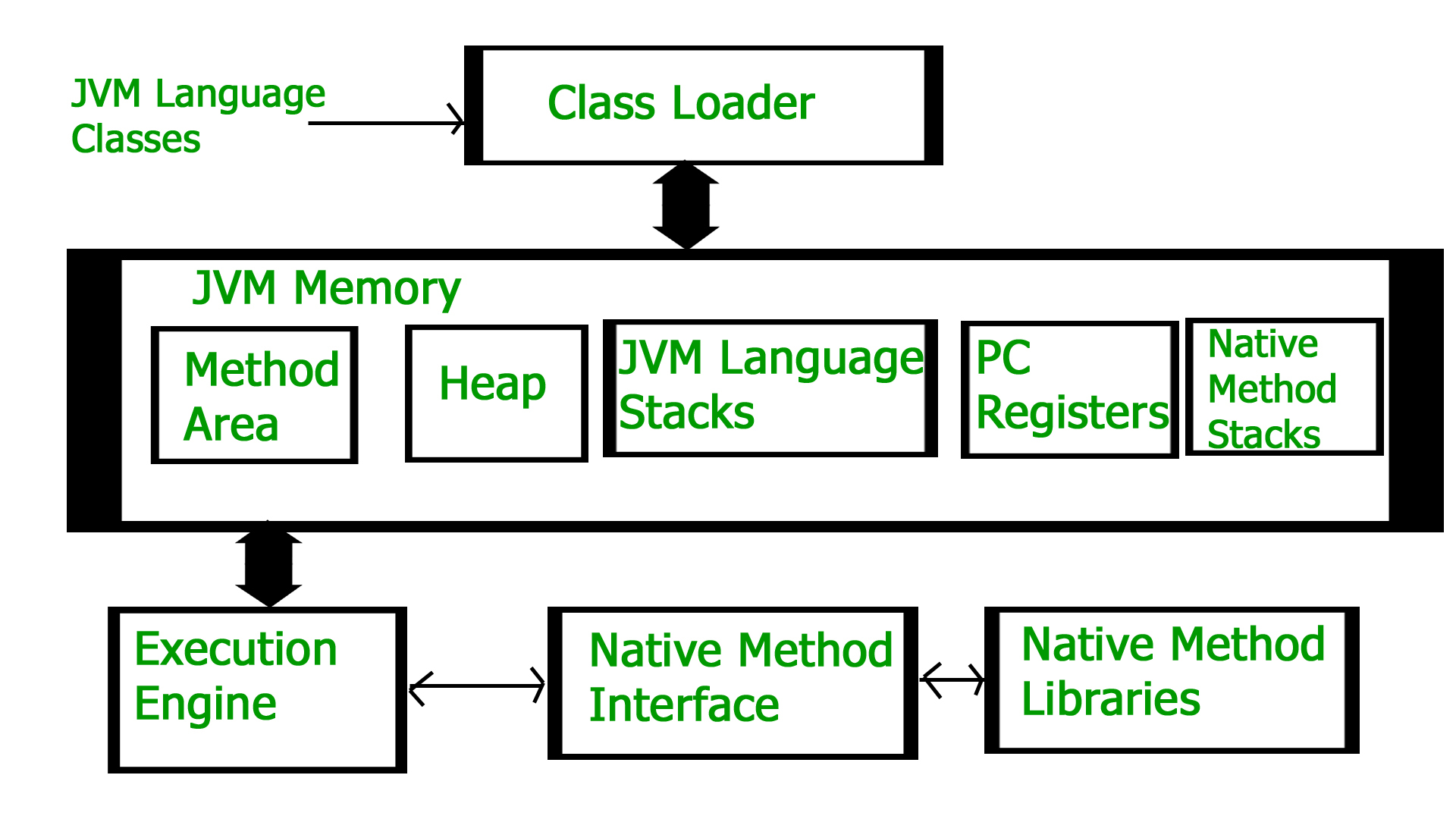
### **Graphs shown in the grafana of JMX-exporter represent:**

### **How JVM Works – JVM Architecture?**

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM. ****

## 

## 

## 

## 

## In every programming language, memory is a vital resource and is

## also scarce in nature. Hence it’s essential that the memory is managed thoroughly without any leaks. [Allocation and deallocation of memory](https://www.geeksforgeeks.org/java-memory-management/) is a critical task and requires a lot of care and consideration. In this article, we will understand the storage of [static methods](https://www.geeksforgeeks.org/static-methods-vs-instance-methods-java/) and [static variables](https://www.geeksforgeeks.org/difference-between-static-and-non-static-variables-in-java/) in Java. [Java Virtual Machine(JVM)](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) is an engine that provides a run time environment to drive the java code. It converts the java byte code into machine language. The JVM has two primary functions. They are:

## It allows java programs to run on any device or OS to fulfil WORA (Write Once Run Anywhere) principle.

## It manages and optimizes program memory.

## The JVM memory manager creates memory pools during the runtime of the program. Let’s understand the concept of memory pools in java. There are two types of memory pools namely the [stack memory and the heap memory](https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/). The main difference between stack memory and the heap memory is that the stack is used to store only the small datatypes whereas heap stores all the instances of the class. And, since java either implicitly or explicitly extends the class from Object.class, every class we create is a collection of objects. This also means that we cannot use a method or a variable unless we instantiate it with a new keyword. As soon as we instantiate the methods, JVM allocates some memory on the heap and it stores the address of the instance on the stack. After this, the methods and variables can be used. In order to get a better understanding of how the new keyword works, let’s take an example. Let’s take a bird class. Whenever a new bird is found, the number of birds need to be incremented. Let’s try to implement this code:

## **Java Heap Space**

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create an object, it’s always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that don’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

It is created when the JVM starts up and used by the application as long as the application runs. It stores objects and JRE classes. Whenever we create objects it occupies space in the heap memory while the reference of that object creates in the stack. It does not follow any order like the stack. It dynamically handles the memory blocks. It means, we need not to handle the memory manually. For managing the memory automatically, [Java](https://www.javatpoint.com/java-tutorial) provides the [garbage collector](https://www.javatpoint.com/Garbage-Collection) that deletes the objects which are no longer being used. Memory allocated to heap lives until any one event, either program terminated or memory free does not occur. The elements are globally accessible in the application.

The heap memory is further divided into the following memory areas:

* Young generation
* Survivor space
* Old generation
* Permanent generation
* Code Cache

### **Java Stack Memory**

The stack memory is a physical space (in RAM) allocated to each thread at run time. It is created when a thread creates. Memory management in the stack follows LIFO (Last-In-First-Out) order because it is accessible globally. It stores the variables, references to objects, and partial results. Memory allocated to stack lives until the function returns. If there is no space for creating the new objects, it throws the java.lang.StackOverFlowError. The scope of the elements is limited to their threads. The [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates a separate stack for each thread.

Java Stack memory is used for the execution of a thread. They contain method-specific values that are short-lived and references to other objects in the heap that is getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as the method ends, the block becomes unused and becomes available for the next method. Stack memory size is very less compared to Heap memory.

## 

### **Heap and Stack Memory in Java Program**

Let’s understand the Heap and Stack memory usage with a simple program.

package com.journaldev.test;

public class Memory {

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

} // Line 9

private void foo(Object param) { // Line 6

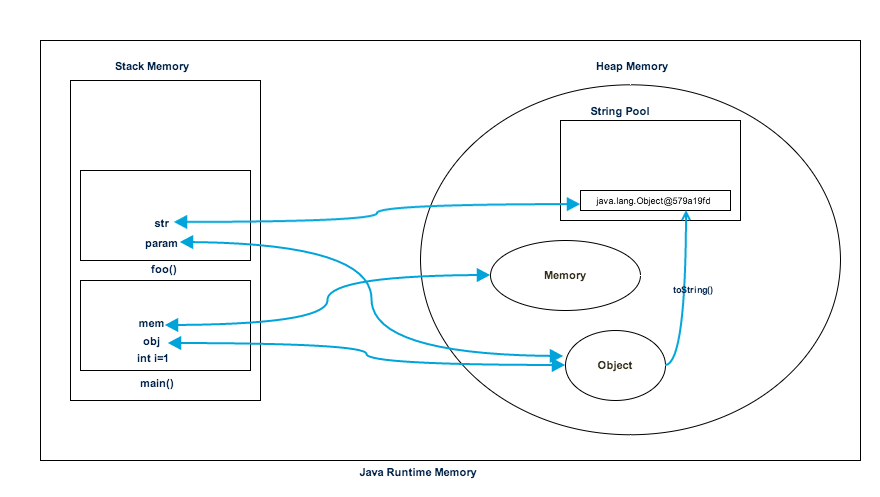
String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8

}

The below image shows the Stack and Heap memory with reference to the above program and how they are being used to store primitive, Objects and reference variables



* As soon as we run the program, it loads all the Runtime classes into the Heap space. When the main() method is found at line 1, Java Runtime creates stack memory to be used by the main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in the 3rd line, it’s created in heap memory and stack memory contains the reference for it. A similar process occurs when we create Memory object in the 4th line.
* Now when we call the foo() method in the 5th line, a block in the top of the stack is created to be used by the foo() method. Since Java is pass-by-value, a new reference to Object is created in the foo() stack block in the 6th line.
* A string is created in the 7th line, it goes in the [String Pool](https://www.digitalocean.com/community/tutorials/what-is-java-string-pool) in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in the 8th line, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also, the program ends at this line, hence Java Runtime frees all the memory and ends the execution of the program.

## **Difference between Java Heap Space and Stack Memory**

Based on the above explanations, we can easily conclude the following differences between Heap and Stack memory.

1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at [Java Garbage Collection](https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-management-in-java).
5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
6. We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

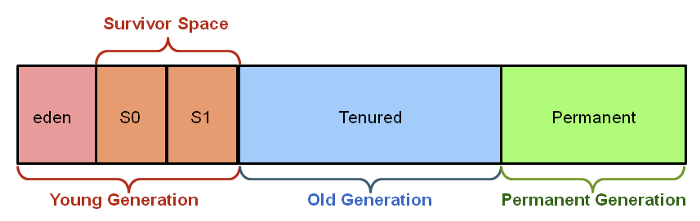
# **Garbage collection and Java**

In programming languages like C, developers have to take close care of memory by allocating and deallocating it accordingly.

Java, on the other hand, has its own **automatic garbage collection** with the idea that most developers don’t have to worry too much about such manual memory tasks, if at all.

In a nutshell, objects in Java are allocated in a **heap** (also known as **heap memory**). Java’s garbage collection algorithm then **goes through the heap** and **marks objects that are in use** by the JVM and then later on **goes back and reclaims the memory of any object that was not marked**.

What is important is the marking itself and the fact that the heap is allocated into smaller parts, called **generations**.

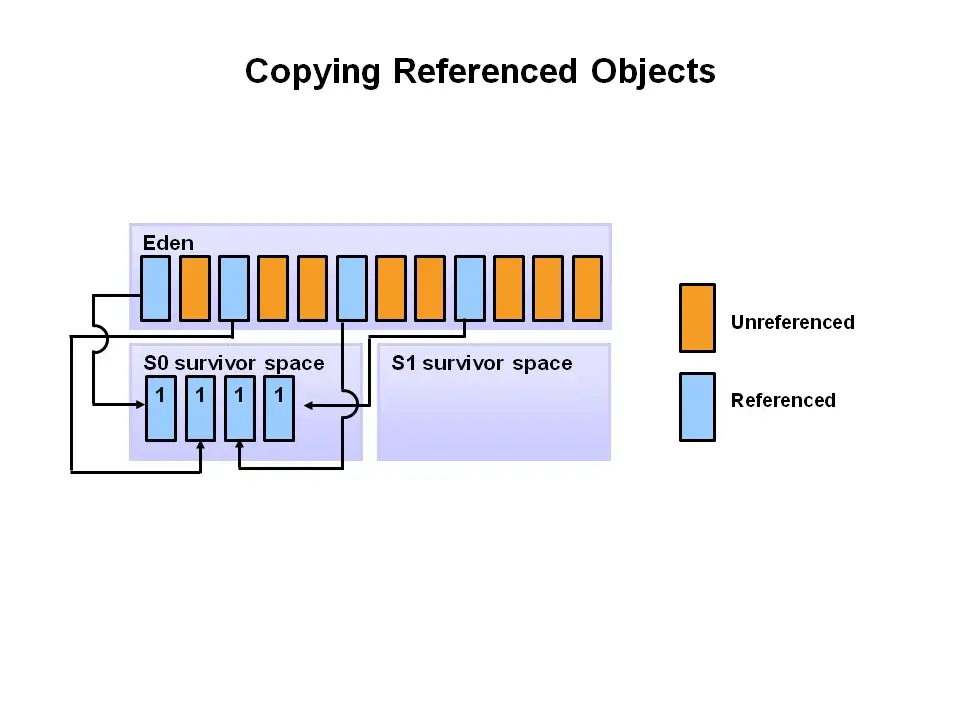


# **The Young Generation**

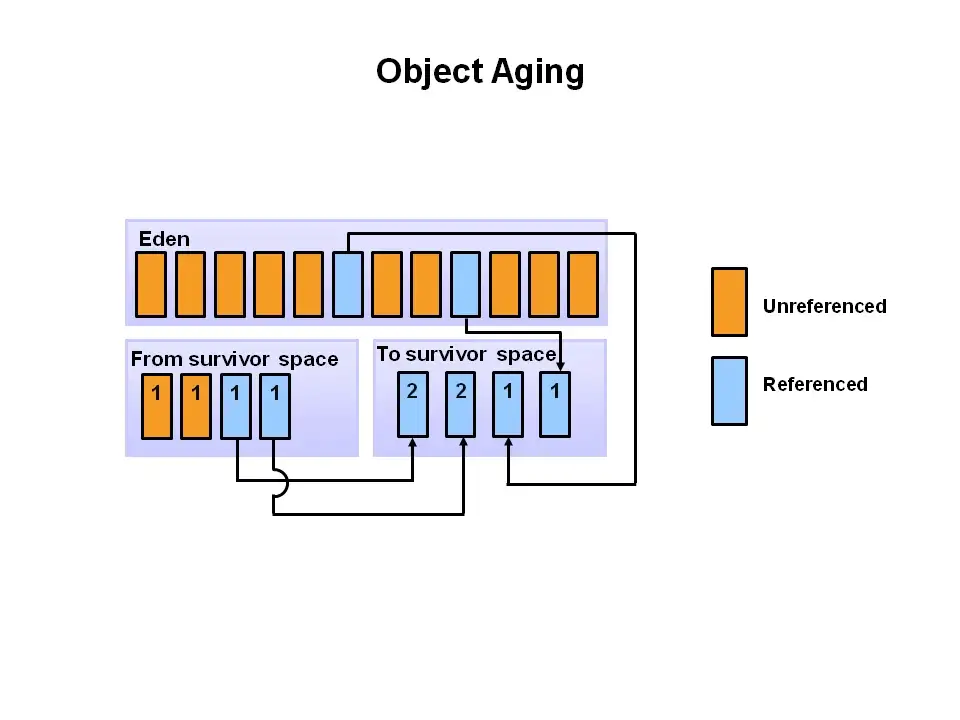
The **young generation** is where all new objects start out (starts). Once they’re allocated in the Java code, they go specifically to this subsection called the **eden space**.

Eventually, the eden space fills up with objects. At this point, a **minor garbage collection** event occurs.

That’s where the **marking algorithm** comes into play. Some objects (those that are **referenced**) are marked, and some (those that are **unreferenced**) are not. Those that had been marked then move onto another subsection of the young generation called **S0** of the **survivor space** (note that the survivor space itself is split into two parts, **S0 and S1**). Those left unmarked are cleared out by Java’s automatic garbage collection.



It stays this way until the eden space fills up again; at this point, a new cycle kicks off. The previous paragraph’s events **happen again**, but in this cycle, it’s a little different. **S0** was populated, and so all marked objects that survive from both the eden space and S0 actually go to the second part of the survivor space called **S1.** In the below diagram, we see that they’re labeled the **from survivor space** and the **to survivor space**, respectively.



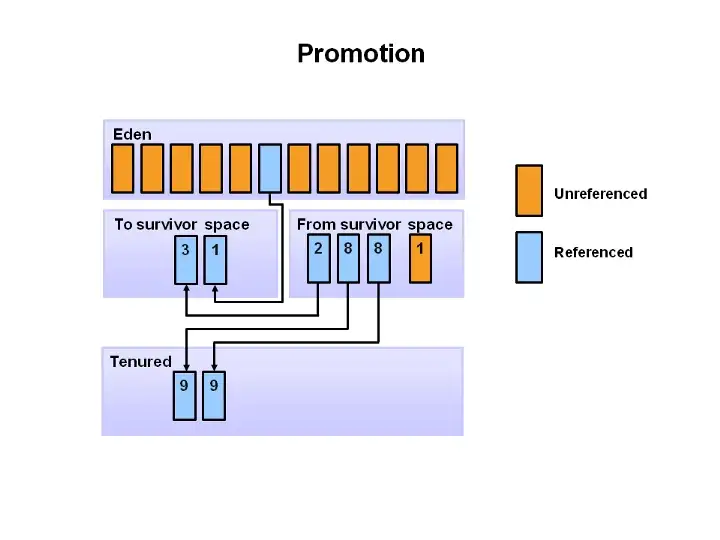
**essentially all new objects start out in the eden space** and then **eventually make their way into a survivor space** as they survive garbage collection cycles.

# **The Old Generation**

The old generation can be thought of as **where long-lived objects lie**. Basically, if objects **reach a certain age threshold** **after multiple garbage collection events in the young generation**, then they can then be moved to the old generation.

When objects get garbage collected from the old generation, a **major garbage collection event** occurs.

Let’s see what a **promotion from the survivor space of the young generation to the old generation** looks like.



The old generation is comprised of only one section called the **tenured generation**. This is why in conversation or in reading sometimes the two terms have come to be mostly interchangeable.

**The events that lead to a clearing of the old generation — again, a major garbage collection event.**

# **The Permanent Generation**

The **permanent generation is *not* populated when the old generation’s objects reach a certain threshold** and then get moved (promoted) to the permanent generation. Again, it doesn’t work this way!

Rather, the permanent generation is **immediately filled up by the JVM** with the metadata that represents the applications’ **classes and methods at runtime**.

The JVM may sometimes follow certain rules to clean out the permanent generation, and when it does, it’s called a **full garbage collection**.

# **What is a “stop the world” event?**

A “stop the world” event sounds pretty dramatic, but think of it in terms of the Java application being the world.

When there’s a **minor garbage collection** (remember: for the young generation) or a **major garbage collection** (for the old generation), then the world stops; in other words, all application threads are completely stopped and have to wait for the garbage collection event to complete.

**Java - Metaspace vs PermGen space**

# java14_1

# 

**Java Metaspace:**

So what is Metaspace and how is it different from Permgen Space?

With JDK8, the permGen Space has been removed. So where will the metadata information be stored now? This metadata is now stored in a native memory called "MetaSpace". This memory is not a contiguous Java Heap memory. It allows for improvements over PermGen space in Garbage collection, auto tuning, concurrent de-allocation of metadata.

**Difference between PermGen space and MetaSpace.**

| **PermGen Space** | **MetaSpace** |
| --- | --- |
| PermGen always has a fixed maximum size. | Metaspace by default auto increases its size depending on the underlying OS. |
| Contiguous Java Heap Memory | Native Memory(provided by underlying OS) |
| Max size can be set using XX:MaxPermSize | Max size can be set using XX:MetaspaceSize |
| Comparatively ineffiecient Garbage collection. Frequent GC pauses and no concurrent deallocation. | Comparatively effiecient Garbage collection. Deallocate class data concurrently and not during GC pause. |

## **What Is the Code Cache?**

## Simply put, JVM Code Cache is an area where JVM stores its bytecode compiled into native code. We call each block of the executable native code a nmethod. The nmethod might be a complete or inlined Java method.

## The just-in-time (JIT) compiler is the biggest consumer of the code cache area. That's why some developers call this memory a JIT code cache.

## **Code Cache Tuning**

## The code cache has a fixed size. Once it is full, the JVM won't compile any additional code as the JIT compiler is now off. Furthermore, we will receive the “CodeCache is full… The compiler has been disabled” warning message. As a result, we'll end up with degraded performance in our application. To avoid this, we can tune the code cache with the following size options:

## InitialCodeCacheSize – the initial code cache size, 160K default

## ReservedCodeCacheSize – the default maximum size is 48MB

## CodeCacheExpansionSize – the expansion size of the code cache, 32KB or 64KB

## Increasing the ReservedCodeCacheSize can be a solution, but this is typically only a temporary workaround.

**Status**: represents the current state of a JMX resource, such as a MBean or an attribute. It can be either "Up" or "Down", depending on the current health of the resource.

**State**: represents the current status of a JMX resource, such as a server, application, or service, as reported by the JMX agent.

**Cpu Time**: represents the total amount of time spent by the CPU executing instructions on behalf of the process or application.

**Cpu system load avg**: represents the average system load over a period of time. It is a measure of the number of computing resources being used on the system

**Cpu cores**: represents the number of physical cores available on the system.

**Total RAM**: represents the total amount of RAM available on the JMX Dashboard

**Total SWAP**: it shows the total amount of swap space used by the system. It is the cumulative amount of swap space used since the system was last booted.

**Cpu Time**: CPU time is a metric used to measure the amount of time the CPU spends in each mode, such as user mode, kernel mode, idle mode, etc.

**Memory Utilisation**: represents the amount of physical memory in use by the application or service. It is used to monitor the memory usage of the application or service over time and can help detect potential memory leaks.

**Swap Utilisation**: it is a metric that monitors the amount of memory being used in the swap space on a system.

**CPU load**: represents the amount of work the CPU is currently doing.

**Physical memory**: Physical Memory is a metric that measures the total amount of physical memory available to the system. It is usually reported in bytes, but can also be reported in other units such as kilobytes or megabytes. It is a key indicator of system performance and can be used to identify memory-related issues.

**Process Memory**: represents the memory used by a process running on the system. It is made up of physical memory and virtual memory, and it is tracked by the MemoryMXBean interface in JMX.

**JVM Memory Used[heap]**: represents the amount of memory that is being used by the Java virtual machine (JVM) to store information in its heap. This heap is used to store objects created by the application and the memory allocated to the heap can be adjusted using the JVM’s command line arguments.

**JVM Memory Used[non heap]**: represents the amount of memory used by the JVM for non-heap purposes. This includes memory used for garbage collection, thread stacks, and other internal operations.

**JVM Memory Usage[heap]:** represents the total memory used by the Java Virtual Machine (JVM) heap, which is the area of memory used for storing objects in the application.

**JVM Memory Usage[non heap]**: represents the memory usage of the non-heap memory areas of the JVM. This includes the memory used by classes and the permanent generation.

**JVM Memory committed:** represents the total amount of memory currently allocated to the Java Virtual Machine (JVM). It is the total amount of memory currently allocated to the JVM for memory management purposes, including the memory for object allocation, class loading, and garbage collection.

**Code Cache**: represents a memory pool that stores the compiled bytecode of Java classes. It is used for caching the compiled code so that it can be quickly reused without having to recompile the same class over again.

**Compressed Class Space**: it is a memory pool within the Java Virtual Machine (JVM) that is used for storing classes that have been compressed to save memory. This memory pool helps to improve the efficiency of the JVM by reducing the amount of memory used for the classes that are loaded.

**Eden Space:** it represents the area of memory that is used to store objects that are newly created by the JVM. This area is where most of the new objects are initially stored, and it is also where most of the garbage collection operations take place.

**Metaspace**: represents the non-heap memory areas of the JVM that are used for class meta-data. It includes the memory for class definitions and associated meta-data for the loaded classes.

**Survivor Space**: it is a pool of memory in the JVM that is used to store objects that have survived the garbage collection process. It is a part of the memory management process used by the JVM to manage and allocate memory resources.

**Tenured Gen**: it represents the memory pool that is used by the JVM to store objects that have been allocated memory and are no longer in use. This pool also holds objects that have been allocated long-term memory and that are not eligible for garbage collection.

**Garbade Collector**

**GC count increase**: GC count increase in the JMX dashboard represents the number of garbage collection operations that have occurred since the Java application was started. Garbage collection is the process of reclaiming memory from objects that are no longer being used.

**GC Time rate**: GC time rate represents the rate at which the garbage collector is running on the JMX dashboard. It measures the amount of time spent in garbage collection activities as a percentage of the total amount of time the application has been running.

**GC Time increase**: GC time increase represents the amount of time spent by the Java Virtual Machine (JVM) to perform garbage collection activities.

**Threading**

**JVM Threads count**: JVM threads count in JMX Dashboard represents the number of threads that are currently running in the Java Virtual Machine. It includes threads from the application, the JVM itself, and any other native threads that the JVM is running.

**JVM Threads count increased:** JVM thread count increased in JMX Dashboard represents the total number of threads that are running in the JVM.

J**VM Threads:** JVM Threads represent the threads being used by the JVM to execute code.

**JVM Threads increased:** JVM threads usually represent an increase in the number of threads that are being used by the JVM.

**##Classes**

**Class Loading count**: Class Loading count is a metric that is displayed in the JMX dashboard and represents the number of classes that have been loaded into the JVM. The higher the count, the more classes the JVM has loaded, which can indicate that the JVM is performing better.

**Class Loading count increase**: It is a representation of the total number of classes that have been loaded into the JVM so far. It is a measure of the performance of the JVM and helps to identify potential memory or class loading issues.

**Class Loaded**: Class Loaded is a JMX metric that tracks the total number of classes loaded in the JVM. This metric is usually used to measure the efficiency of the class loading subsystem, as well as to monitor memory usage.

**Class Loaded increase:** Class Loaded increase represents the number of classes that have been loaded by the JVM since the start of the application.

**EXTRAS:**

**Q1. Why does CPU usage always in %?**

CPU usage is always expressed as a percentage because it is a measure of how much of the total available computing power is being used at any given moment. It is expressed as a percentage because it is a relative measure rather than an absolute one, allowing us to compare the number of computing resources being used at different times.

**Q2. Why is RAM always shown in absolute value?**

RAM is always expressed in absolute values because it is a measure of the amount of memory available to the system. Unlike CPU usage, RAM is an absolute measure, meaning that the amount of RAM available will remain the same regardless of the amount of processing power being used.

**Q3. Can we find the absolute value of CPU usage?**

No, it is not possible to find an absolute value of CPU usage. CPU usage is a relative measure of how much of the total available computing power is being used at any given moment, and therefore cannot be expressed in absolute values.

**System:**

**Status**: represents the current state of a JMX resource, such as a MBean or an attribute. It can be either "Up" or "Down", depending on the current health of the resource.

**State**: represents the current status of a JMX resource, such as a server, application, or service, as reported by the JMX agent.

**Cpu Time**: represents the total amount of time spent by the CPU executing instructions on behalf of the process or application.

**Cpu system load avg**: represents the average system load over a period of time. It is a measure of the number of computing resources being used on the system

**Cpu cores**: represents the number of physical cores available on the system.

**Total RAM**: represents the total amount of RAM available on the JMX Dashboard

**Total SWAP**: it shows the total amount of swap space used by the system. It is the cumulative amount of swap space used since the system was last booted.

**Open file description**: represents the number of open files or sockets associated with a particular process. The dashboard displays the number of open files and sockets in a graphical format, allowing users to quickly identify any potential performance issues that may be caused by an excessive amount of open files.

**Cpu Time**: CPU time is a metric used to measure the amount of time the CPU spends in each mode, such as user mode, kernel mode, idle mode, etc.

**Memory Utilisation**: represents the amount of physical memory in use by the application or service. It is used to monitor the memory usage of the application or service over time and can help detect potential memory leaks.

**Swap Utilisation**: it is a metric that monitors the amount of memory being used in the swap space on a system.

**CPU load**: represents the amount of work the CPU is currently doing.

**Open File Descriptions**: represents the number of open files associated with each JMX Dashboard.

**Memory**

**Physical memory**: Physical Memory is a metric that measures the total amount of physical memory available to the system. It is usually reported in bytes, but can also be reported in other units such as kilobytes or megabytes. It is a key indicator of system performance and can be used to identify memory-related issues.

**Process Memory**: represents the memory used by a process running on the system. It is made up of physical memory and virtual memory, and it is tracked by the MemoryMXBean interface in JMX.

**Memory JVM**

**JVM Memory Used[heap]**: represents the amount of memory that is being used by the Java virtual machine (JVM) to store information in its heap. This heap is used to store objects created by the application and the memory allocated to the heap can be adjusted using the JVM’s command line arguments.

**JVM Memory Used[non heap]**: represents the amount of memory used by the JVM for non-heap purposes. This includes memory used for garbage collection, thread stacks, and other internal operations.

**JVM Memory Usage[heap]:** represents the total memory used by the Java Virtual Machine (JVM) heap, which is the area of memory used for storing objects in the application.

**JVM Memory Usage[non heap]**: represents the memory usage of the non-heap memory areas of the JVM. This includes the memory used by classes and the permanent generation.

**JVM Memory committed:** represents the total amount of memory currently allocated to the Java Virtual Machine (JVM). It is the total amount of memory currently allocated to the JVM for memory management purposes, including the memory for object allocation, class loading, and garbage collection.

**Memory JVM Pool**

**Code Cache**: represents a memory pool that stores the compiled bytecode of Java classes. It is used for caching the compiled code so that it can be quickly reused without having to recompile the same class over again.

**Compressed Class Space**: it is a memory pool within the Java Virtual Machine (JVM) that is used for storing classes that have been compressed to save memory. This memory pool helps to improve the efficiency of the JVM by reducing the amount of memory used for the classes that are loaded.

**Eden Space:** it represents the area of memory that is used to store objects that are newly created by the JVM. This area is where most of the new objects are initially stored, and it is also where most of the garbage collection operations take place.

**Metaspace**: represents the non-heap memory areas of the JVM that are used for class meta-data. It includes the memory for class definitions and associated meta-data for the loaded classes.

**Survivor Space**: it is a pool of memory in the JVM that is used to store objects that have survived the garbage collection process. It is a part of the memory management process used by the JVM to manage and allocate memory resources.

**Tenured Gen**: it represents the memory pool that is used by the JVM to store objects that have been allocated memory and are no longer in use. This pool also holds objects that have been allocated long-term memory and that are not eligible for garbage collection.

**Garbage Collector:**

**GC count increase**: GC count increase in the JMX dashboard represents the number of garbage collection operations that have occurred since the Java application was started. Garbage collection is the process of reclaiming memory from objects that are no longer being used.

**GC Time rate**: GC time rate represents the rate at which the garbage collector is running on the JMX dashboard. It measures the amount of time spent in garbage collection activities as a percentage of the total amount of time the application has been running.

**GC Time increase**: GC time increase represents the amount of time spent by the Java Virtual Machine (JVM) to perform garbage collection activities.

**Threading:**

**JVM Threads count**: JVM threads count in JMX Dashboard represents the number of threads that are currently running in the Java Virtual Machine. It includes threads from the application, the JVM itself, and any other native threads that the JVM is running.

**JVM Threads count increased:** JVM thread count increased in JMX Dashboard represents the total number of threads that are running in the JVM.

**JVM Threads**:JVM Threads represent the threads being used by the JVM to execute code.

**JVM Threads increased:** JVM threads usually represent an increase in the number of threads that are being used by the JVM.

**Classes:**

**Class Loading count**: Class Loading count is a metric that is displayed in the JMX dashboard and represents the number of classes that have been loaded into the JVM. The higher the count, the more classes the JVM has loaded, which can indicate that the JVM is performing better.

**Class Loading count increase**: It is a representation of the total number of classes that have been loaded into the JVM so far. It is a measure of the performance of the JVM and helps to identify potential memory or class loading issues.

**Class Loaded**: Class Loaded is a JMX metric that tracks the total number of classes loaded in the JVM. This metric is usually used to measure the efficiency of the class loading subsystem, as well as to monitor memory usage.

**Class Loaded increase:** Class Loaded increase represents the number of classes that have been loaded by the JVM since the start of the application.

**EXTRAS:**

**Q1. Why is CPU usage always in %?**

CPU usage is always expressed as a percentage because it is a measure of how much of the total available computing power is being used at any given moment. It is expressed as a percentage because it is a relative measure rather than an absolute one, allowing us to compare the number of computing resources being used at different times.

**Q2. Why is RAM always shown in absolute value?**

RAM is always expressed in absolute values because it is a measure of the amount of memory available to the system. Unlike CPU usage, RAM is an absolute measure, meaning that the amount of RAM available will remain the same regardless of the amount of processing power being used.

**Q3. Can we find the absolute value of CPU usage?**

No, it is not possible to find an absolute value of CPU usage. CPU usage is a relative measure of how much of the total available computing power is being used at any given moment, and therefore cannot be expressed in absolute values.