 regions based on the minimum Java heap size

-XX:MaxGCPauseMillis=200

Sets a target value for desired maximum pause time. The default value is 200 milliseconds. The specified value does not adapt to your heap size



-XX:ParallelGCThreads=n

Sets the value of the STW worker threads. Sets the value of n to the number of logical processors. The value of n is the same as the number of logical processors up to a value of 8.

For  $N > 8$  available processors, the number of GC threads will be computed as  $3 + 5N/8$ .

-XX:ConcGCTThreads=n

Sets the number of parallel marking threads. Sets n to approximately 1/4 of the number of parallel garbage collection threads (ParallelGCThreads).

-XX:InitiatingHeapOccupancyPercent=45

Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.

-XX:MaxTenuringThreshold=n

Maximum value for tenuring threshold. The default value is 15.

`-XX:G1ReservePercent=n`

Sets the amount of heap that is reserved as a false ceiling to reduce the possibility of promotion failure. The default value is 10.

`-XX:G1HeapRegionSize=n`

With G1 the Java heap is subdivided into uniformly sized regions. This sets the size of the individual sub-divisions. The default value of this parameter is determined ergonomically based upon heap size. The minimum value is 1Mb and the maximum value is 32Mb.

## **G1 Tuning options:**

Main goal of G1 is low latency, set the Pause goal

-XX:MaxGCPauseMillis=300

When to start a concurrent GC cycle

-XX:InitiatingHeapOccupancyPercent=n  
(percent of an entire heap, not just old gen)

Too Low -> Unnecessary GC overhead

Too High -> "Space Overflow" -> Full GC

## **G1 Tuning options (continued)**

`-XX:G1MixedGCLiveThresholdPercent=65`

Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 65 percent.

`-XX:G1HeapWastePercent=10`

Sets the percentage of heap that you are willing to waste. The Java HotSpot VM does not initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage.

## **-XX:G1MixedGCCountTarget=8**

Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at most G1MixedGCLiveThresholdPercent live data.

To High-> Unnecessary overhead

To Low-> Long Pauses

## **-XX:G1OldCSetRegionThresholdPercent=10**

Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle.

To High -> More aggressive collecting (More live objects to copy)

To Low -> Wasting some heap

**Some options cause "PauseTime Target to be ignored!"**

**-Xmn (Fixed young generation size)**

To enable the G1 Collector use:

```
java -Xmx50m -Xms50m -XX:+UseG1GC -XX:MaxGCPauseMillis=200
```

**-XX:+UseG1GC** - Tells the JVM to use the G1 Garbage collector.

**-XX:MaxGCPauseMillis=200** - Sets a target for the maximum GC pause time. This is a soft goal, and the JVM will make its best effort to achieve it. Therefore, the pause time goal will sometimes not be met. The default value is 200 milliseconds.

**-XX:InitiatingHeapOccupancyPercent=45** - Percentage of the (entire) heap occupancy to start a concurrent GC cycle. It is used by G1 to trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations. A value of 0 denotes 'do constant GC cycles'. The default value is 45 (i.e., 45% full or occupied).



## G1 GC Commands :

### Option and Default Value

-XX:+UseG1GC

### Description

Use the Garbage First (G1) Collector

-XX:MaxGCPauseMillis=n

Sets a target for the maximum GC pause time. This is a soft goal, and the JVM will make its best effort to achieve it.

-XX:InitiatingHeapOccupancyPercent=n

Percentage of the (entire) heap occupancy to start a concurrent GC cycle. It is used by GCs that trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations (e.g., G1). A value of 0 denotes 'do constant GC cycles'. The default value is 45.

-XX:NewRatio=n

Ratio of new/old generation sizes. The default value is 2.

-XX:SurvivorRatio=n

Ratio of eden/survivor space size. The default value is 8.

-XX:MaxTenuringThreshold=n

Maximum value for tenuring threshold. The default value is 15.

-XX:ParallelGCThreads=n

Sets the number of threads used during parallel phases of the garbage collectors. The default value varies with the platform on which the JVM is running.

-XX:ConcGCThreads=n

Number of threads concurrent garbage collectors will use. The default value varies with the platform on which the JVM is running.

-XX:G1ReservePercent=n

Sets the amount of heap that is reserved as a false ceiling to reduce the possibility of promotion failure. The default value is 10.

-XX:G1HeapRegionSize=n

With G1 the Java heap is subdivided into uniformly sized regions. This sets the size of the individual sub-divisions. The default value of this parameter is determined ergonomically based upon heap size. The minimum value is 1Mb and the maximum value is 32Mb.

## Logging GC with G1

(1) `-verbosegc` (which is equivalent to `-XX:+PrintGC`) sets the detail level of the log to fine.

`-Xloggc:gc.log -XX:+PrintGC`

(2) `-XX:+PrintGCDetails` sets the detail level to finer. The options shows the following information:

`-Xloggc:gc.log -XX:+PrintGCDetails`

(3) `-XX:+UnlockExperimentalVMOptions -XX:G1LogLevel=finest` sets the detail level to its finest. Like finer but includes individual worker thread information.

**`XX:+PrintAdaptiveSizePolicy`** : This option will provide many ergonomic details that are purposefully kept out of the **`-XX:+PrintGCDetails`** option.

The option `-XX:+G1SummarizeRSetStats` can be used to provide a window into the total number of RSet coarsenings to help determine if concurrent refinement threads are able to handle the updated buffers.

`-XX:G1SummarizeRSetStatsPeriod=n`

This option summarizes RSet statistics every  $n$ th GC pause

# G1 GC Log Format

**1** **2015-09-14T12:32:24.398-0700: 0.356: [GC pause (G1 Evacuation Pause) (young), 0.0215287 secs]**

**4** [Parallel Time: 20.0 ms, **GC Workers: 8**]

[GC Worker Start (ms): Min: 355.9, Avg: 356.3, Max: 358.4, Diff: 2.4]

[Processed Buffers: Min: 0, Avg: 1.1, Max: 5, Diff: 5, Sum: 9]

:

:

[Free CSet: 0.0 ms]

**5** **[Eden: 12.0M(12.0M)->0.0B(14.0M) Survivors: 0.0B->2048.0K Heap: 12.6M(252.0M)->7848.3K(252.0M)]**

**6** **[Times: user=0.08 sys=0.00, real=0.02 secs]**

**1** **2015-09-14T12:32:24.398-0700: 0.356** – indicates the time at which this GC event fired. Here **0.356** indicates that 356 milliseconds after the Java process was started this GC event was fired.

**2** **GC pause (G1 Evacuation Pause)** — Evacuation Pause is a phase where live objects are copied from one region (young or young + old) to another region.

**3** **(young)** – indicates that this is a Young GC event.

**4** **GC Workers: 8** – indicates the number of GC worker threads.

**5** **[Eden: 12.0M(12.0M)->0.0B(14.0M) Survivors: 0.0B->2048.0K Heap: 12.6M(252.0M)->7848.3K(252.0M)]** – This line indicates the heap size changes:

**Eden: 12.0M(12.0M)->0.0B(14.0M)** - indicates that Eden generation's capacity was 12mb and all of the 12mb was occupied. After this GC event, young generation occupied size came down to 0. Target Capacity of Eden generation has been increased to 14mb, but not yet committed. Additional regions are added to Eden generation, as demands are made.

**Survivors: 0.0B->2048.0K** - indicates that Survivor space was 0 bytes before this GC event. But after the event Survivor size increased to 2048kb. It indicates that objects are promoted from Young Generation to Survivor space.

**Heap: 12.6M(252.0M)->7848.3K(252.0M)** – indicates that capacity of heap size was 252mb, in that 12.6mb was utilized. After this GC event, heap utilization dropped to 7848.3kb (i.e. 5mb (i.e. 12.6mb – 7848.3kb) of objects has been garbage collected in this event). And heap capacity remained at 252mb.

**6** **Times: user=0.08, sys=0.00, real=0.02 secs**

# JVM Ergonomics

JVM Ergonomics is the process by which the Java Virtual Machine (JVM) and garbage collection tuning, such as behavior-based tuning, improve application performance.

The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler.

## Garbage Collector, Heap, and Runtime Compiler Default Selections

A class of machine referred to as a server-class machine has been defined as a machine with the following:

- 2 or more physical processors

- 2 or more GB of physical memory

On server-class machines, the following are selected by default:

- Throughput garbage collector

- Initial heap size of  $1/64$  of physical memory up to 1 GB

- Maximum heap size of  $1/4$  of physical memory up to 1 GB

- Server runtime compiler



The G1 collector is designed for applications that:

Can operate concurrently with applications threads like the CMS collector.

Compact free space without lengthy GC induced pause times.

Need more predictable GC pause durations.

Do not want to sacrifice a lot of throughput performance.

Do not require a much larger Java heap.

G1 is the HotSpot low-pause collector

Long term replacement for CMS

Low pauses valued more than max throughput

- > For majority of java apps
- > For others, ParallelGC will still be available

Tuning based on max Stop-The-World Pause

- > -XX:MaxGCPauseMillis=<> (default 250 ms)
- > ex: java -Xmx32G -XX:MaxGCPauseMillis=100

G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS, there are differences that make G1 a better solution.

One difference is that G1 is a compacting collector. G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions. This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues.

Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.

**Tune Latency/Responsiveness**

## **Tune Latency/Responsiveness**

step involve several iterations of refining the Java heap size configuration, evaluating garbage collection duration and frequency, possibly switching to a different garbage collector, and further fine-tuning of space sizes in the event of a change to a different garbage collector.

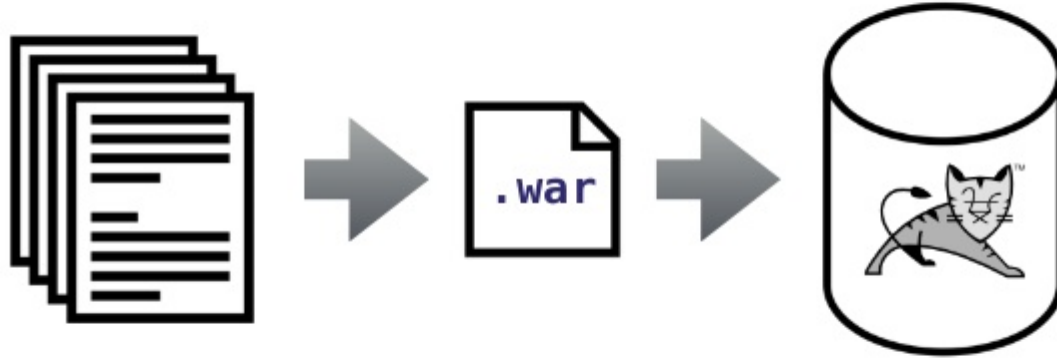
There are two possible outcomes of this step:

- ✓ Application latency requirements are met
- ✓ Application latency requirements are not met.

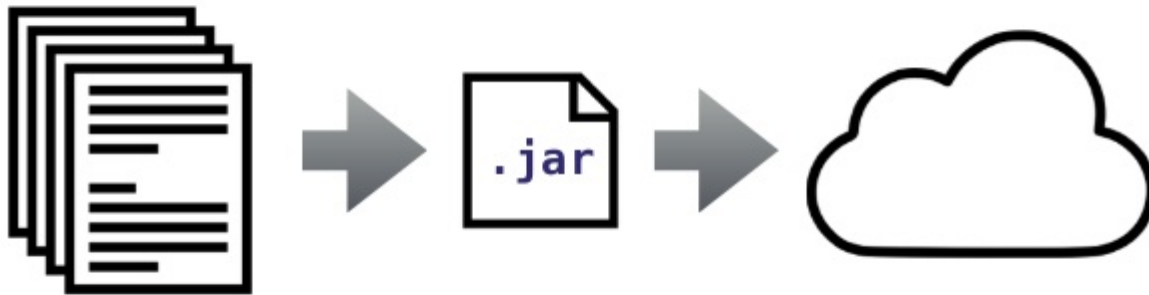
Possible activities that may drive changes to improve the application's latency might include:

- a. Heap profiling and making changes to the application to reduce object allocation or object retention
- b. Changing the JVM deployment model to lessen the amount of work or load taken on by a JVM

## Traditional JVM Deployment



## Modern JVM Deployment



Microservices???



# **GC Monitoring**

## **What is GC Monitoring?**

Garbage Collection Monitoring refers to the process of figuring out how JVM is running GC. For example, we can find out:

when an object in young has moved to old and by how much, or when stop-the-world has occurred and for how long.

## **Steps to optimize GC**

### 1. Understand the basics of GC

Understanding how GC works is important because of the large number of variables that need to be tuned.

(Ref :

<http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>)

## 2. Scope out GC requirements

There are certain characteristics of GC that we should optimize, to reduce its overhead on application performance. Like throughput and latency

Stop-the-world collectors pause the application threads to collect garbage. The duration and frequency of these pauses should not adversely impact the application's ability to adhere to the SLA.

Concurrent GC algorithms contend with the application threads for CPU cycles. This overhead should not affect the application throughput.

Non-compacting GC algorithms can cause heap fragmentation, which leads to long stop-the-world pauses due to full GC. The heap fragmentation should be kept to a minimum.

Garbage collection needs memory to work. Certain GC algorithms have a higher memory footprint than others. If the application needs a large heap, make sure the GC's memory overhead is not large.

A clear understanding of GC logs and commonly used JVM parameters is necessary to easily tune GC behavior should the code complexity grow or workload characteristics change.

GC attacks by Linux

## **IO Starvation**

-> Symptom : GC log shows "low user time, low system time, long pause"

-> Cause : GC threads stuck in kernel waiting for IO, usually due to journal commits or FS flush of changes by gzip or log rolling

## **Memory starvation:**

-> Symptom : GC log shows "low user time, high system time, long pause"

-> Cause : Memory pressure triggers swapping or scanning for free memory

## Solutions for GC-attacks

### IO Starvation

-> Strategy : Even out workload to disk drives  
(flush every 5s rather than 30s)

```
sysctl -w vm.dirty_writeback_centisecs = 500
```

```
sysctl -w vm.dirty_expire_centisecs = 500
```



## Memory Starvation

- > Strategy : Pre-allocate memory to JVM heap and protect it against swapping or scanning
- > Turn on `-XX:+AlwaysPreTouch` option in JVM
- > `sysctl -w vm.swappiness=0` to protect heap and anonymous memory
- > JVM start up has 2 second delay to allocate all memory (17GB!!)

	Description	Notes
<b>-Xms&lt;size&gt;</b>	Sets the minimum heap size	In production you should set the min and max the same
<b>-Xmx&lt;size&gt;</b>	Sets the maximum heap size	
<b>-XX:NewSize=&lt;size&gt;</b>	Sets the minimum young generation size	In production you should set the min and max the same
<b>-XX:MaxNewSize=&lt;size&gt;</b>	Sets the maximum young generation size	
<b>-XX:NewRatio=&lt;number&gt;</b>	Sets the ratio of the size of the young generation as compared to the tenured generation	Use either the NewSize/MaxSize arguments or the NewRatio argument but not both.

- **XX:SurvivorRatio=<number>** Sets the ratio of the size of the eden space compared to one survivor spaces vary the ratio based on the young generation size, a ratio of 8 for small young generations (10MB) and 32 for larger ones (100MB)
- **XX:MaxTenuringThreshold=<number>** Indicates the number of minor collections that an object must survive before being automatically placed in the tenured generation Usually you should use a value of 32
- **XX:MaxPermSize=<size>** Sets the size of the permanent generation Don't set this unless you run out of space
- **XX:+DisableExplicitGC** This option forces the JVM to ignore calls to the *System.gc()* method

**--Xss1024k** Thread Stack is the memory that is used by each thread to push/pop functions and variables used for function calls

**-XX:+CMSClassUnloadingEnabled** the GC will sweep PermGen, too, and remove classes which are no longer used.

Code Cache (non-heap): HotSpot JVM also includes a "code cache" containing memory used for compilation and storage of native code

-XX:ReservedCodeCacheSize=32m      Reserved code cache size (in bytes)

Permanent Generation (non-heap): holds all the reflective data of the virtual machine itself, such as class and method objects.

-XX:+HeapDumpOnOutOfMemoryError writes heap dump on  
OutOfMemoryError (recommended)

-XX:+HeapDumpOnCtrlBreak writes heap dump together with thread dump on  
CTRL+BREAK

### **Virtual Space-1: (MaxNewSize – NewSize)**

The First Virtual Space is actually shows the difference between the -XX:NewSize and -XX:MaxNewSize. Or we can say that it is basically a difference between the Initial Young Size and the Maximum Young Size.

### **Virtual Space-2: (MaxHeapSize – InitialHeapSize)**

The Second Virtual Space is actually the Difference between the Maximum Heap size (-Xmx) and the Initial Heap Size (-Xms). This is called as virtual space because initially the JVM will allocate the Initial Heap Size and then according to the requirement the Heap size can grow till the MaxHeapSize.

```
JAVA_OPTS="-server -Xms2g -Xmx2g  
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC  
-Dsun.rmi.dgc.client.gcInterval=3600000  
-Dsun.rmi.dgc.server.gcInterval=3600000
```

```
-XX:PermSize=256m -XX:MaxPermSize=256m  
-Dtomcat.util.buf.StringCache.byte.enabled=true  
-Dtomcat.util.buf.StringCache.char.enabled=true  
-Dtomcat.util.buf.StringCache.trainThreshold=5  
-Dtomcat.util.buf.StringCache.cacheSize=2000  
-Djava.net.preferIPv4Stack=true"
```

```
JAVA_OPTS="$JAVA_OPTS -XXloggc:log/appgc.log -  
XX:+PrintGCDetails -XX:ParallelGCThreads=20 -  
XX:SurvivorRatio=10 -XX:TargetSurvivorRatio=90  
-XX:MaxTenuringThreshold=30  
-XX:+HeapDumpOnOutOfMemoryError"
```

## **Stack size**

Each thread in the VM gets a stack. The stack size will limit the number of threads that you can have, too big of a stack size and you will run out of memory as each thread is allocated more memory than it needs.

-Xss determines the size of the stack: -Xss1024k. If the stack space is too small, eventually you will see an exception class `java.lang.StackOverflowError`.

We can change stack size settings at the OS level for Linux. A call to `ulimit` may be necessary, and is usually done with a command in `/etc/profile`:

Limit thread stack size on Linux `ulimit -s 2048`

**NEW** : A thread is in a new state when it has just been created and the start() method hasn't been invoked

**RUNNING** : A thread is in a runnable state when the start method has been invoked and the JVM actively schedules it for execution

**BLOCKED / WAITING FOR MONITOR ENTRY**: Thread enters this state when it isn't able to acquire the necessary monitors to enter a synchronized block. A thread waits for other threads to relinquish the monitor. A thread in this state isn't doing any work

**WAITING** : A thread enters this state when it relinquishes its control over the monitor by calling Object.wait(). It waits till the time other threads invoke Object.notify() or Object.notifyAll(). In this state too the thread isn't doing any work

**TIMED WAITING** : A thread enters this state when it relinquishes its control over the monitor by calling Object.wait(long). It waits till the time other threads invoke Object.notify() / Object.notifyAll() or when the wait time expires. In this state too the thread isn't doing any work

**SLEEPING / WAITING ON CONDITION**: A thread sleeps when it calls Thread.sleep(long). As the name suggests the thread isn't doing in this state either Terminated



**DEMO - GC Test**

Heap	GC Algorithm	Useful Work	Longest Pause
-Xmx2g	-XX:+UseParallelGC	52.82%	0.43515s
-Xmx2g	XX:+UseConcMarkSweepGC	65.29%	0.54668s
-Xmx2g	-XX:+UseG1GC	53.13%	0.51114s
-Xmx4g	-XX:+UseParallelGC	76.55%	0.56021s
-Xmx4g	XX:+UseConcMarkSweepGC	74.23%	1.13218s
-Xmx4g	-XX:+UseG1GC	74.77%	0.95308s

## Tuning for Latency

Let us assume we have a requirement stating that all jobs must be processed in under 500ms. Knowing that the actual job processing takes just 50 ms we can simplify and deduct the latency requirement for individual GC pauses.

Our requirement now states that no GC pause can stop the application threads for longer than 450 ms.

<b>-Xmx2g</b>	<b>-XX:+UseParallelGC</b>	<b>52.82%</b>	<b>0.43515s</b>
---------------	---------------------------	---------------	-----------------

## Tuning for Throughput

Let us assume that we have a throughput goal to process 11,000,000 jobs/hour. The example configurations used again give us a configuration where the requirement is fulfilled:

<b>-Xmx4g</b>	<b>-XX:+UseParallelGC</b>	<b>76.55%</b>	<b>0.56021s</b>
---------------	---------------------------	---------------	-----------------

we can see that the CPUs are blocked by GC for 23.45% of the time, leaving 76.55% of the computing power for useful work. For simplicity's sake we will ignore other safe points in the example. Now we have to take into account that:

1. One job is processed in 100 ms by a single core
2. Thus, in one minute, 60,000 jobs could be processed by one core
3. In one hour, a single core could thus process 3.6 M jobs
4. We have four cores available, which could thus process  $4 \times 3.6 \text{ M} = 14.4 \text{ M}$  jobs in an hour

With this amount of theoretical processing power we can make a simple calculation and conclude that during one hour we can in reality process 76.55% of the 14.4 M theoretical maximum resulting in 11,023,200 processed jobs/hour, fulfilling our requirement.

## Tuning for Capacity

Let us assume we have to deploy our solution to the commodity-class hardware with up to four cores and 4 G RAM available. From this we can derive our capacity requirement that the maximum heap space for the application cannot exceed 2 GB

<b>-Xmx2g</b>	<b>-XX:+UseParallelGC</b>	<b>52.82%</b>	<b>0.43515s</b>
-Xmx2g	XX:+UseConcMarkSweepGC	65.29%	0.54668s
-Xmx2g	-XX:+UseG1GC	53.13%	0.51114s

# Java Concurrency- Thread Pool

A thread pool is represented by an instance of the class `ExecutorService`.

With an `ExecutorService`, we can submit task that will be completed in the future.



Here are the type of thread pools you can create with the Executors class :

- ✓ Single Thread Executor
- ✓ Cached Thread Pool
- ✓ Fixed Thread Pool
- ✓ Scheduled Thread Pool
- ✓ Single Thread Scheduled Pool

**Single Thread Executor** : A thread pool with only one thread. So all the submitted task will be executed sequentially.

Method : `Executors.newSingleThreadExecutor()`

**Cached Thread Pool** : A thread pool that create as many threads it needs to execute the task in parralel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method :

`Executors.newCachedThreadPool()`

**Fixed Thread Pool** : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method :

`Executors.newFixedThreadPool()`

**Scheduled Thread Pool** : A thread pool made to schedule future task. Method : `Executors.newScheduledThreadPool()`

Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task.

Method : `Executors.newSingleThreadScheduledExecutor()`

# JBoss 7 Thread Pool Configuration

Thread Pools address two different problems:

- deliver improved performance when executing large numbers of asynchronous tasks
  - due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including Threads, consumed when executing a collection of tasks.
- 
- Bounded Threads configuration
  - Unbounded Threads configuration
  - Queueless Thread Pool configuration
  - Scheduled Thread configuration

## Data source connection pool :

```
<datasource jndi-name="MySqlDS" pool-name="MySqlDS_Pool"
enabled="true" jta="true" use-java-context="true" use-ccm="true">
<connection-url>jdbc:mysql://localhost:3306/MyDB
</connection-url>
<driver>mysql</driver>
<pool>
<min-pool-size>10</min-pool-size>
<max-pool-size>30</max-pool-size>
<prefill>true</prefill>
</pool>
<timeout>
<blocking-timeout-millis>30000</blocking-timeout-millis>
<idle-timeout-minutes>5</idle-timeout-minutes>
</timeout>
</datasource>
```

In server log the below error is a strong clue that we need to look at connection pooling:

```
21:57:57,781 ERROR [stderr] (http-executor-threads - 7) Caused by:  
javax.resource.ResourceException: IJ000655: No managed connections  
available within configured blocking timeout (30000 [ms])
```

```
21:57:57,782 ERROR [stderr] (http-executor-threads - 7)      at  
org.jboss.jca.core.connectionmanager.pool.mcp.SemaphoreArrayListM  
anagedConnectionPool.getConnection
```

## EJB connection pool

```
<pools>
<bean-instance-pools>
<strict-max-pool name="slsb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-
unit="MINUTES"/>
<strict-max-pool name="mdb-strict-max-pool" max-pool-size="20"
instance-acquisition-timeout="5" instance-acquisition-timeout-
unit="MINUTES"/>
</bean-instance-pools>
</pools>
```



EJB does not need a costly initialization, then avoid object pool :

1.<stateless>

2.<!--

3.<bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>

4.-->

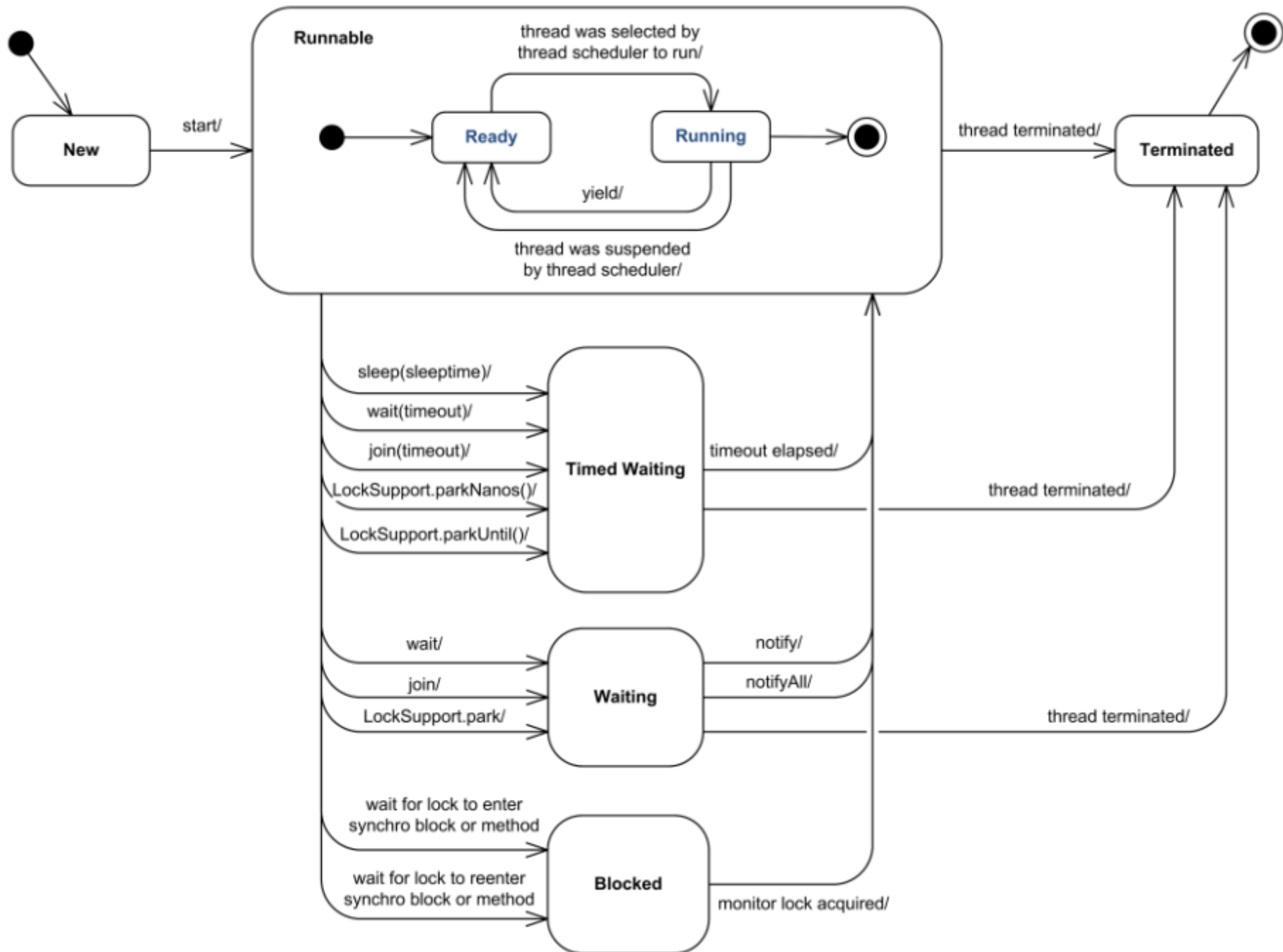
5.</stateless>

http thread pool

```
<subsystem xmlns="urn:jboss:domain:web:1.0">  
  
  <connector enable-lookups="false" enabled="true"  
    executor="http-executor"  
    max-connections="200"  
    max-post-size="2048" max-save-post-size="4096"  
    name="http" protocol="HTTP/1.1"  
    proxy-name="proxy" proxy-port="8081"  
    redirect-port="8443" scheme="http"  
    secure="false" socket-binding="http" />  
  
  ...  
</subsystem>
```

```
<subsystem xmlns="urn:jboss:domain:threads:1.1">
    <bounded-queue-thread-pool
name="http-executor">
        <core-threads count="25"/>
        <queue-length count="50"/>
        <max-threads count="100"/>
        <keepalive-time time="10" unit="seconds" />
    </bounded-queue-thread-pool>
</subsystem>
```

# Understanding Thread States



- **NEW** : A thread has not started yet.
- **RUNNABLE** : Thread is running state but it can be in state of waiting.
- **BLOCKED** : Thread is waiting to acquire monitor lock to enter into a synchronized block/method after calling `Object.wait()`
- **WAITING** : A thread is in waiting state due to calling one of the following methods
  - `Object.wait()` : It causes current thread to wait until it been notified by method *notify()* or *notifyAll()*.
  - `Object.join()` : Waits for current thread to die.
  - `LockSupport.park` : Disables the current thread for thread scheduling purposes unless the permit is available.

• **TIMED\_WAITING** : Current thread is waits for another thread for specified time to perform the action.

- **Thread.sleep (long timeInMilliSecond)** : Makes current thread to cease the execution for specified time.
- **Object.wait (long timeInMilliSecond)** : Causes current thread to wait for specified time until time elapsed or get notified by notify() or notifyAll().
- **Thread.join (long millis)** : Current thread waits for specified time to die the thread.
- **LockSupport.parkNanos (long nanoSeconds)** : Disables the current thread for thread scheduling purposes, for up to the specified waiting time, unless the permit is available.
- **LockSupport.parkUntil ()**

• **TERMINATED** : When thread completed its execution.

NEW : A thread is in a new state when it has just been created and the start() method hasn't been invoked

RUNNING : A thread is in a runnable state when the start method has been invoked and the JVM actively schedules it for execution

BLOCKED / WAITING FOR MONITOR ENTRY: Thread enters this state when it isn't able to acquire the necessary monitors to enter a synchronized block. A thread waits for other threads to relinquish the monitor. A thread in this state isn't doing any work



WAITING : A thread enters this state when it relinquishes its control over the monitor by calling `Object.wait()`. It waits till the time other threads invoke `Object.notify()` or `Object.notifyAll()`. In this state too the thread isn't doing any work

TIMED WAITING : A thread enters this state when it relinquishes its control over the monitor by calling `Object.wait(long)`. It waits till the time other threads invoke `Object.notify()` / `Object.notifyAll()` or when the wait time expires. In this state too the thread isn't doing any work

SLEEPING / WAITING ON CONDITION: A thread sleeps when it calls `Thread.sleep(long)`. As the name suggests the thread isn't doing in this state either

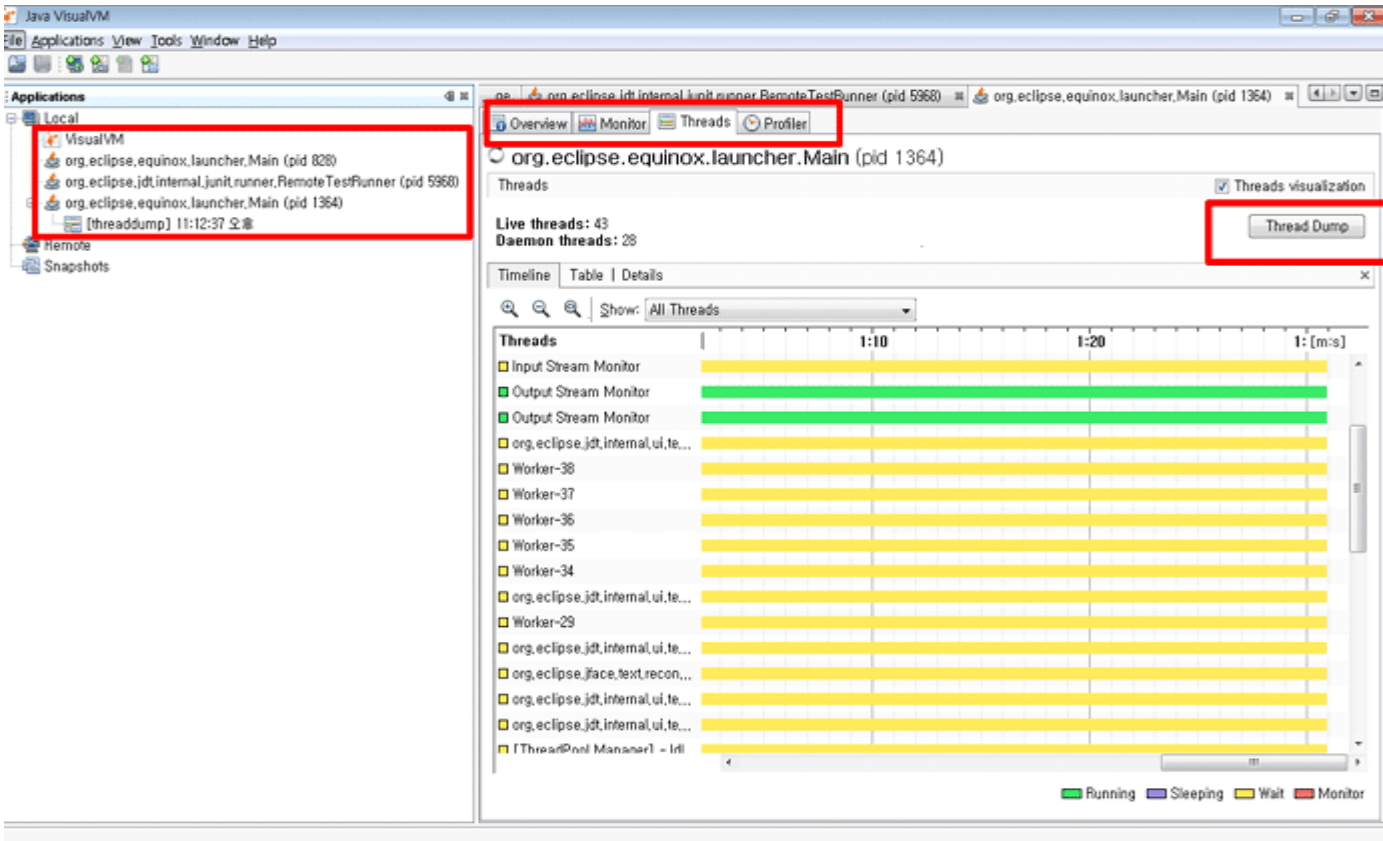
Terminated

Other

## **Analysing Thread Dumps**

# Thread Dump Using jVisualVM

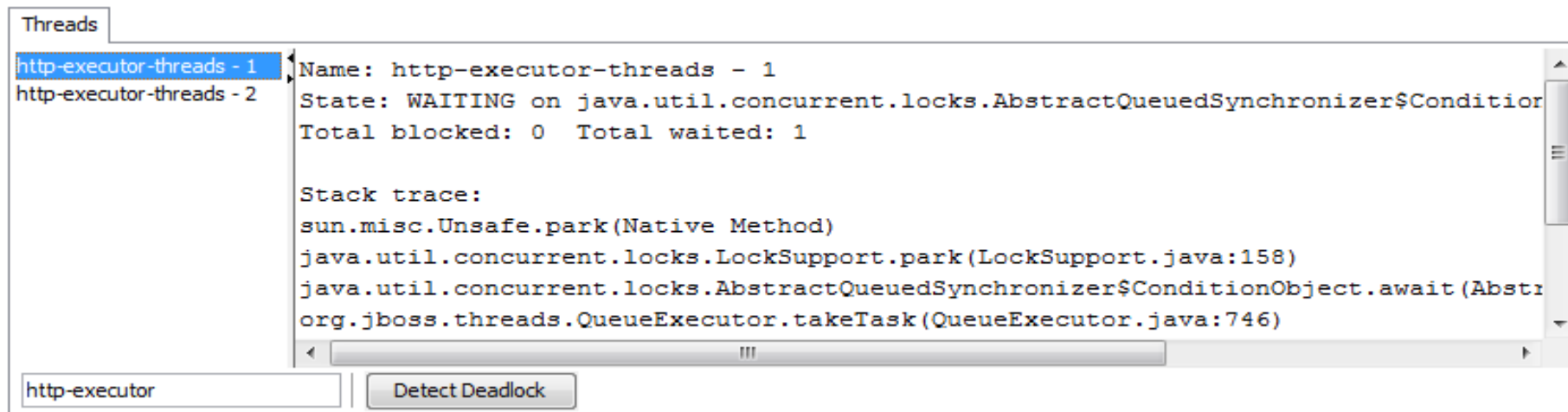
Generate a thread dump by using a program such as jVisualVM.



Ex1 : C:\Java\jdk1.7.0\bin\jvisualvm

Ex2 : C:\Java\jdk1.7.0\bin\jstat -f <process-id>

**idle thread** should look like, by looking at its stack trace:



The screenshot shows the 'Threads' window in a Java IDE. The window has a tab labeled 'Threads' and a list of threads on the left. Two threads are listed: 'http-executor-threads - 1' (highlighted in blue) and 'http-executor-threads - 2'. The main pane displays the details for the selected thread 'http-executor-threads - 1'. It shows the thread's name, state, and stack trace. The state is 'WAITING on java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject'. The stack trace shows the thread is waiting on a lock, with the following frames: sun.misc.Unsafe.park(Native Method), java.util.concurrent.locks.LockSupport.park(LockSupport.java:158), java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject.await(Abst..., org.jboss.threads.QueueExecutor.takeTask(QueueExecutor.java:746). At the bottom of the window, there is a text field containing 'http-executor' and a button labeled 'Detect Deadlock'.

Threads

http-executor-threads - 1  
http-executor-threads - 2

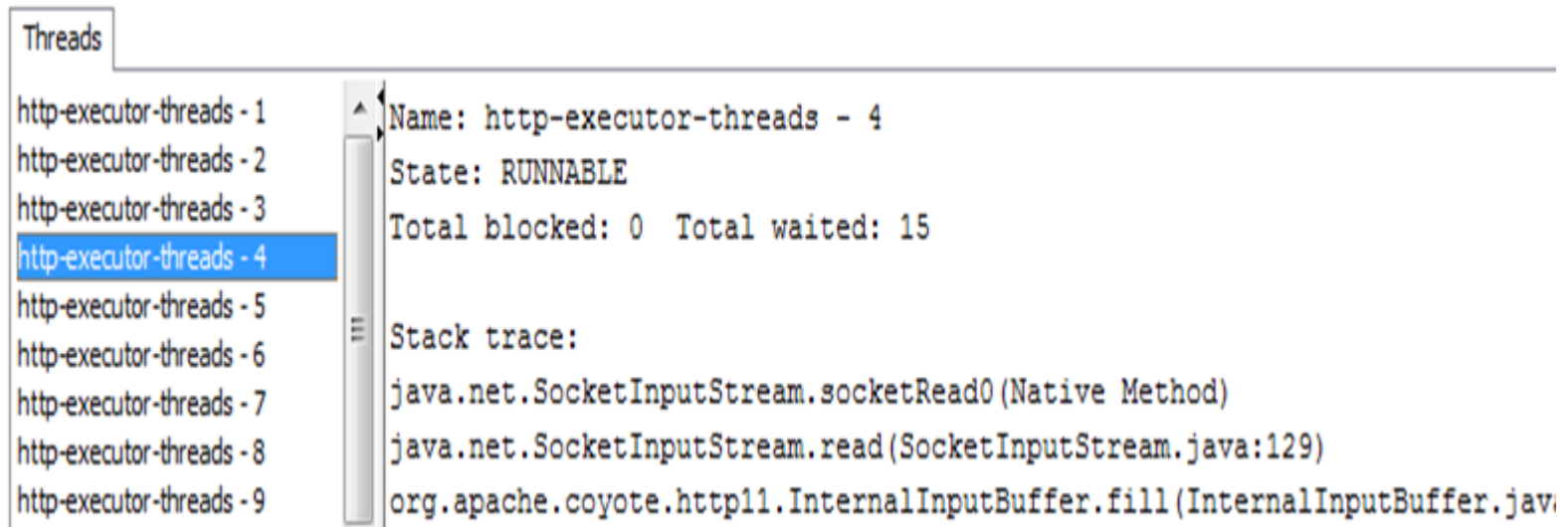
Name: http-executor-threads - 1  
State: WAITING on java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject  
Total blocked: 0 Total waited: 1

Stack trace:  
sun.misc.Unsafe.park(Native Method)  
java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)  
java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject.await(Abst...,  
org.jboss.threads.QueueExecutor.takeTask(QueueExecutor.java:746)

http-executor

Detect Deadlock

HTTP thread is **busy** at doing input/output operations which could mean, for example, the web server is acquiring data from an external resource.



The screenshot shows a Java IDE's 'Threads' window. On the left, a list of threads is displayed, with 'http-executor-threads - 4' selected and highlighted in blue. The right pane shows the details for this selected thread:

- Name: http-executor-threads - 4
- State: RUNNABLE
- Total blocked: 0 Total waited: 15
- Stack trace:
  - java.net.SocketInputStream.socketRead0 (Native Method)
  - java.net.SocketInputStream.read (SocketInputStream.java:129)
  - org.apache.coyote.http11.InternalInputBuffer.fill (InternalInputBuffer.java:111)

**Eliminate Runnable threads:** Runnable threads are already being actively scheduled and there is no need to extensively focus on them in the first pass.

**Eliminate Idle threads:** It is natural that some threads in the application are in the idle state. They would either be waiting for an incoming connection or sleeping for a period of time before they wake. Eliminate such threads from your analysis which would be ideally found idling cycles.

The below threads can be safely ignored :

**"tcpConnection-9011-7736"** daemon prio=1 tid=0x351750b8  
nid=0x7eab in **Object.wait()** [0x2fdf4000..0x2fdf50b0]  
at java.lang.Object.wait(Native Method)  
- waiting on <0x43eef7c8> (a java.lang.Object)  
at com.caucho.server.TcpServer.accept(TcpServer.java:648)

**"Store org.hibernate.cache.StandardQueryCache Expiry Thread"**

daemon prio=1 tid=0x3becd4e0 nid=0x1c6e **waiting on condition**  
[0x35389000..0x353891b0] at java.lang.Thread.sleep(Native Method)

**"Store org.hibernate.cache.UpdateTimestampsCache Expiry Thread"**

daemon prio=1 tid=0x397868a8 nid=0x1c6c **waiting on condition**  
[0x3548a000..0x3548b0b0] at java.lang.Thread.sleep(Native Method)



**Analyse BLOCKED and WAITING threads:** This is perhaps the most important part of analysis. It is important to ask the question why are these threads in either the BLOCKED or WAITING state when they should have been in the RUNNABLE state.

These threads are waiting for monitors which have been acquired by other threads.

"tcpConnection-9011-7709" in the BLOCKED state waiting to lock a monitor:

**"tcpConnection-9011-7709"** daemon prio=1 tid=0x35029e40 nid=0x7e61  
**waiting for monitor entry** [0x2fe74000..0x2fe75e30]  
at java.beans.Introspector.getPublicDeclaredMethods(Introspector.java:1249)  
- **waiting to lock <0x3d9bb6b0> (a java.lang.Class)**

"tcpConnection-9011-7640" has obtained the monitor.

**"tcpConnection-9011-7640"** daemon prio=1 tid=0x35177a40 nid=0x7dc7  
**runnable** [0x305fa000..0x305fcf30]  
at java.beans.Introspector.findExplicitBeanInfo(Introspector.java:410)  
- **locked <0x3d9bb6b0> (a java.lang.Class)**

## Analyse thread dumps over a duration

Compare the the sanapshot's :

Snapshot 1:

**"tcpConnection-9011-7696"** daemon prio=1 tid=0x3466c258  
nid=0x7e4f in **Object.wait()** [0x2727b000..0x2727bfb0]

Snapshot 2:

**"tcpConnection-9011-7696"** daemon prio=1 tid=0x3466c258  
nid=0x7e4f **runnable** [0x2727b000..0x2727bfb0]

## **out-of-threads**

Consider the following scenario:

- > Response time of your key pages is rapidly dropping
- > The application throughput is more or less constant
- > The application server and database servers are running with permissible limits of system parameters

# It is possible that you may be seeing a sudden increase of users hitting the application. Based on the analysis the only immediate recourse would be to increase the number of threads in the thread pool to cater to this increased demand. We need to consider the impact of such changes before making the changes.

## Resource Contention

Resource contention typically happens when the threads are fighting for the same resources. If a majority of threads in the application are fighting for the same resource this could speak about poor application design.

The most common symptoms of resource contention issues are :

- > The throughput / responsiveness of the application has fallen considerably and the application is too slow
- > The CPU utilization is low
- > All the application resources (db connection pools etc) seem to free and available
- > In thread dumps most the thread would appear in the BLOCKED / WAITING FOR MONITOR ENTRY state

Problem 1 :

all the threads are waiting to acquire lock on the monitor  
<0x44008de0>

**"tcpConnection-9011-8009"** daemon prio=1 tid=0x31446a80  
nid=0x49f5 **waiting for monitor entry** [0x2edf5000..0x2edf6e30]  
at  
com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadC  
ache.put(AbstractConcurrentReadCache.java:1648)  
**waiting to lock <0x44008de0>**

**This lock is obtained by the below thread:**

**" tcpConnection-9011-7817"** daemon prio=1 tid=0x24fee130  
nid=0x491c **runnable** [0x1513e000..0x15140130]  
at java.lang.System.identityHashCode(Native Method)  
**locked < 0x44008de0 >**

## Analysis :

operating system had run out of file descriptors and hence thread `tcpConnection-9011-7817` couldn't really persist the cached object to the disk and release the monitor.

## Solutions :

Increasing the limit on the number of file descriptors (`ulimit`) and restarting the application.

## Application Threads

All three threads are in a blocked state (indicated by blocked), which means that they are all trying to enter synchronized blocks. Thread-0 is trying to take Object@0x01226300[fat lock] but this is held by Thread-2.

Both Thread-2 and Thread-1 are trying to take Object@0x012262F8[thin lock] but this lock is held by Thread-0. This means that Thread-0 and Thread-2 form a deadlock, while Thread-1 is blocked.



"Thread-0" id=11 idx=0x1e tid=48408 prio=5 alive, in native, blocked

-- **Blocked trying to get lock: java/lang/Object@0x01226300[fat lock]**

at jrockit/vm/Threads.waitForSignal(J)Z(Native Method)

at jrockit/vm/Locks.fatLockBlockOrSpin(ILjrockit/vm/ObjectMonitor;II)V(Unknown Source)

at

jrockit/vm/Locks.lockFat(Ljava/lang/Object;ILjrockit/vm/ObjectMonitor;Z)Ljava/lang/Object;(Unknown Source)

at

jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)

at jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)

at util/repro/Thread1.run(DeadLockExample.java:34)

^-- **Holding lock: java/lang/Object@0x012262F0[thin lock]**

^-- **Holding lock: java/lang/Object@0x012262F8[thin lock]**

at jrockit/vm/RNI.c2java(IIII)V(Native Method)

-- end of trace

"Thread-1" id=12 idx=0x20 tid=48412 prio=5 alive, in native, blocked

-- **Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]**

at jrockit/vm/Threads.sleep(I)V(Native Method)

at jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown Source)

at jrockit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)

at jrockit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)

at util/repro/Thread2.run(DeadLockExample.java:48)

at jrockit/vm/RNI.c2java(IIII)V(Native Method)

-- end of trace

"Thread-2" id=13 idx=0x22 tid=48416 prio=5 alive, in native, blocked

-- **Blocked trying to get lock: java/lang/Object@0x012262F8[thin lock]**

at jrookit/vm/Threads.sleep(I)V(Native Method)

at jrookit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I(Unknown Source)

at

jrookit/vm/Locks.monitorEnterSecondStage(Ljava/lang/Object;I)Ljava/lang/Object;(Unknown Source)

at jrookit/vm/Locks.monitorEnter(Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)

at util/repro/Thread3.run(DeadLockExample.java:65)

^-- **Holding lock: java/lang/Object@0x01226300[fat lock]**

at jrookit/vm/RNI.c2java(IIII)V(Native Method)

-- end of trace

## **Problem :**

The below thread is unable to get database connection:

```
"Thread-1" prio=5 tid=0x00a861b8 nid=0xbdc in Object.wait()
[0x02d0f000..0x02d0fb68]
    at java.lang.Object.wait(Native Method)
    - waiting on <0x22aadfc0> (a
org.tw.testyard.thread.ConnectionPool)
    at java.lang.Object.wait(Unknown Source)
    at
org.tw.testyard.thread.ConnectionPool.getConnection(ConnectionPool.
java:39)
    - locked <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)
```

Solution :

We see such things happening often in your application it is time to revisit the connection pool configuration.

In any case the connection pool size should be at least as big as the number of worker threads in your application.

# We may see the similar issues when dealing with resources such as db connections, network connections or file handles.

Stuck Thread

What is Stuck Thread?

A Stuck Thread is a thread which is processing a request for more than maximum time that is configured in a server.

hogging thread

A hogging thread is a thread which is taking more than usual time to complete the request and can be declared as Stuck .

In Weblogic:

A thread declared as Stuck if it runs over 600 secs

WebLogic has polar which runs every 2 secs,It checks for the number of requests completed in last two minutes

check how much times each took to complete

Then, it takes the average time of all completed request

Then multiply average time with a the value of the request taken more time.



For example –

At a particular moment, total number of completed requests in last two seconds – 4

Total time took by all 4 requests – 16 secs

Req1 took – 5 secs, Req2 took – 3 secs, Req3 took – 7 secs, Req4 took – 1 sec

Average time =  $16/4 = 4$  secs

$7*4 = 28$  secs

Now weblogic check all executed threads to see which taking more than 28 secs, if any then that thread(s) declared as Hogged Thread.

Race Condition

A race condition occurs in programming when two or more execution threads modify a shared, or critical, resource.

Race conditions can result in run time errors that are difficult to isolate and to repair.

The term "race" is used because the threads can be regarded as racing each other to complete operations on a variable or other shared resource.

## Logging tuning

- >Choosing the appropriate handler to output your logs.
- >Choose a log level which provides just the amount of information you need and nothing else.
- >Choose an appropriate format for your logs

For persistence layer, fine tune :

1. L1 & L2 caches
2. Number of database hits (N+1 query problems)
3. Connection management
4. Lazy initialization exception

## Pizza delivery example

–

Do you want your pizza hot?

- 

Low latency

–

Or do you want your pizza to be inexpensive?

- 

High throughput – lots of pizzas per hour

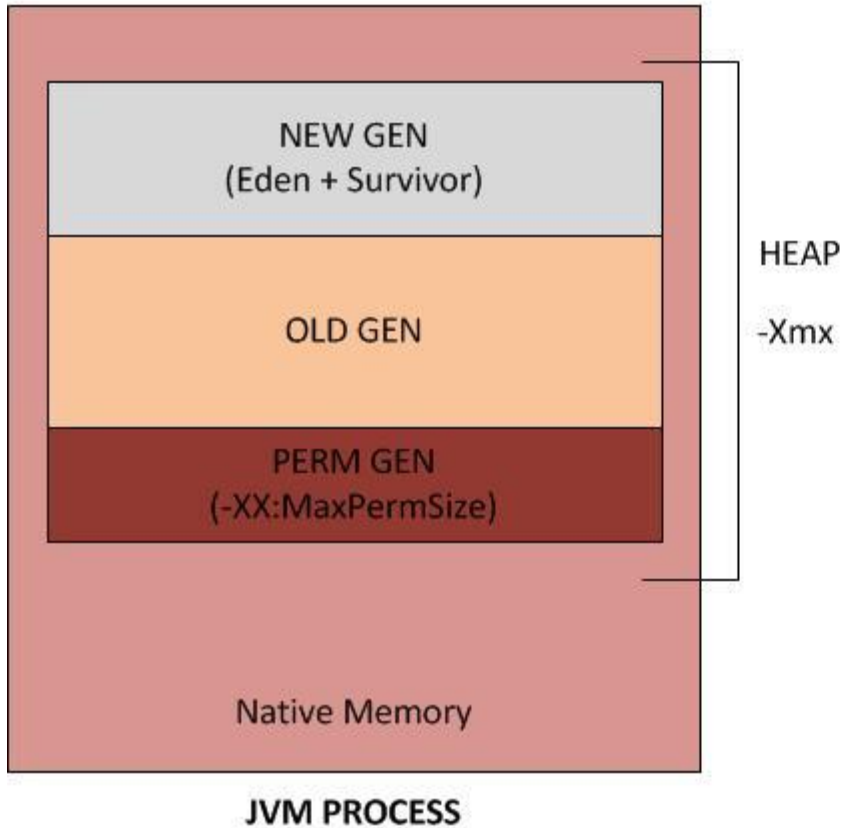
–

Two different delivery strategies for pizza company

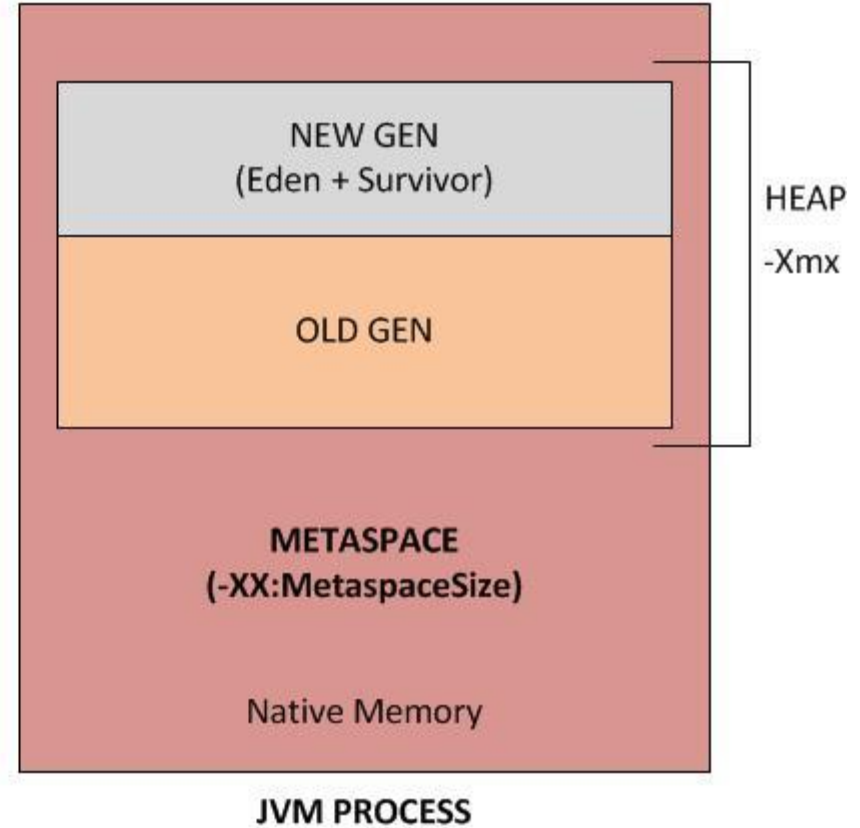
**Understand the OutOfMemoryError Exception**

## JAVA 8 MEMORY MANAGEMENT

Pre Java 8



Java 8





One common indication of a memory leak is the `java.lang.OutOfMemoryError` exception.

- ✓ Usually, this error is thrown when there is insufficient space to allocate an object in the Java heap. In this case, The garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further.

- ✓ Also, this error may be thrown when there is insufficient native memory to support the loading of a Java class.

- ✓ In a rare instance, a `java.lang.OutOfMemoryError` may be thrown when an excessive amount of time is being spent doing garbage collection and little memory is being freed.

**Exception in thread thread\_name:  
java.lang.OutOfMemoryError: Java heap space**

Cause: The detail message Java heap space indicates object could not be allocated in the Java heap.

This error does not necessarily imply a memory leak.

The problem can be as simple as a configuration issue, where the specified heap size (or the default size, if it is not specified) is insufficient for the application.

In other cases, and in particular for a long-lived application, the message might be an indication that the application is unintentionally holding references to objects, and this prevents the objects from being garbage collected.

This is the Java language equivalent of a memory leak.

Note: The APIs that are called by an application could also be unintentionally holding object references.

One other potential source of this error arises with applications that make excessive use of finalizers.

If a class has a finalize method, then objects of that type do not have their space reclaimed at garbage collection time. Instead, after garbage collection, the objects are queued for finalization, which occurs at a later time.

In the Oracle Sun implementation, finalizers are executed by a daemon thread that services the finalization queue.

If the finalizer thread cannot keep up, with the finalization queue, then the Java heap could fill up and this type of `OutOfMemoryError` exception would be thrown.

One scenario that can cause this situation is when an application creates high-priority threads that cause the finalization queue to increase at a rate that is faster than the rate at which the finalizer thread is servicing that queue.

Action :

The JConsole management tool can be used to monitor the number of objects that are pending finalization. This tool reports the pending finalization count in the memory statistics on the Summary tab pane. The count is approximate, but it can be used to characterize an application and understand if it relies a lot on finalization.

jmap utility can be used with the -finalizerinfo option to print information about objects awaiting finalization.

**Exception in thread thread\_name:  
java.lang.OutOfMemoryError: GC Overhead limit  
exceeded**

Cause: The detail message "GC overhead limit exceeded" indicates that the garbage collector is running all the time and Java program is making very slow progress.

After a garbage collection, if the Java process is spending more than approximately 98% of its time doing garbage collection and if it is recovering less than 2% of the heap and has been doing so far the last 5 (compile time constant) consecutive garbage collections, then a java.lang.OutOfMemoryError is thrown.

This exception is typically thrown because the amount of live data barely fits into the Java heap having little free space for new allocations.

Action: Increase the heap size.

The `java.lang.OutOfMemoryError` exception for GC Overhead limit exceeded can be turned off with the command line flag `-XX:-UseGCOverheadLimit`.



## **Exception in thread thread\_name: java.lang.OutOfMemoryError: Metaspace**

Cause: Java class metadata (the virtual machines internal presentation of Java class) is allocated in native memory (referred to here as metaspace).

If metaspace for class metadata is exhausted, a `java.lang.OutOfMemoryError` exception with a detail `MetaSpace` is thrown.

The amount of metaspace that can be used for class metadata is limited by the parameter `MaxMetaSpaceSize`, which is specified on the command line.

When the amount of native memory needed for a class metadata exceeds `MaxMetaSpaceSize`, a `java.lang.OutOfMemoryError` exception with a detail `MetaSpace` is thrown.

Action:

If MaxMetaSpaceSize, has been set on the command-line, increase its value.

MetaSpace is allocated from the same address spaces as the Java heap. Reducing the size of the Java heap will make more space available for MetaSpace

**Exception in thread thread\_name:  
java.lang.OutOfMemoryError: request size bytes  
for reason. Out of swap space?**

Cause: The detail message "request size bytes for reason. Out of swap space?" appears to be an OutOfMemoryError exception.

However, the Java HotSpot VM code reports this apparent exception when an allocation from the native heap failed and the native heap might be close to exhaustion.

The message indicates the size (in bytes) of the request that failed and the reason for the memory request. Usually the reason is the name of the source module reporting the allocation failure, although sometimes it is the actual reason.

Action: When this error message is thrown, the VM invokes the fatal error handling mechanism

In the case of native heap exhaustion, the heap memory and memory map information in the log can be useful.

we might need to use troubleshooting utilities on the operating system to diagnose the issue further

## **Exception in thread thread\_name: java.lang.OutOfMemoryError: Compressed class space**

Cause: On 64-bit platforms a pointer to class metadata can be represented by a 32-bit offset (with UseCompressedOops). This is controlled by the command line flag UseCompressedClassPointers (on by default). If the UseCompressedClassPointers is used, the amount of space available for class metadata is fixed at the amount CompressedClassSpaceSize. If the space needed for UseCompressedClassPointers exceeds CompressedClassSpaceSize, a java.lang.OutOfMemoryError with detail Compressed class space is thrown.

Action: Increase CompressedClassSpaceSize to turn off UseCompressedClassPointers. Note: There are bounds on the acceptable size of CompressedClassSpaceSize. For example -XX:CompressedClassSpaceSize=4g, exceeds acceptable bounds will result in a message such as

## **Exception in thread thread\_name: java.lang.OutOfMemoryError: reason stack\_trace\_with\_native\_method**

Cause: If the detail part of the error message is "reason stack\_trace\_with\_native\_method" and a stack trace is printed in which the top frame is a native method, then this is an indication that a native method has encountered an allocation failure. The difference between this and the previous message is that the allocation failure was detected in a Java Native Interface (JNI) or native method rather than in the JVM code.

Action: If this type of the OutOfMemoryError exception is thrown, we might need to use native utilities of the OS to further diagnose the issue.

## Monitoring Tools Overview

## jstat -class option

Class Loader Statistics	
Column	Description
Loaded	Number of classes loaded.
Bytes	Number of Kbytes loaded.
Unloaded	Number of classes unloaded.
Bytes	Number of Kbytes unloaded.
Time	Time spent performing class load and unload operations.



## -compiler Option

HotSpot Just-In-Time Compiler Statistics	
Column	Description
Compiled	Number of compilation tasks performed.
Failed	Number of compilation tasks that failed.
Invalid	Number of compilation tasks that were invalidated.
Time	Time spent performing compilation tasks.
FailedType	Compile type of the last failed compilation.
FailedMethod	Class name and method for the last failed compilation.

## -gc option

Column	Description
S0C	Current survivor space 0 capacity (KB).
S1C	Current survivor space 1 capacity (KB).
S0U	Survivor space 0 utilization (KB).
S1U	Survivor space 1 utilization (KB).
EC	Current eden space capacity (KB).
EU	Eden space utilization (KB).
OC	Current old space capacity (KB).
OU	Old space utilization (KB).
PC	Current permanent space capacity (KB).
PU	Permanent space utilization (KB).
YGC	Number of young generation GC Events.
YGCT	Young generation garbage collection time.
FGC	Number of full GC events.
FGCT	Full garbage collection time.
GCT	Total garbage collection time.

-gccause

Column	Description
LGCC	Cause of last Garbage Collection.
GCC	Cause of current Garbage Collection.

-printcompilation Option

Compiled -> Number of compilation tasks performed by the most recently compiled method.

Size->Number of bytes of bytecode of the most recently compiled method.

Type->Compilation type of the most recently compiled method.

Method->Class name and method name identifying the most recently compiled method. Class name uses "/" instead of "." as namespace separator. Method name is the method within the given class. The format for these two fields is consistent with the HotSpot - XX:+PrintCompilation option.

-gcutil

Column	Description
S0	Survivor space 0 utilization as a percentage of the space's current capacity.
S1	Survivor space 1 utilization as a percentage of the space's current capacity.
E	Eden space utilization as a percentage of the space's current capacity.
O	Old space utilization as a percentage of the space's current capacity.
P	Permanent space utilization as a percentage of the space's current capacity.
YGC	Number of young generation GC events.
YGCT	Young generation garbage collection time.
FGC	Number of full GC events.
FGCT	Full garbage collection time.
GCT	Total garbage collection time.

# Introduction To JMX

## **What is JMX ?**

Java Management Extensions (JMX) is an upcoming open technology specification that defines the management architecture, which enables managing of applications and services.

This technology also allows Java developers to integrate their applications with existing network management solutions.

## **Need for JMX**

Before JMX, instrumentation is done for each protocol, i.e. instrumentation will be done separately for SNMP access, HTTP access, RMI access and other protocols for the same manageable information.

## What is Instrumentation ?

For example, let us take a manageable parameter of a Servlet, say number of concurrent users. This specifies the load or utilization of the servlet or Web application.

Let us say you have a servlet method as listed below:

```
class MyServlet{
    static int concurrentusers; //instrumentation code

    public xxx service(...)
    {
        // start of the method
        concurrentusers++; //instrumentation code

        // end of the method  }

    public int getConcurrentusers()
    {
        return MyServlet.concurrentusers; //instrumentation cod  }}
```

Thus the "concurrentUsers" count can be accessed with any protocol with the above instrumentation done just once. Thus, "Instrument Once and Access with any Protocol" is made possible with the advent of JMX.

The following points highlights the need for JMX:

JMX allows common instrumentation of management information which is protocol-neutral.

JMX allows a centralized management of managed beans, or MBeans, which act as wrappers for applications, components, or resources in a distributed network.

JMX is becoming a core part of application development and management.

JMX is widely adopted nowadays. It is built into Application Servers.

JMX is Java-based technology.

JMX is applicable to many domains -- an application, an EJB, a JSP ...

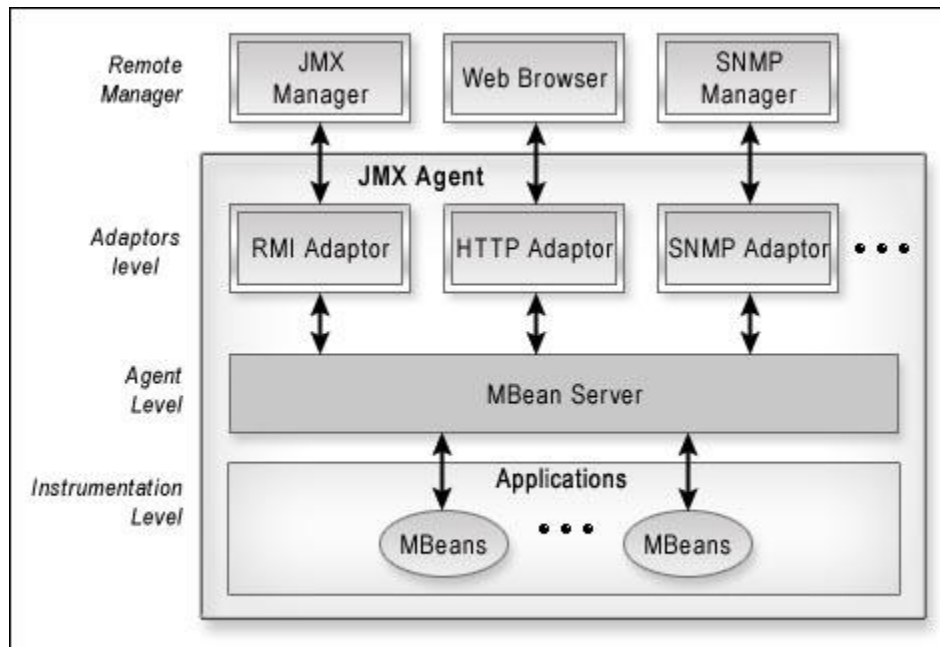


# JMX Architecture

As per JMX specification, JMX architecture is divided into four levels:

1. Instrumentation level
2. Agent level
3. Adaptors level
4. Remote Manager/ distribution level

A diagram representing the JMX architecture is shown below:



The instrumentation level defines an entity known as an MBean. MBeans represent the managed resources. Thus, the instrumentation level provides a way to access the managed resources (application) via these MBeans.

The Adaptors level is not completely defined in the JMX specification. This level contains the components which help in communicating the MBeans. These components are called protocol adaptors and connectors.

The agent level contains the component called MBeanServer which aids in the communication between the adaptors level and the instrumentation level. The agent level also provides a set of services that can be used by the management clients.

Remote Manager is where the managers reside. These managers allow user to view the management information and also to manage the resource.

The managers will communicate with the Adaptors level using a common protocol.

The JMX Manager and the RMI Adaptor communicate using RMI protocol.

The Web browser and the HTTP Adaptor communicate using HTTP Protocol. The SNMP Manager and the SNMP Adaptor communicate using SNMP Protocol.

## **Instrumentation Level**

A manageable resource can be a "Servlet, JSP, EJB, Java class, Java method, Java variable, or any legacy non-Java applications". A JMX manageable resource is developed in Java, or at least offers a Java wrapper, and has been instrumented so that it can be managed by JMX-compliant applications.

The instrumentation of a manageable resource is provided by one or more Managed Beans, or MBeans. The instrumentation level provides a specification for defining the MBeans. In addition, the instrumentation level also specifies a notification mechanism which allows MBeans to generate and propagate notification events to components of other levels.

## **What is an MBean ?**

Managed Bean (MBean) represents a manageable resource. An MBean is a Java object, which follows some semantics. The fields or properties of this Java object are called attributes. The methods of this Java object are called operations.

### **Definition of MBean**

MBeans are the JMX objects which exposes the management information in the form of attributes and operations.

### **Semantics for an MBean**

An MBean must be a public, non-abstract class.

An MBean must have atleast one public constructor.

An MBean must implement its own corresponding MBean interface or implement the `javax.management.DynamicMBean` interface.

Optionally, an MBean can implement the `javax.management.NotificationBroadcaster` interface.

## **Types of MBeans**

MBeans can be classified into four types. They are

1. Standard MBean
2. Dynamic MBean
3. Model MBean
4. Open MBean